République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de La Recherche Scientifique

Université Mouloud MAMMERI de Tizi-Ouzou

Faculté de Génie électrique et Informatique

Département Informatique

*Mémoire de Fin d'études*

*de Master Académique*

Spécialité: Réseaux, Mobilité et Systèmes Embarqués

Thème

# Évaluation Expérimentale de l'influence de la Configuration de Cache Mémoire sur l'exécution d'un Programme

Présenté par

LOUNAS OUIZA FENZI SAMIRA

Dirigé par M. HEMDANI

Année Universitaire 2014-2015

# Table des matières

# Table des figures

# Liste des tableaux

# Introduction générale

L'exécution d'un programme est un processus séquentiel itérant les étapes suivantes :

1. chercher et lire une instruction,
2. lire un ensemble de données d'entrée, exécuter un calcul,
3. enregistrer les résultats pour les futures références, et finalement,
4. décider quelle instruction on doit chercher après.

Ce caractère séquentiel laisse penser qu'il y a peu de gain de performance à attendre du mode d'exécution lui-même et que ce sont surtout les progrès dans les technologies des semi-conducteurs, fournissant des circuits plus rapides, qui peuvent faire exécuter chacune des étapes ci-dessus plus rapidement, le résultat étant une réduction du temps requis pour exécuter le programme tout entier. Cependant, Les systèmes modernes à haute performance utilisent les techniques qui leur permettent d'être plus concurrentiels au sujet de l'exécution du programme en faisant une meilleure utilisation de ce que la technologie à semi-conducteurs doit offrir à n'importe quel moment donné. Un ensemble de techniques utilisées par ces systèmes de calcul se fondent sur des observations empiriques au sujet du comportement de programme pour être plus efficace. La motivation sous-tendant ces techniques est que les programmes, la plupart du temps, ne se comportent pas aléatoirement. Tout ceci est mieux compris si nous considérons comme exemple le cache.

La technique de cache vise à rapprocher mieux le bloc de mémoires idéalement grand et rapide qui correspond au modèle de stockage de données typique pour les langages de programmation actuels à leurs utilisateurs. Il est bien connu que les programmes ont une tendance à l'accès au même emplacement mémoire à plusieurs reprises (localité temporelle) ou aux emplacements mémoire voisins (localité spatiale), les deux phénomènes peuvent apparaître dans le même temps. La technique de cache exploite cette observation empirique en plaçant un ensemble d'emplacements mémoire récemment consultés dans une structure de stockage petite et rapide, un cache. En conséquence, beaucoup si ce n'est la plupart des accès mémoire seront servis dans le cache, tirant bénéfice de sa basse latence (sa rapidité de réponse) et sa largeur de bande (combien de données il peut fournir en même temps). Les données placées dans le cache sont rarement référencées avant d'y être remplacées.

Vu l'importance de rôle du cache dans la gestion de la mémoire, notre étude portera sur les différents paramétres qui influence sur sa performance. Alors l'objectif est de déterminer les valeurs de ces différents paramétres qui, combinées entre eux, donneront des résultats satisfaisants lors de l'évaluation de la performance du cache

Dans ce contexte, notre travail est axé sur l'évaluation expérimentale de l'influence de la configuration de cache mémoire sur l'exécution d'un programme. Afin de bien mener cette étude, nous avons opté d'organiser notre mémoire en quatre chapitresă :

Dans le chapitre I, nous décrivons les différentes architectures de mémoire cache et leurs principales caractéristiques. Ce chapitre présente le contexte où intervient notre étude.

Le chapitre II est consacré à la description de la démarche expérimentale des architectes utilisée pour quantifier leurs propositions, basée sur une simulation appliquée à un jeu de programmes tests (benchmarks).

Puis le chapitre III présente la démarche que nous avons suivie pour la conception de notre travail et son objectifs.

Enfin le chapitre IV constitue une synthèse de notre travail par l'implémentation, la simulation de diverses configurations de mémoire cache et la présentation des résultats relevés.

# Chapitre 1

## Les Caches Mémoire

## 1.1 Introduction

Depuis de nombreuses années l'écart entre la performance du processeur et celle de la mémoire tend à s'accroître. Cet accroissement s'est accéléré avec l'avènement des processeurs RISC. Depuis le début des années 80, le principal mécanisme architectural permettant de masquer la latence de la mémoire est le cache. Les mémoires cache ont été introduites pour réduire le temps d'accès aux informations en raison de l'écart croissant entre la fréquences des microprocesseurs et la latence d'accès à la mémoire principale.[**17**]

Dans ce chapitre, nous allons rappeler des notions fondamentales concernant la mémoire et nous nous intéresserons en particulier à la mémoire cache puisque la mémoire centrale de l'ordinateur possède une vitesse bien moins importante que le processeur. Il existe néanmoins des mémoires beaucoup plus rapides, mais dont le coût est très élevé. La solution consiste donc à inclure ce type de mémoire (cache) rapide à proximité du processeur et d'y stocker temporairement les principales données devant être traitées par le processeur.

Les caches sont très efficaces pour réduire le temps d'accès moyen aux informations en exploitant les propriétés de localité spatiale et temporelle des applications. Néanmoins, l'augmentation de la taille des mémoires cache a pour effet d'augmenter également la latence d'accès aux informations. Pour pallier cette limite, les hiérarchies de mémoires cache ont été introduites. Elles disposent d'un premier niveau de cache, proche du processeur, de petite capacité avec une faible latence d'accès, permettant ainsi de répondre rapidement aux requêtes mémoire du processeur. Ce niveau est suivi d'un ou plusieurs niveaux de caches, que nous appelons niveaux inférieurs, disposant d'une plus grande capacité mais d'une latence d'accès plus importante. Ces niveaux permettent de réduire les latences d'accès aux données moins fréquemment utilisées, comparativement à un accès systématique à la mémoire principale.

Différentes structures de caches existent : les caches à correspondance directe, les caches totalement associatifs et les caches associatifs par ensembles . Pour ces types de cache, lors du chargement d'une ligne de cache, les autres lignes déjà présentes dans le même ensemble se trouvent en conflit dès lors que cet ensemble est plein. Il faut alors déterminer une ligne de cache à évincer. Pour ce faire, il existe différentes politiques de remplacement de cache et d'algorithmes de mise à jour.

## 1.2 Les types de mémoires

Les différentes mémoires peuvent être classées comme indiqué sur le schéma de la figure 1.1 ci-dessous :

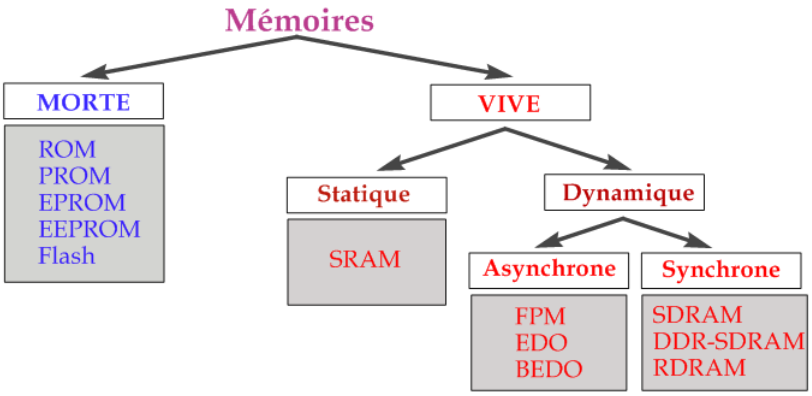



FIGURE 1.1 – Classifications des mémoires

On distingue la mémoire vive (ou RAM) et la mémoire morte (ou ROM).[**1**]

### 1.2.1 La mémoire morte (ROM)

Les mémoires mortes ou ROM (Read Only Memory) sont des mémoires non volatiles qui sont vouées à être accédées en lecture en fonctionnement normal. Elles contiennent du code et des données qui ne sont pas amenés à changer souvent. Les ROM contiennent généralement les routines d'accès de base aux périphériques.

- **ROM classiques**

L'information contenue dans ces mémoires est enregistrée de manière irréversible lors de la fabrication du circuit. Le principe de réalisation de ces mémoires est le réseau de diodes.

- **PROM**

La mémoire PROM (Programmable ROM) est une ROM à enregistrement irréversible qui peut être programmée par un utilisateur grâce à un dispositif appelé programmateur de PROM.

- **EPROM**

Les EPROM (Erasable Programmable ROM) présentent l'avantage de pouvoir être effacées et réécrites. Pour effacer une EPROM il faut la soumettre à un rayonnement ultraviolets pendant 30 minutes. Elles sont facilement reconnaissables en raison de la présence sur leur face supérieure d'une fenêtre de quartz obturée par un adhésif afin de les soustraire aux rayonnements ultraviolets naturels (soleil, néon).

- **EEPROM**

Les EEPROM (Electrically EPROM) quant à elles utilisent un procédé électrique pour l'effacement des données. Elles sont donc bien plus pratiques que les EPROM.

- **Flash**

Les mémoires Flash réalisent la synthèse des technologies EEPROM et SRAM. Elles sont rapides, ne nécessitent pas de rafraîchissement. Elle sont effaçables et reprogrammables rapidement par blocs de 64 Ko.Malheureusement son prix de revient est assez élevé et on ne peut la reprogrammer qu'un nombre limité de fois (environ 100.000). [**1**]

### 1.2.2 La mémoire vive (RAM)

La mémoire vive ou RAM (Random Access Memory) pour mémoire à accès aléatoire, est une mémoire volatile, cela signifie que si l'on coupe l'alimentation, les données qu'elle contient sont perdues. Ses caractéristiques sont les suivantes :

elle sert à stocker les programmes exécutés par le processeur elle est accessible en lecture et en écriture elle est organisée sous forme matricielle

Il existe deux grandes familles de mémoires vives : les RAM Statiques (SRAM) les RAM Dynamiques (DRAM).[**1**]

#### 1.2.2.1 La mémoire vive statique (SRAM)

Chaque bit d'une SRAM est formé par une bascule (latch) constituée par 4 à 6 transistors. L'information stockée peut être maintenue sans dégradation pendant une centaine d'heures. L'intérêt de ce type de mémoire est sa vitesse (quelques ns) mais son coût est prohibitif. En effet, une bascule est composée de deux portes NOR constituée chacune de deux transistors (soit un total de 4 transistors pour un bit d'information). En conséquence on utilisera la SRAM lorsque le facteur vitesse est critique et notamment pour des mémoires de petite taille comme la mémoire cache.

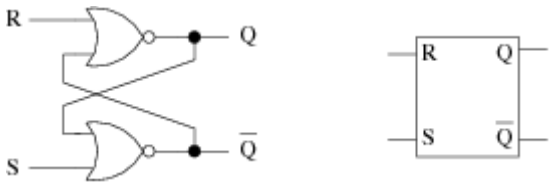



FIGURE 1.2 – Bascule RS constituée de 2 portes NOR

Les mémoires statiques sont composées de bascules. La bascule (latch ou flip-flop en anglais) est composée de deux portes NOR (ou NAND) , c'est-à-dire que la sortie de l'une est raccordée à l'entrée de l'autre.

- Le bascule RS comporte :

Deux entrées notées R (Reset) et S (Set) deux sorties désignées par Q et ¬Q qui ont normalement des valeurs inverses.

En fonctionnement normal les entrées R et S sont en position basse (ou à 0). On doit alors mettre une des entrées en position haute (1) pour changer l'état de la bascule. L'étude de la bascule RS montre que :

la mise à 1 momentanée de S force Q à 1 et ¬ Q à 0 la mise à 1 momentanée de R force Q à 0 et ¬ Q à 1 la mise à 1 momentanée de R et S provoque une situation indésirable. Cela provoque la mise à 0 de Q et ¬ Q. Lorsque S et R reviennent en position basse l'état de la sortie est déterminé par la porte qui revient à 0 en premier.

On supposera que le cas R = S = 1, qui consiste à vouloir mettre la bascule à 0 et à 1 ne se produira jamais. [**1**]

#### 1.2.2.2 La mémoire vive dynamique (DRAM)

Chaque bit d'une DRAM est constitué par un transistor et un condensateur :

le condensateur détermine la valeur du bit : le bit vaut 1 si le condensateur est chargé, il vaut 0 dans le cas contraire. le transistor gère l'accès au condensateur (voir la figure1.3).

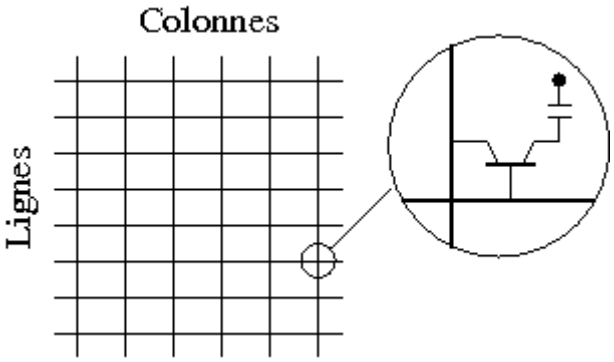



FIGURE 1.3 – Le transistor et le condensateur de la DRAM

L'inconvénient des DRAM est que le condensateur possède une tendance naturelle à se décharger. Pour que l'information reste cohérente, on va devoir réaliser un rafraîchissement de la mémoire toutes les quelques millisecondes. Ce rafraîchissement consiste à lire et à réécrire la donnée.

- Les avantages de la DRAM sont :

sa grande densité d'intégration (car un bit est représenté par un transistor), son faible coût de fabrication, sa faible consommation électrique (entre un sixième à la moitié de

celle des SRAM).

En revanche le temps d'accès de la DRAM est plus important que celui de la SRAM (environ 50-70 ns).

- **Mémoires synchrones et asynchrones**

**Mémoire asynchrone** : pour ce type de mémoire, l'intervalle de temps entre deux accès mémoire consécutif n'est pas régulier. Le processeur ne sait donc pas quand l'information qu'il attend est disponible et doit attendre (wait-state) que la mémoire lui transmette les données.

- RAM FPM (asynchrone)

la mémoire FPM ( Fast Page Mode) fut utilisée sur les processeurs 80386 et 80486 d'Intel. Elle permet d'améliorer la vitesse de transfert des données en accélérant le transfert de données contiguës.

- RAM EDO (asynchrone)

La mémoire EDO (Extended Data Out) est technologiquement identique à la FPM mais consiste en une amélioration logicielle qui permet l'envoi de données pendant le décodage de la prochaine adresse mémoire à accéder.

**Mémoire synchrone** : la cadence de sortie des informations est régulière, on évite ainsi les états d'attente (wait state) du processeur.

- Mémoire SDRAM (synchrone)

La mémoire SDRAM (Synchronous DRAM) est capable d'échanger des données en se synchronisant avec le signal d'horloge pour des fréquences de 66, 100 ou 133 Mhz. La mémoire est découpée en deux bancs ce qui permet d'accéder à l'un pendant la configuration du second. Cette mémoire est utilisable en mode rafale 2,4,8 ou 512 octets.

- Mémoire DDR-SDRAM

La Double Data Rate SDRAM est un standard mis au point par Nec, Samsung et Toshiba. Elle double le taux de transfert des actuelles DRAM en permettant de lire les données sur le front montant et sur le front descendant du signal d'horloge. La DDR SDRAM fonctionne à 100, 133, 150, 166, 200 ou 233 Mhz et est capable d'atteindre des vitesses de transfert de l'order de 1.6 à 4,2 Go/s .

- Comparaison SRAM / DRAM

Étant donné les caractéristiques des SRAM et DRAM, on peut en déduire les propriétés suivantes :

les DRAM sont plus lentes que les SRAM car durant le rafraîchissement on ne peut accéder aux données cependant, les DRAM ont une densité d'intégration plus grande que les SRAM : en effet le coupe transistor + condensateur occupe moins de place que les 4 à 6 transistors des SRAM

En résumé :

la SRAM est rapide mais chère la DRAM est lente mais bon marché. [**1**]

## 1.3 La mémoire cache

Un cache est une petite mémoire, 10 à 25 fois plus rapide qu'une mémoire conventionnelle. Il conserve une copie des informations récemment et couramment utilisées par les programmes. Les informations présentes dans le cache sont accédées plus rapidement que si elles étaient en mémoire principale.[**8**]

Mémoire cache est la traduction littérale de l'expression anglaise cache Memory, qui vient elle même de mémoire cachée, principe inventé à Grenoble dans les années 1960, l'académie française propose plutôt le terme antémémoire qui est une mémoire de taille réduite mais à accès très rapide , située entre le processeur et la mémoire centrale contenant une partie du code et des données manipulées par le processeur. Cette mémoire est organisée de façon à ce que le processeur fasse référence à elle la plupart du temps.[**2**]

Le cache est composé de deux parties : un répertoire (Directory) et une mémoire à accès aléatoire (RAM). La partie mémoire est divisée en un nombre de lignes de tailles identiques pouvant stocker chacune un bloc mémoire de plusieurs mots. Le répertoire, qui généralement est organisé en associativité totale, est constitué de blocs composés d'une partie index, qui contient une partie de l'adresse mémoire d'une ligne présente dans le cache, et d'un certain nombre de bits de contrôle indiquant l'état de la ligne et assurant la cohérence des données du cache avec celles en mémoire. Le répertoire contient une entrée pour chaque ligne du cache.[**8**]

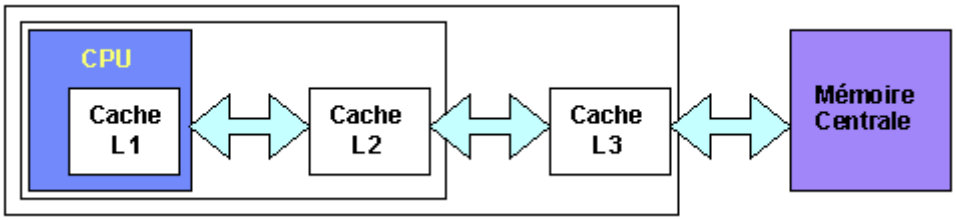



FIGURE 1.4 – Architecture de cache mémoire

On distingue en fonction de leur taille et de leur localité (la figure 1.4) : [**9**]

1. la mémoire cache de premier niveau (L1) située à l'intérieur du processeur.
2. la mémoire cache de second niveau (L2) située dans le boîtier du processeur .
3. la mémoire cache de troisième niveau (L3) située à présent au niveau du processeur.

## 1.4 Fonctionnement d'une cache

Le cache contient une copie des données originelles lorsqu'elles sont coûteuses ( en terme de temps d'accès ) à récupérer ou à calculer par rapport au temps d'accès au cache. Une fois les données stockées dans le cache, l'utilisation future de ces données peut être réalisée en accédant à la copie en cache plutôt qu'en récupérant ou recalculant les données, ce qui abaisse le temps d'accès moyen.

Le processus fonctionne ainsi :

1. L'élément demandeur (microprocesseur) demande une information.
2. Le cache vérifie s'il possède cette information. S'il la possède, il la retransmet à l'élément demandeur, on parle alors de succès de cache. S'il ne la possède pas il la demande à l'élément fournisseur (mémoire principale), on parle alors de défaut de cache.
3. L'élément fournisseur traite la demande et renvoie la réponse au cache.
4. Le cache la stocke pour utilisation ultérieure et la retransmet à l'élément demandeur. [**3**]

### 1.4.1 Principe de localité

L'enchaînement des références faites par un programme lors de son exécution dépend de la manière dont celui-ci est conçu et compilé. Il a été observé que les programmes ont

tendance à répéter souvent les mêmes opérations. Ainsi, il est unanimement admis qu'un programme passe 90% de son temps à exécuter 10% de son code.

La théorie du cache est basée sur le principe de la localité. La première notion, celle de localité spatiale

- **Localité Spatiale :** Les données voisines de la données actuellement utilisées seront probablement bientôt utilisées. Il faut charger de la DRAM non seulement la donnée dont on a besoin mais les données voisines : un bloc de plusieurs données (exemple 4 mots mémoire).

***Exemple*** : les tableaux et l'exécution séquentielle des instructions.

la deuxième, celle de localité temporelle

- **Localité temporelle :**Les données récemment utilisées (référencées) par le processeur seront probablement réutilisées bientôt.Il faut retenir (stocker) une donnée qui a été récemment utilisée.

***Exemple*** : les boucles.

Ainsi, afin de déterminer automatiquement quelles sont les données à conserver, le cache utilise les propriétés de localité des applications. Si une zone mémoire est référencée, il est probable quelle même ou une zone mémoire proche sera référencée dans un future très proche.

La volonté d'exploiter la localité des références affecte de façon non négligeable la conception d'une hiérarchie mémoire. L'exploitation de la localité temporelle justifie l'utilisation de l'algorithme de remplacement LRU (Least Recently Used) qui consiste à évacuer le bloc le moins récemment référencé. De même, la localité spatiale est susceptible d'influencer la granularité utilisée dans le système de hiérarchie mémoire.

Il existe plusieurs manières de profiter de la localité des programmes, les méthodes logicielles sont les plus utilisées. Elles consistent soit à améliorer la localité des références mémoires en transformant le code de l'application pour changer l'ordre des références, cela peut se faire au niveau de la compilation, ou bien à modifier le placement des données en mémoire, ce qui change directement l'ordre des accès mémoire et donc la localité des programmes. [8]

### 1.4.2 Défauts de cache

Il existe trois types de défauts de cache en système mono processeur et quatre dans les environnements multiprocesseurs :

- **Les défauts obligatoire (Cold Cache)** : ils correspondent à la première demande du processeur pour une donnée/instruction spécifique et ne peuvent être évités.
- **Les défauts de capacité (Capacity miss) :** l'ensemble des données nécessaires au programme excèdent la taille du cache, qui ne peut donc pas contenir toutes les données nécessaires.
- **Les défauts de conflits (Conflict miss) :** deux adresses distinctes de la mémoire de niveau supérieur sont enregistrés au même endroit dans le cache et s'évincent mutuellement, créant ainsi des défauts de cache.
- **Les défauts de cohérences (Cache Coherency)** : ils sont dus à l'invalidation de lignes de la mémoire cache afin de conserver la cohérence entre les différents caches des processeurs d'un système multi- processeurs. [**3**]

## 1.5 Les niveaux de caches

Entre processeur et mémoire généralement il existe 3 niveaux de cache. Les caches se sont rapprochés progressivement du die du processeur à mesure que la finesse de gravure diminuait et permettait de placer plus de transistors sur le die. Initialement la mémoire cache se trouvait sur la carte mère. [**1**]

### 1.5.1 Cache de niveau 1

C'est une mémoire intégrée directement dans le microprocesseur qui permet d'accélérer les communications entre un microprocesseur et un composant servant à stocker les données (RAM,disque dur). Sa taille est essentiellement fonction de l'architecture du microprocesseur.AMD a toujours privilégié des caches L1 de grande taille, 64 Ko pour le K6 et 128 Ko pour l'Athlon, tandis que Intel privilégie des caches L1 de petite taille, 32 Ko pour les PentiumII/III. Elle est scindée en deux parties de taille égale. L'une stocke les instructions des programmes, l'autre les données des programmes. Les autres caches ne font pas cette distinction. [**1**]

### 1.5.2 Cache de niveau 2

Mémoire cache qui sert d'intermédiaire entre cache mémoire de niveau 1 et la mémoire RAM. Elle ne différencie pas données et programmes, elle est moins rapide que le cache L1, mais sa taille est plus importante (de 256Ko à 2 Mo).

Ce cache suit une stratégie Least Recently Used, contrairement au cache L1. Les données récemment utilisées y sont conservées. Ce cache est garanti comme étant toujours cohérent pour tous les multiprocesseurs. Les threads peuvent donc modifier une variable en cache L2, ce changement sera immédiatement visible pour tous les autres threads. Les

opérations atomiques peuvent s'effectuer directement en cache L2 et n'invalident pas le cache. [**1**]

### 1.5.3 Cache de niveau 3

Type de cache qui est aujourd'hui courant sur des machines haut de gamme, et composé de mémoire SRAM implantée directement sur la carte mère. Sa taille varie de 1 Mo à 8 Mo. [**1**]

### 1.5.4 Cache exclusif et inclusif

Cela désigne la manière dont vont coopérer les caches entre eux, en particulier les caches L1 et L2.

**Le cache inclusif** est la méthode la plus ancienne et la plus courante. Elle était utilisée sur tous les microprocesseurs jusqu'à l'arrivée des Duron et des Athlon Thunderbirds d'AMD. Lorsqu'une donnée part de la RAM vers la CPU, elle passe par le cache L2 puis par le cache L1. Une donnée peut donc être stockée, au même instant, à la fois dans le cache L1 et dans le cache L2. Il peut y avoir une redondance d'informations.

**Le cache exclusif** est apparu avec les Duron et les Athlon Thunderbirds. Les caches L1 et L2 vont fonctionner comme s'il y'avait qu'un seul cache. Une donnée ne peut être au même instant dans les deux caches. Cependant, l'accès est plus long que dans le cadre des méthodes de cache inclusif. Cela est d'autant plus vrai que le cache est grand. [**1**]

## 1.6 Organisations du cache

La mémoire cache ne dispose pas l'espace d'adressage propre car elle est sensée contenir un fragment de code et de données provenant de n'importe quelle zone mémoire. Il existe plusieurs façons d'organiser l'espace du cache pour contenir ces données. Dans tous les cas, le processeur référence le cache avec l'adresse mémoire de l'emplacement demandé. Le système cache doit pouvoir réaliser une correspondance entre l'espace du cache et l'espace mémoire. Plusieurs stratégies sont possibles.[**8**]

### 1.6.1 Mémoire cache à correspondance directe(directe mapped)

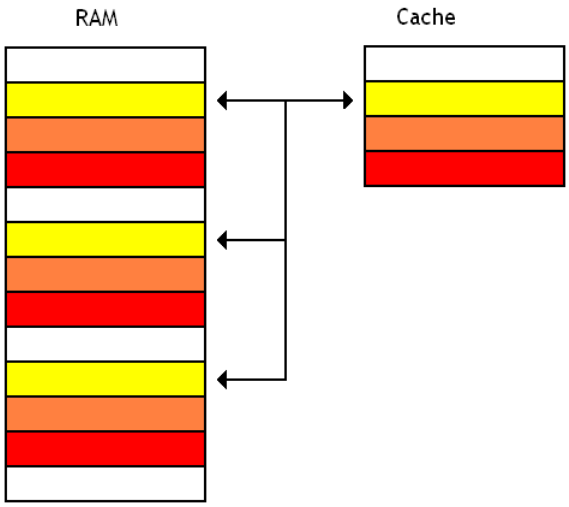



FIGURE 1.5 – Mémoire cache directe

Avec ce genre de cache, le contenu d'une adresse mémoire sera chargée dans une ligne de cache prédéfinie et impossible à modifier : ce sera toujours la même, quelques soient les circonstances (la figure 1.5)

Dans cette organisation, le cache est composé de mémoires SRAM rapides et d'un répertoire. Chaque bloc du cache permet de sauvegarder un blocs mémoire déterminé par les bits de poids faibles de l'adresse. Les bits de poids forts restant sont sauvegardés dans le cache (le répertoire) pour identifier le bloc mémoire qui est actuellement sauvegardé dans le cache.[**8**]

Si la mémoire cache est décomposée en P blocs, la mémoire centrale est composée de N blocs de même taille. Chaque bloc mémoire i est placé dans un bloc cache bien défini qui est le bloc d'adresse (i Mod p) L'adresse générée par le processeur est donc divisée en deux champs étiquette(TAG) et INDEX. Le cache utilise le champ INDEX pour accéder à l'entrée associée du répertoire puis compare le contenu de cette entrée au champs étiquette de l'adresse. si les deux champs correspondent, le bloc est effectivement dans le cache succès (Hit) et l'opération est effectuée sur ce dernier. Si non un défaut (Miss), l'opération est effectuée sur la mémoire et le bloc référencé est mis dans le cache et le répertoire mis à jour comme le montre la figure 1.6 [**8**]

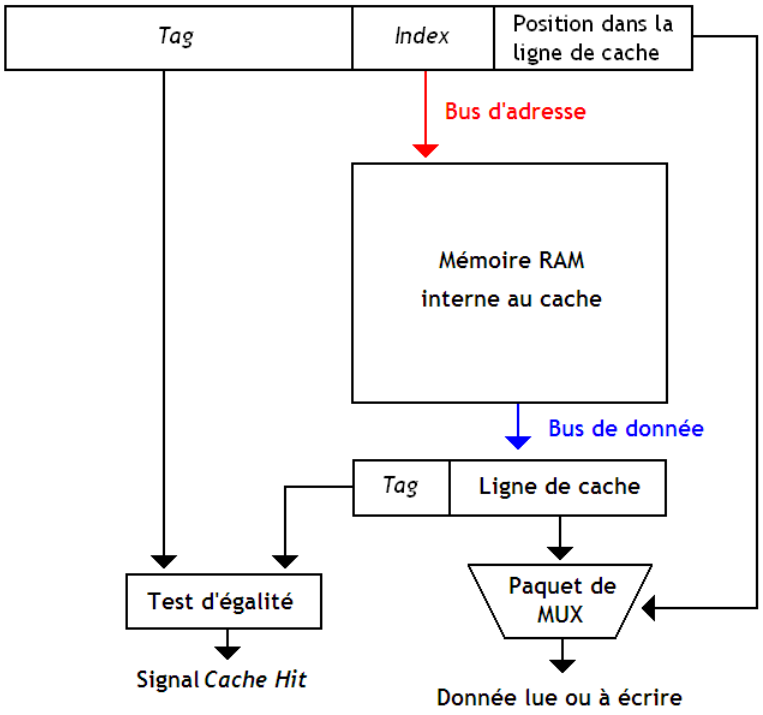



FIGURE 1.6 – Implémentation d'une mémoire cache directe

Les blocs mémoire ayant le même index étant associés au même bloc cache, il ne peut y avoir à un moment donné que des blocs d'indexes différents dans le cache. Cette situation engendre beaucoup de conflits quand un programme emploie, par exemple, deux vecteurs sauvegardés dans des zones mémoires ayants le même index. La compilation et l'édition des liens des programmes peut réduire ces conflits en veillant à ne pas affecter des adresses ayant le même indexe à ce genre de données. Cette stratégie est avantageuse dans la mesure où son implémentation est simple et ne nécessite pas d'algorithme de remplacement. Ce qui augmente la vitesse des opérations du cache. Cependant , ces performance sont limitées dans le cas de fortes références vers des blocs différents ayants le même indexe. Le taux de succès de cette stratégie est inférieur à celui de l'associativité totale (Fully associative) mais au fur et à mesure que la taille du cache augmente, le Hit ratio augmente et la différence devient insignifiante.[**5**]

### 1.6.2 Mémoire cache complètement associative(full associative)

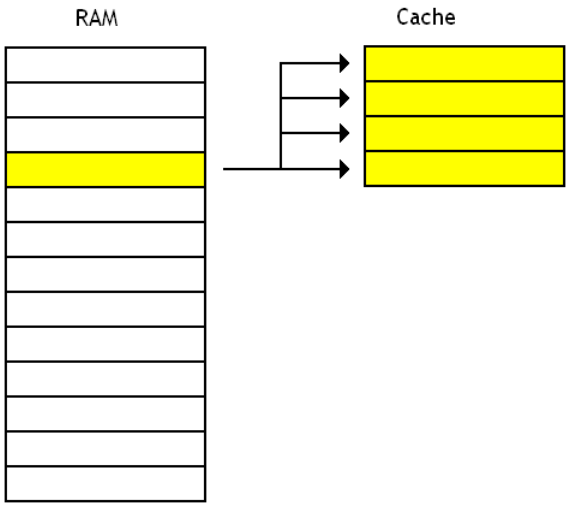



FIGURE 1.7 – Mémoire cache complètement associative

Avec les caches complètement associatif, toute donnée chargée depuis la mémoire peut être placée dans n'importe quelle ligne de cache, sans aucune restriction(figure 1.7).[**5**]

Chaque ligne de la mémoire de niveau supérieur peut être écrite à n'importe quelle adresse de la mémoire cache.

Cette méthode requiert beaucoup de logique car elle donne accès à de nombreuses possibilités. Ceci explique pourquoi l'associativité complète n'est utilisée que dans les mémoires cache de petite taille.[**3**]

Dans l'organisation associative, l'indexe présent dans l'adresse est comparé avec tous les autres indexes dans le cache. Ce schéma offre une grande flexibilité mais peut consommer énormément de temps si les indexes étaient comparés séquentiellement.

L'avantage de l'organisation associative est de pouvoir utiliser un algorithme de remplacement plus efficace en évitant les problèmes de placement, car chaque bloc mémoire peut être mis dans n'importe quel ligne de cache.[**8**]

**Implémentation**

La figure 1.8 donne un schéma synoptique pour le fonctionnement de cette organisation. D'abord, on doit déterminer si on a un défauts de cache ou un succès de cache, Vu que notre donnée peut être placée dans n'importe quelle ligne de cache, il faut vérifier les étiquettes (Tags) de toutes les lignes de cache. Chaque étiquette est donc relié à un comparateur, qui vérifiera l'égalité entre le étiquette de la ligne de cache, et le Tag extrait de l'adresse accédée en lecture ou en écriture. Si un seul de ces comparateur renvoie "Vrai" (en clair, 1), alors on a un succès de Cache . Sinon, c'est un défaut de cache. Le signal qui indique l'apparition d'un succès de Cache est donc un ET logique entre toutes les sorties des comparateurs.

Toutes nos lignes de caches sont reliées à un bus interne qui permettra de relier nos lignes de cache à l'extérieur. Si les deux étiquettes (Tags) sont identiques, cela signifie que

la ligne de cache associée est la bonne, et que c'est celle-ci qu'on doit lire ou écrire. Il faut donc la connecter à ce bus, et déconnecter les autres. Pour cela, on relie donc la sortie du comparateur à des transistors chargés de connecter ou de connecter les lignes de cache sur notre bus .

Ensuite, il ne reste plus qu'à sélectionner la portion de la ligne de cache qui nous intéresse, grâce à un paquet de multiplexeurs, comme pour les caches à correspondance directe.de [**5**]

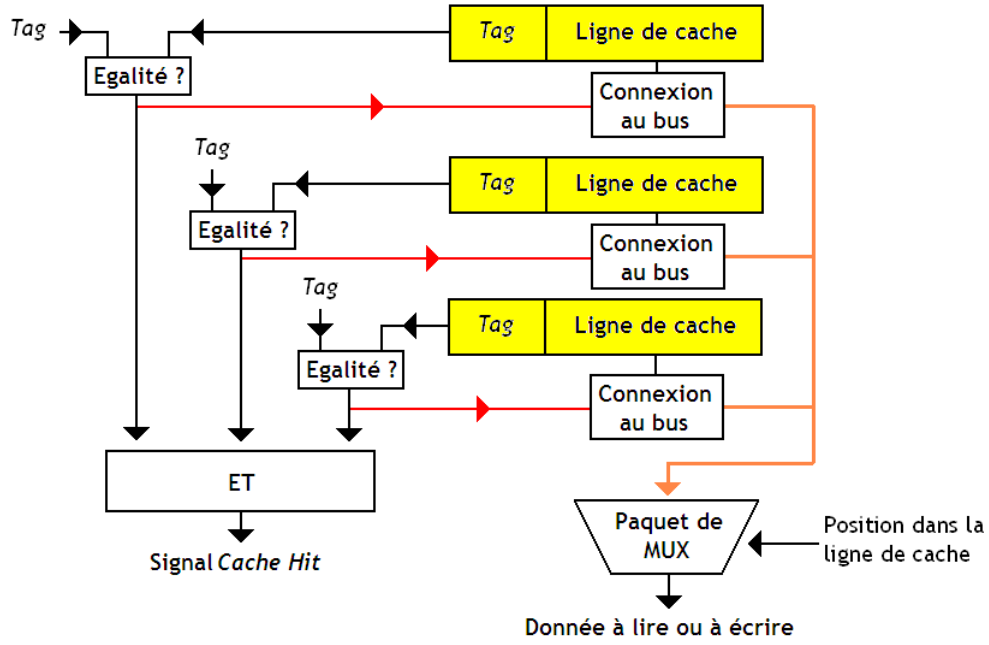



FIGURE 1.8 – Implémentation d'un cache mémoire completement asociative

### 1.6.3 Mémoire cache associative par ensemble (set associative)

Cette organisation représente un compromis entre l'organisation directe et l'organisation associative. C'est l'organisation la plus utilisée dans la conception des systèmes de mémoires. La figure 1.9 donne un schéma d'une organisation associative avec des ensembles de taille de trois lignes.[**8**]

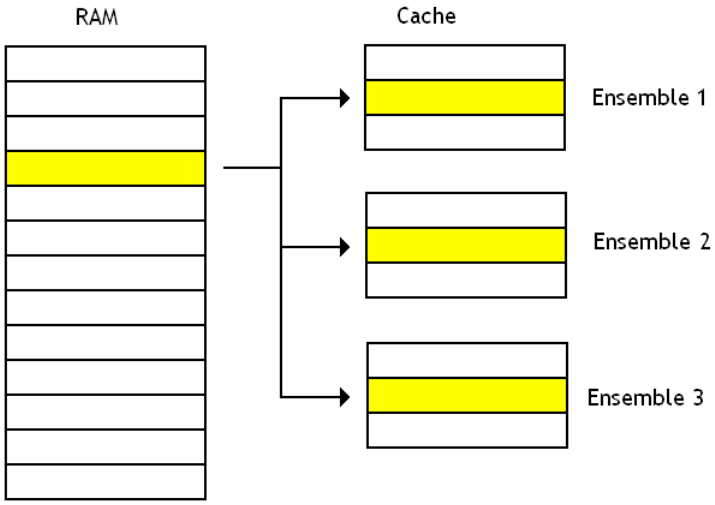



FIGURE 1.9 – Mémoire cache associative par ensemble

Pour simplifier, ces caches sont composés de plusieurs ensembles de lignes de caches. On peut décider ce choisir dans quel bloc une donnée peut être chargée, mais pas la ligne de cache exacte dans laquelle charger notre donnée : les lignes de cache à l'intérieur d'un ensemble sont associées à des adresses mémoires en utilisant la stratégie correspondance directe.[5]

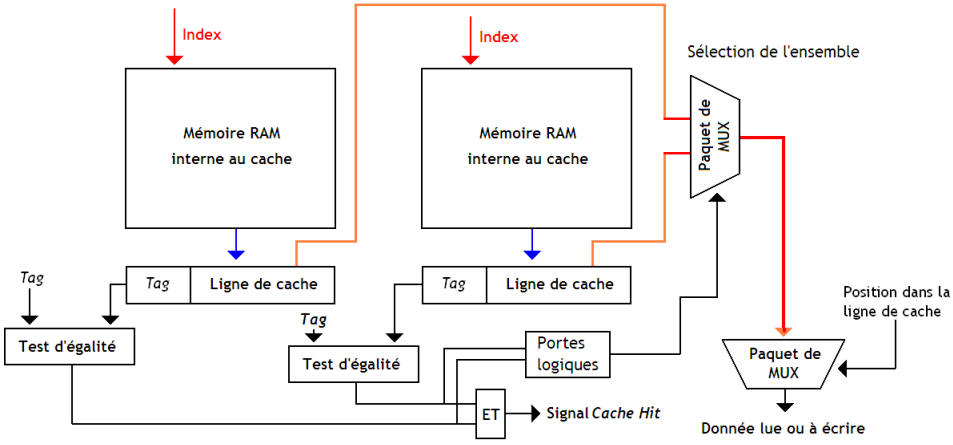



FIGURE 1.10 – Implémentation d'un cache mémoire associative par ensembles

L'organisation est identique à celle d'un cache complètement Associative, à part que chaque ensemble étiquitte-Ligne de cache est remplacé par une mémoire RAM qui en contient plusieurs. Et pour sélectionner une donnée dans cette RAM comme la figure 1.10 montre, il suffit d'envoyer son Index dans le bus d'adresse de toutes ces mémoires RAM, et de récupérer les couples étiquitte-Ligne de cache. On vérifie alors les étiquettes de ces couples, et si jamais il y a correspondance avec l'étiquette de l'adresse à lire ou écrire, on connecte la ligne de cache (celle reçue en sortie de notre mémoire RAM) sur le bus relié au MUX. Et ensuite, le MUX sélectionne la donnée voulue dans notre ligne.[5]

### 1.6.4 Mémoire cache par secteur(sector mapping)

Dans cette organisation, la mémoire est partitionné en un ensemble de secteurs. Chaque secteur est composé de plusieurs blocs. Le cache est aussi partitionné en secteurs et blocs de même taille. Chaque secteur de la mémoire peut être associé à n'importe quel secteur cache et le tag est sauvegardé dans le cache pour identifier le secteur mémoire associé au secteur cache. toutefois quand un défaut de cache se produit, (miss), le secteur n'est pas totalement chargé mais uniquement le bloc concerné et les autres blocs (de l'ancien secteur) sont invalidés en mettant à jour un bit dit bit de validité. [**8**]

## 1.7 Caches unifiés ou caches séparés

Pour fonctionner, un processeur a besoin de données et d'instructions. Il existe donc deux solutions pour l'implémentation des mémoires cache :

**Cache unifié** : données et instructions sont enregistrées dans la même mémoire cache,

**Les caches séparés :** sont organisé en deux cache distincts, un pour les données et un autre pour les instructions.

Séparer données et instructions permet notamment d'augmenter la fréquence de fonctionnement du processeur, qui peut ainsi accéder simultanément à une donnée et une instruction.C'est notamment la configuration retenue par Intel pour les caches de niveau 1. [**3**]

## 1.8 Stratégie de remplacement

L'algorithme de remplacement a pour rôle de déterminer la ligne à remplacer. Le chargement d'une ligne dans le cache se fait au détriment d'une ligne déjà présente si cette dernière a été modifiée, elle est mise à jour au niveau de la mémoire. Dans le cas contraire elle est tout simplement écrasée.

La politique de remplacement est très influencée par la localité spatiale ou temporelle des références. Les lignes contenues dans le cache n'ont pas la même importance, certaines seront référencées à nouveau pendant qu'elles sont dans le cache et d'autres non.

A chaque moment, le cache contient deux types de lignes, des lignes vivantes et des lignes mortes. Les lignes vivantes sont importantes pour le cache car leur référence crée un succès. Quant aux lignes mortes, elles ne seront plus accédées durant leur présence dans le cache, elles sont inutiles et créent une pollution du cache.

Il est à noter que l'organisation directe n'a pas besoin d'une méthode de remplacement spécifique, une ligne mémoire ne peut être mise que dans une seule ligne du cache, donc il n'existe qu'une seule possibilité de remplacement.

Divers stratégies sont employées, principalement :

- choisir un bloc candidat de manière aléatoire
- choisir le plus ancien bloc du cache (FIFO, First In First Out)
- choisir le bloc le moins récemment utilisé (LRU Least Recently Used)
- choisir le bloc le moins fréquemment utilisé (LFU Least Frequently Used)

Les stratégie concernant l'utilisation (LFU, LRU) sont les plus efficace (vient ensuite la stratégie aléatoire). Les stratégies aléatoire et FIFO sont plus faciles à implanter. [**5**]

- **Politique de remplacement Random (aléatoire)**

C'est l'algorithme de remplacement le plus simple à implémenter. Le choix de la ligne à remplacer se fait d'une façon complètement aléatoire sans faire attention aux références. Son implémentation peut être obtenue par un compteur cyclique qui sera incrémenté à chaque référence et qui donnera à chaque fois que c'est nécessaire la ligne à remplacer. [**8**]

- **Politique FIFO(First In First Out)**

Dans l'algorithme FIFO les lignes de la mémoire cache sont effacées dans l'ordre où elles sont arrivées dans la mémoire cache, utilisant ainsi l e principe de localité de la manière la plus simple possible. Cette simplicité de conception, qui lui a valu un franc succès dans l'industrie, est malheureusement obtenue au détriment de l'efficacité. Ainsi, selon Smith, le nombre de défauts de cache obtenus par cet algorithme est entre 12 et 20 % plus important que pour le LRU.

Pour implémenter cet algorithme, la solution la plus simple permet de se contenter d'un simple compteur. Prenons un cache Fully Associative : dans celui-ci, une nouvelle donnée peut aller n'importe où dans le cache. On peut profiter de cette propriété pour insérer les données dans le cache les unes à la suite des autres.

La ligne de cache la plus ancienne sera localisée à un certain endroit, et la plus récente sera localisée juste avant. Il nous suffit juste de se souvenir de la localisation de la donnée la plus ancienne, et le tour est joué. Cela peut se faire avec un compteur unique pour tout le cache si celui-ci est Fully Asscoaitive. Pour un cache Set Associative, on doit avoir un compteur par ensemble. [**9**]

- **Politique LFU(Least Frequently Used)**

LFU quant à lui garde trace de la fréquence d'accès de ces lignes et remplace la moins fréquemment utilisée. Le point faible de cet algorithme est une pollution du cache. En effet, des lignes qui ont été accédées très fréquemment mais qui ne sont plus utilisées dans le futur tendent à rester dans la mémoire cache.

La méthode la plus simple d'utiliser un algorithme LFU est d'attribuer un compteur pour chaque bloc qui est chargé dans la mémoire cache. Chaque fois il est fait référence à ce que le bloc compteur est augmenté d'un. Lorsque le cache atteint sa capacité et dispose d'un nouveau bloc en attente d'être inséré le système va rechercher pour le bloc avec le compteur le plus bas et le retirer de la mémoire cache. [**9**]

- **Politique LRU(Least Recently Used)**

L'algorithme de remplacement LRU est le plus utilisé dans la gestion mémoire.la donnée remplacée est celle qui a été utilisée le moins récemment. Cet algorithme se base sur le principe de localité temporelle, il est implémenté en utilisant des compteurs associés à chaque ligne du cache.

Implémenter cet algorithme LRU peut se faire de différentes manières. Dans tous les cas, ces techniques sont basées sur un même principe : les circuits reliés au cache doivent enregistrer les accès au cache pour en déduire la ligne la moins récemment accédée. Évidemment, mémoriser l'historique des accès au cache ne se fait pas sans matériel adapté, matériel qui prendra de la place et des transistors.

un autre par compteur qui est incrémenté à intervalles réguliers et remis à zéro dès que la ligne est référencée. A tout moment, la valeur de chaque compteur indique l'âge de la ligne depuis sa dernière référence. La ligne LRU est celle ayant la valeur de compteur la plus élevée. [**9**]

- **MRU (Most Recently Used)**

Avec l'algorithme MRU, La donnée qui est effacée est celle qui a été utilisée le plus récemment. Cet algorithme s'implémente simplement avec un registre, dans lequel on place le numéro de la dernière ligne de cache utilisée.

Cet algorithme de remplacement est très utile quand un programme traverse des tableaux du premier élément jusqu'au dernier. Dans ces conditions, les données du tableau sont rarement réutilisées, rendant le cache inutile. En utilisant la politique MRU, notre cache ne se remplira pas avec ces données inutiles et conservera ses données utiles. Il est prouvé que dans ces conditions, l'algorithme MRU est optimal. Mais dans toutes les autres conditions, cet algorithme a des performances assez misérables.[**5**]

- **Approximations de l'algorithme LRU**

L'implémentation de LRU qui consiste a choisir la ligne de cache la moins récemment utilisée demande des circuits contenant beaucoup de transistors.

Pour résoudre ce problème, nos processeurs implémentent des variantes du LRU, moins

coûteuses en transistors, mais qui ne sont pas exactement du LRU : ils donnent un résultat assez semblable au LRU, mais un peu plus approximatif. En clair, ils ne sélectionnent pas toujours la ligne de cache la moins récemment utilisée, mais une ligne de cache parmi les moins récemment utilisée. [**5**]

- **Splitted LRU**

L'algorithme consiste à choisir le bloc de cache le moins récemment utilisé(s'il est Set Associative), et de choisir aléatoirement une ligne de cache dans ce bloc. Pour implémenter cet algorithme, il nous suffit d'une simple bascule qui mémorise le bloc la moins récemment utilisé, et d'un circuit qui choisit aléatoirement une ligne de cache dans ce bloc. On peut aussi généraliser cette technique avec plus de deux blocs de cache : il suffit juste de rajouter quelques circuits. Dans ce cas, cette technique s'adapte particulièrement bien avec des cache N-ways associatifs : il suffit d'utiliser autant de morceaux que de voies, de sous-caches direct-mapped. En effet, avec un cache direct-mapped, on a juste à savoir quelle est la voie la moins utilisée, sans se préoccuper de la ligne exacte : la ligne dans laquelle écrire dépendra uniquement de l'adresse de la donnée chargée, en raison du caractère direct-mapped de la voie. [**5**]

- **PLRUm**

Autre algorithme, un peu plus efficace : le Pseudo LRU de type m. Cet algorithme est assez simple : à chaque ligne de cache, on attribue un bit. Ce bit sert à indiquer de façon approximative si la ligne de cache associée est une candidate pour un remplacement ou non. Si ce bit est à 1, cela veut dire : attention, cette ligne n'est pas une candidate pour un remplacement. Inversement, si ce bit est à zéro, la ligne peut potentiellement être choisie pour laisser la place aux jeunes.

Lorsque l'on lit ou écrit dans une ligne de cache, ce bit est mit à 1, au fil du temps, toutes les lignes de cache finiront par avoir leur bit à 1. Aussi, l'algorithme permet de remettre les pendules à l'heure. Si tous les bits sont à 1, on les remet tous à zéro, sauf pour la dernière ligne de cache accédée. [**5**]

- **PLURt(Arbre de décision binaire)**

La première approximation est fondée sur un arbre de décision binaire. Elle ne requiert que N-1 bits par ensemble dans un cache associatif de N voies. Ces différents bits pointent la ligne considérée comme pseudo-LRU. Les bits de l'arbre de décision binaire pointant sur la voie hitée sont inversés pour protéger cette ligne de l'éviction. [**5**]

- **LRU amélioré**

L'algorithme LRU, ainsi que ses variantes approximatives, sont très efficaces dans la majorité des programmes. Du moment que notre programme respecte relativement bien la localité temporelle, cet algorithme donnera de très bons résultats : le Hit Ratio du cache sera très élevé. Par contre, cet algorithme se comporte assez mal dans certaines circonstances, et notamment quand on traverse des tableaux. Dans ces conditions, on n'a pas la moindre localité temporelle, mais une bonne localité spatiale. Pour résoudre ce petit inconvénient, des variantes du LRU existent.

Une première de ces variante, l'algorithme 2Q(L'idée de cet algorithme est de créer deux queues de taille fixe afin d'éviter la pollution de la mémoire cache), utilise deux caches, voire coupe le cache en deux. Le premier cache sert pour les données accédées une seule fois, et utilise une politique de remplacement de type FIFO. Le second fonctionne en LRU. Évidemment, les données lues une seconde dois doivent migrer du cache FIFO vers le cache LRU : cela consomme du hardware, de l'énergie et des cycles d'horloge. Les processeurs n'utilisent donc pas cette technique, mais celle-ci est utilisée dans les caches de disque dur.

D'autres variantes du LRU combinent plusieurs algorithmes à la fois et vont choisir lequel de ces algorithmes est le plus adapté à la situation. Notre cache pourra ainsi détecter si il vaut mieux utiliser du MRU, du LRU, ou du LFU suivant la situation. [**5**]

## 1.9 Différentes politiques de gestion de la hiérarchie mémoire

- ***Politique de gestion non-inclusive***

Lors d'un défaut de cache, dans les hiérarchies de caches non-inclusives, la ligne de cache contenant l'information demandée est systématiquement chargée dans tous les niveaux où le défaut se produit. Dans ce type de hiérarchie, bien qu'il n'y ait pas de mécanisme pour forcer l'inclusion, la plupart des informations contenues dans le premier niveau sont également contenues dans les niveaux de caches inférieurs.

- ***Politique de gestion inclusive***

Dans les hiérarchies de caches inclusives, l'ensemble des informations contenues dans un niveau de cache est également contenu dans chacun des niveaux de caches inférieurs. Lors d'un défaut de cache, le comportement est similaire à celui des hiérarchies de caches non-inclusives, la ligne de cache contenant l'information demandée est chargée dans tous les

niveaux où le défaut se produit. Par contre, en cas d'éviction d'une ligne de cache dans un niveau de cache, la ligne est invalidée dans tous les niveaux supérieurs pour forcer l'inclusion. L'intérêt de ce type de hiérarchie est de simplifier le maintien de la cohérence des données. Par exemple dans un système multiprocesseurs, lorsqu'une ligne de cache doit être supprimée, il suffit de l'invalider dans le dernier niveau de cache, les autres niveaux étant traités par le mécanisme forçant l'inclusion.

***Politique de gestion exclusive*** : Dans les hiérarchies de caches exclusives, l'ensemble des informations contenues dans la hiérarchie est stocké dans un seul niveau de cache. Cette gestion permet d'éviter la duplication d'information dans les différents niveaux de caches et ainsi d'augmenter virtuellement la capacité de stockage de la hiérarchie. De ce fait, les hiérarchies de caches exclusives peuvent stocker plus d'informations que les hiérarchies inclusives et non-inclusives.

Ces différentes politiques de gestion sont couramment utilisées dans les processeurs généralistes. Citons par exemple les processeurs Intel Pentium II, III, IV qui mettent en ¡uvre une hiérarchie de caches non-inclusive ou les processeurs AMD Athlon qui mettent en ¡uvre une hiérarchie de caches exclusive. D'autres processeurs encore utilisent différentes politiques de gestion le long de la hiérarchie. Ainsi, le processeur IBM Power5 utilise une politique de gestion inclusive entre les deux premiers niveaux tandis que le dernier niveau dispose d'une politique de gestion exclusive avec les deux premiers niveaux. [**1**]

## 1.10 Types de caches mémoires

La conception d'un cache reste délicate, celle ci est basée sur des critères d'efficacité et de performances élevés, beaucoup de paramètres dépendent de son environnement d'utilisation, du type d'applications utilisées et de son emplacement dans la hiérarchie mémoire. Toutes ces contraintes sont à l'origine de la conception de différents types de caches mieux intégrés à leur environnement d'utilisation. Nous citerons parmi eux : les multicaches utilisés dans les systèmes multiprocesseurs, les caches données instructions séparés, les caches données instructions unifiés et semi unifiés, les caches mixtes, les caches utilisateur/supérviseur utilisés essentiellement dans les caches processeurs internes et externes. Enfin les caches multiniveaux dont le cache victime et le cache externe sont des exemples. [**8**]

- **Cache non bloquant**

L'intérêt majeur d'un cache est qu'il doit répondre rapidement à une requête. Les données d'un cache non bloquant continuent à être accédées même si un défaut de cache est en cours de traitement. Sur un cache bloquant, le cache n'est plus accessible après un défaut tant

que la donnée n'est pas revenue de la mémoire. L'exécution de toutes les instructions dans le microprocesseur s'arrête rapidement.Un cache non bloquant peut supporter plusieurs défauts de cache et les résoudre de manière pipelinée. [**8**]

- **Le cache victime**

Dans le but d'augmenter le taux de succès, un cache victime peut être ajouté au cache principal. Celui-ci est utilisé comme second niveau de cache. Lors de la résolution d'un défaut de cache, la ligne remplacée est écrite dans un petit cache associatif appelé cache victime. Quand un défaut de cache se produit dans le cache principal, la résolution du défaut commence par consulter le cache victime pour s'assurer qu'il ne contient pas la ligne recherchée. Si la ligne n'est pas présente au niveau du cache victime la recherche se poursuit dans le niveau mémoire supérieur. L'utilisation d'un cache victime a montré une réduction significative des défauts de cache mais peut constituer un facteur de dégradation de performances important. Enfin, notons que les défauts du cache victime sont qualifiés d'externes par opposition aux défauts internes propres au cache principal. [**8**]

## 1.11 Les stratégies de recherche

Une donnée peut être ramenée vers le cache de plusieurs manières. L'algorithme de recherche doit décider quand et comment une donnée doit être ramenée vers le cache. Il existe trois (03) méthodes pour ramener une donnée vers le cache : la recherche à la demande, l'anticipation et la recherche sélective. [**8**]

- **La recherche à la demande** consiste à ramener un bloc vers le cache quand celui-ci est référencé. Cette stratégie est simple à implémenter mais n'améliore en rien l'efficacité d'un cache. Certes, les performances d'un cache dépendent d'autres paramètres telle la taille du cache, mais la recherche à la demande est le résultat d'un défaut de cache et c'est précisément ce paramètre que l'on doit réduire. [**8**]
- **L'anticipation** implique ramener une donnée avant qu'elle ne soit référencée. Ce type de recherche peut s'avérer très efficace. Ainsi, l'anticipation séquentielle qui consiste à ramener le bloc i+1 si le bloc i est référencé peut réduire jusqu'à 50% le nombre de défauts de cache sous certaines conditions de taille du cache. [**8**]
- **La recherche sélective** permet un choix sur les données à ramener vers le cache. L'algorithme de recherche peut décider de ne pas ramener une donnée vers le cache et de la charger directement de la mémoire ou du disque en court-circuitant le cache. Un exemple est donné dans le cas des systèmes à multiprocesseur où les données communes peuvent ne pas être mises en cache pour assurer une meilleure

cohérence de celles-ci. [**8**]

## 1.12 Les stratégies de mise a jour

Le cache est une mémoire temporaire, quand une donnée est modifiée au niveau du cache, cette modification doit être répercutée sur les niveaux supérieurs de la hiérarchie mémoire. Ainsi, une cohérence de données doit être maintenue au niveau de la hiérarchie mémoire. Plusieurs politiques de mise à jour peuvent être envisagées :

- **Écriture simultanée (write-through)** : Lorsque une ligne est modifiée dans le cache, cette modification est immédiatement répercutée dans les niveaux supérieurs de la hiérarchie mémoire. Cette politique garantit à tout moment la cohérence entre les différents niveaux mais augmente le trafic sur le bus de données.
- **Recopie (write back)** : La ligne est marquée modifiée, elle ne sera recopiée dans les niveaux supérieurs de la hiérarchie que si elle est la cible d'un remplacement au niveau du cache.

Si à présent, on décide d'écrire une donnée dans une ligne absente du cache, l'une des deux options suivantes est utilisée :

- **Écriture attribué (write-allocate)** : La ligne est chargée dans le cache puis modifiée.
- **Écriture non attribuée ( no write-allocate)** : la ligne est modifiée directement dans le niveau supérieur de la hiérarchie et non chargée dans le cache.

Généralement, l'option write-allocate est associée à la politique d'écriture write back, et l'option no write allocate à la politique d'écriture write through. [**3**]

# 1.13 Éléments de la conception du cache

Dans la présente section, nous donnons un aperçu sur chaque paramètre de conception du cache. Par l'explication de chacun d'entre eux, nous allons montrer leurs rôles décisifs dans la bonne conception du cache. Ces éléments servent principalement à classer, à différencier les architectures du cache et sont récapitulés dans l& table 1.1 [**16**] :

| Éléments de la conception du cache | Description |
|---|---|
| Taille du cache | -Taille du cache L1 de 16 à 32 Ko. |
| | -Taille du cache L2 de 256 à 512 Ko. |
| Taille des lignes | -Entre 8 et 64 Octets. |
| Associativité | -Cache directe. |
| | -Cache Completement associative. |
| | -Cache associative par ensemble. |
| Algorithme de remplacement | -LRU(Least Recenty Used,moins récement utilisé). |
| | -FIFO(First In First Out). |
| | -Aléatoire. |
| Stratégies d'écriture | -Ecriture simultanée(Write Through ). |
| Nombre de caches | -Ecriture différée (Write Back). |

TABLE 1.1 – Description des éléments de la conception du cache

### 1.13.1 La taille du cache

La taille du cache doit être suffisamment réduite pour minimiser le temps d'accès au cache. Car plus il est grand, plus il faut de parcours pour accéder à une information dans le cache. En conséquence, les caches de grande taille tendent à être légèrement plus lents. De plus, les performances de ce dernier étant particulièrement sensibles à la nature de la charge de travail, il est impossible de parvenir à une taille de cache optimale unique . Le tableau 1.2 liste les tailles de cache pour des processeurs actuels et anciens.

| Processeur | Type | Année | il1/dl1 | L2 | L3 |
|---|---|---|---|---|---|
| IBM360/85 | Gros ordinateur | 1968 | 16 à 32 Ko | _ | _ |
| Pentium | PC | 1993 | 8Ko/8Ko | 256 à 512Ko | _ |
| Pentium 4 | PC/Serveur | 2000 | 8Ko/8Ko | 256 Ko | _ |
| Itanium | PC/Serveur | 2001 | 16Ko/16Ko | 96 Ko | 4 Mo |
| Intel Core i7 | PC/Serveur | 2009 | 64 Ko | 256 Ko | 6 Mo |

TABLE 1.2 – Exemples de taille de cache pour différents processeurs

Nous remarquons d'une façon générale que la taille des caches augmente quasiment par puissance de deux et cela par niveau et par année aussi. Comme pour le cas des processeurs Pentium 4 et Itanium, dans l'espace d'un an la taille du L1 à doubler. Les raisons de cette augmentation peuvent être nombreuses, on peut citer notamment le besoin des applications en en calculs intensifs et en stockages des données. Donc, il est impossible de parvenir à une taille de cache unique pour tous les processeurs. **[16]**

### 1.13.2 Taille de bloc

La relation entre la taille de bloc et le taux de succès est complexe. En effet, cela

dépend des caractéristiques de localité d'un programme donné. Si les mots demandés par le processeur se trouvent dans un voisinage proche alors le taux de succès augmente avec l'augmentation de la taille de la ligne du cache. Et inversement, le taux de succès diminue lorsque la probabilité d'exploiter les informations récemment utilisées devient plus faible. Donc il est difficile de définir une valeur optimale définitive, généralement la taille de bloc du cache est située entre 16 et 128 octets. [**16**]

### 1.13.3 L'associativité

Le rôle de la fonction de correspondance intervient dans la mesure où les blocs de cache sont moins nombreuses que les blocs de mémoire principale. Il faut, en outre, trouver un moyen de déterminer quel bloc de la mémoire principale occupe le bloc de cache. Le choix de la fonction de correspondance détermine l'organisation du cache. Comme expliqué précédemment, on peut employer trois techniques de correspondance : directe, completement associative et associative par ensemble. [**16**]

## 1.14 Conclusion

Pour conclure, la mémoire cache permet d'augmenter la vitesse moyenne de communication entre un processeur et des composants de stockage comme la RAM ou un disque dur. On peut utiliser plusieurs cache en cascade. Celui qui est le plus près de la CPU est le plus rapide et le plus petit . On parle de cache L1, L2 et L3. Quand on dispose de plusieurs niveaux de cache, on peut les faire fonctionner de façon indépendante ( inclusif ) soit de concert ( exclusif ), comme s'il s'agissait d'un seul cache. Il existe trois méthodes d'organisation des informations dans le cache : " Direct mapped " où chaque ligne de cache correspond à un bloc déterminé de la RAM ; le " Fully associative " où chaque ligne de cache peut gérer n'importe quel bloc de la RAM ; et le " N-way associative " , compromis des autres méthodes où l'on regroupe n lignes de caches pour les affecter à un bloc déterminé. Le fonctionnement normal entre une mémoire cache et un composant de stockage ( RAM, disque dur ) implique que toute écriture qui se fait dans le cache se fasse aussi en mémoire, c'est la méthode appelée write - through. Pour l'améliorer, on utilise la méthode write-back (écriture différée ). Cependant elle ne garantit pas la cohérence du cache, c'est à dire que les données contenues dans le cache doivent correspondent à celles situées dans le composant de stockage (RAM, disque dur ). Dans ce cas, on dit que l'intégrité des données est préservée.

Le chapitre suivant va être consacré à étudier le simulateur simplescalar v2.0 pour ARM et présenter les benchmarks de la suite mediabench et mibench.

# Chapitre 2

## Environnement Expérimental

## 2.1 Introduction

Pour des travaux de recherche en architecture comme pour de nombreuses applications, il est nécessaire de pouvoir disposer d'un simulateur logiciel de processeurs existants. Par exemple, l'optimisation de programmes sur les nouvelles architectures complexes de processeurs requiert une compréhension détaillée du comportement du programme sur l'architecture en raison des nombreuses interactions entre composants du processeur.En raison de la complexité matérielle de ces systèmes, il n'est plus possible de construire des prototypes réels (matériels) pour explorer les différents choix de conception, on requiert donc de plus en plus souvent au test logiciel de ces systèmes informatiques ce qui nécessite également de disposer des simulateurs des processeurs inclus dans le système. D'un autre coté Les processeurs actuels sont de plus en plus compliqués car ils utilisent des techniques très évoluées afin d'avoir de meilleures performances. Cependant, cette complexité fait qu'il est plus difficile de les tester et de comprendre leur fonctionnement. De plus, il est impossible d'effectuer des modifications sur le processeur afin de tester des nouveaux concepts. C'est pour cela que des simulateurs (tels que SPIM, SimOS ou SimpleScalar) ont été créés. Ces simulateurs permettent facilement d'implémenter de nouveaux concepts et de les tester. Le simulateur SimpleScalar a été créé également dans cette optique.[**17**]

Dans le cadre de notre travail nous nous utilisons le simulateur simplescalar pour analyser l'exécution des programmes sur divers architecture, permet en effet, de faire des simulation en mode fonctionnelle et détaillé de la micro-architecture en utilisant des simulateur comme sim-fast et sim-safe. Et le simulateur utilisé dans notre travail c'est le sim-cache qui permet de paramétrer le sous système de caches

## 2.2 SimpleScalar

SimpleScalar est un ensemble d'outils de simulation qui est employé pour analyser l'exécution de programme sur divers ensembles d'architecture. Les simulateurs inclus dans le paquet (SimpleScalar) donnent à l'utilisateur une variété d'options et de niveaux pour simuler, commençant d'une simulation simple et rapide, à une simulation de détail où l'utilisateur peut examiner les divers techniques d'implémentation matériel telles que les caches non-bloquants, l'exécution spéculative, et la prédiction de branchement. SimpleScalar inclus également des outils de visualisation d'exécution, recours d'analyse statistique, et infrastructure de débogage et vérification (SimpleScalar intègre également un debugger se nommant DLite!. Il s'interface avec les simulateurs grâce à l'utilisation de quelques fonctions). Il est de plus possible d'utiliser le compilateur GCC , ainsi que les utilitaires qui lui sont associés, afin de générer des programmes pour SimpleScalar. De plus, il y en a une variété de benchmarks (à l'instar de SPEC) qui ont été recompilés afin de pouvoir être utilisés sur le simulateur.

Les principaux avantages sont sa possibilité (il a en effet été testé sur de nombreuses

platte formes), sa flexibilité, son extensibilité et ses performances. Il est notamment permis de le modifier selon sa convenance pour usage personnel. Les tests standards (benchmarks ) de Mediabench et Mibench ont été spécialement compilés pour pouvoir être utilisés sur ce simulateur.

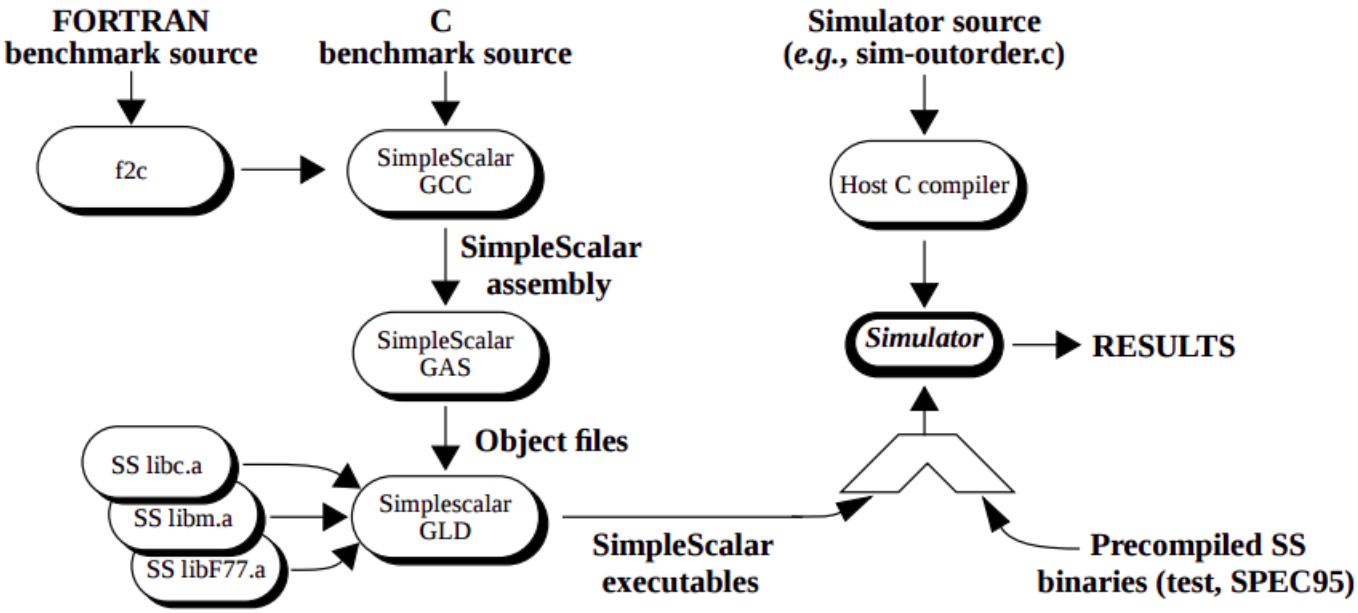



FIGURE 2.1 – Vue de l'ensemble du simplescalar

Mais il est aussi possible de tester ses propres programmes (écrits en C ou en FORTRAN) en les compilant avec une version adaptée de GNU GCC, fournie dans la suite simplescalar (figure 2.1).

En entrée du simulateur, les fichiers sources C ou FORTRAN sont compilés par une version croisée de GCC (après conversion en C par f2c pour les programmes FORTRAN) produisant en sortie un code binaire interprétable par le simulateur.Celui-ci se présente sous forme d'un programme exécutable conduit à partir d'une description de l'architecture et d'une description du jeu d'instructions, tous deux décrits par un ensemble de fichiers C. Le simulateur doit être compilé préalablement sur la machine hôte qui va simuler le processeur.Le simulateur ainsi construit charge le code binaire produit par le compilateur croisé et exécute celui-ci.Il produit une trace de l'exécution et des statistiques relatives à l'exécution. La trace peut éventuellement être mise en forme plus tard pour pouvoir être examinée aisément.

La simulation produit non seulement des statistiques détaillées sur l'utilisation des ressources internes (le taux de défauts de cache "Miss-rate :mots demandés n'existent pas dans le cache", nombre d'accès TLB (Translation Lockaside Buffer ), nombre de sauts mal prédits,....etc.) mais fournit également une trace de l'exécution . **[6]**

## 2.3 Les jeux d'instructions émulées par SimpleScalar

Les simulateurs de SimpleScalar peuvent émuler les jeux d'instructions Alpha, PISA, ARM, et les jeux d'instructions x86. L'outil inclut une infrastructure de définition de machine qui permet à la plupart des détails architecturaux d'être séparés des réalisations de simulateur. Tous les simulateurs distribués avec la version courante de SimpleScalar peuvent exécuter des programmes de n'importe lequel des jeux d'instructions cités ci- dessus. L'architecture modulaire permet à Simplescalar d'émuler même les jeux d'instructions complexes CISC (tel que, x86).

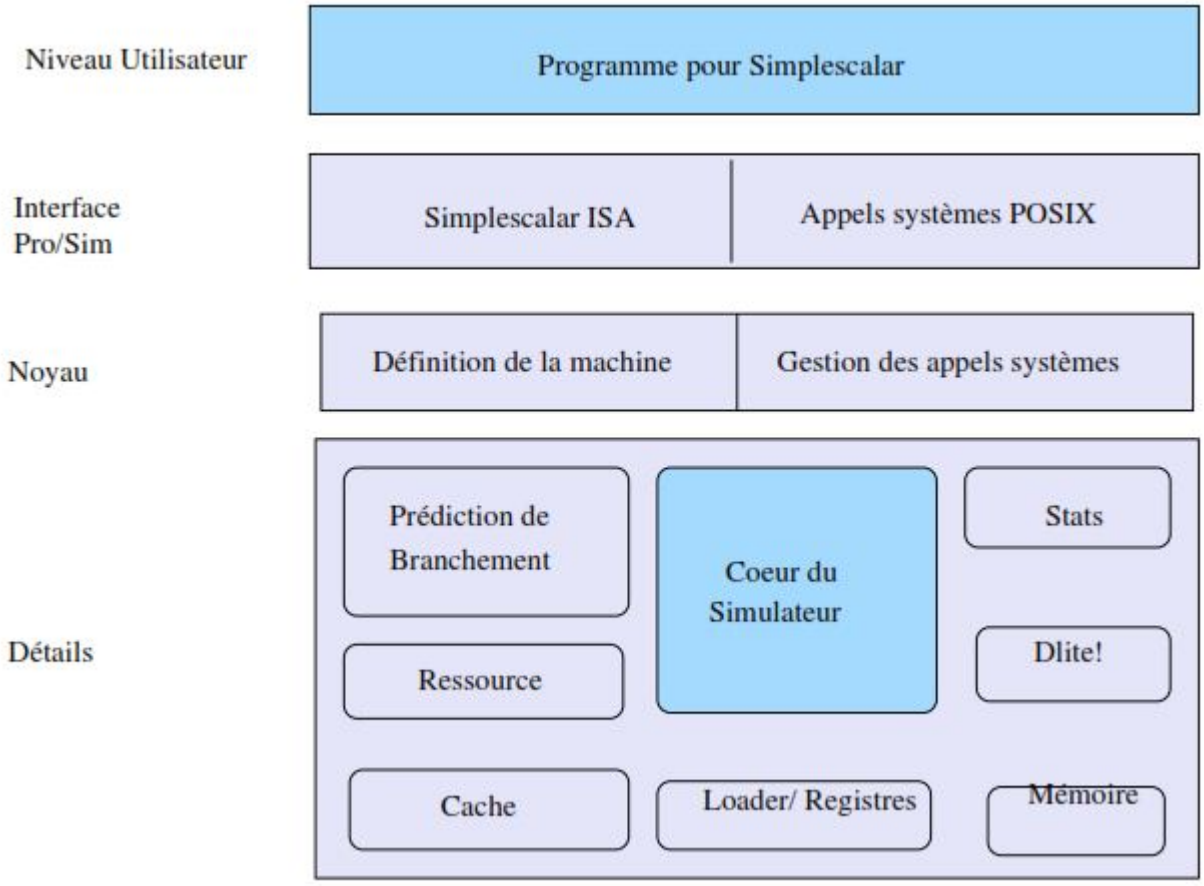



FIGURE 2.2 – Architecture générale de SimpleScalar

La figure 2.2 présente l'architecture des différents simulateurs SimpleScalar. Les appels systèmes POSIX (Portable Operating System Interface :c'est le standard IEEE1003, a défini un ensemble de fonctions disponibles sur tous les systèmes Unix) sont exécutés mais non simulés (l'exécution permet d'obtenir l'effet souhaité pour le programme simulé mais le code des appels systèmes ne fait pas partie de la simulation). Le simulateur décode les instructions PISA (Portable Instruction Set Architecture) à partir de la définition de la machine PISA. La simulation de chaque instruction décodée par le coeur du simulateur met en oeuvre, selon le simulateur choisi, diverses ressources parmi lesquelles les caches, la

prédiction de sauts, le débogage avec Dlite ! (lightweight symbolic debugger), la mémoire ou l'affichage de statistiques de la simulation. [**10**]

## 2.4 Les plate formes qui supportent SimpleScalar

SimpleScalar est supporté par la plupart des plate formes 32 bits et 64 bits basées sur UNIX et Windows NT comme système d'exploitation. L'architecture logicielle interne de l'outil comporte un module d'interfaçage qui facilite la communication avec d'autres plate formes. La plupart des développeurs et des utilisateurs de SimpleScalar (y compris SimpleScalar LLC) emploient SimpleScalar sous Linux/x86. [**6**]

## 2.5 Fonctionnement des différents simulateurs

Le simulateur comporte huit programmes allant du plus simple (donc celui qui s'exécute le plus rapidement) à celui qui comporte plusieurs fonctions telles que l'exécution dans le désordre : Ces différents simulateurs sont : sim-fast, sim-safe, sim-cache, sim-cheetah, sim-bpred, simeio, sim-profile et sim-outorder. Nous allons maintenant décrire le fonctionnement de ces simulateurs, les arguments qu'ils acceptent et leur structure interne. [**6**]

### 2.5.1 Les simulateurs de base : *sim-fast* et *sim-safe*

Ces deux simulateurs implémentent le minimum de fonctions.

Le premier et le plus rapide, sim-fast est implémenté dans sim-fast.c. Il possède très peu de fonctions. Il exécute les instructions séquentiellement et ne simule pas l'exécution en parallèle des instructions. sim-fast est optimisé pour exécuter les instructions rapidement et ne possède pas de cache et ne supporte pas le débugger D Lite !. Une autre version de sim-fast se nomme sim-safe . La différence entre ces deux versions est que sim-safe vérifiée que les instructions sont alignées et pour chaque accès à la mémoire, la permission est vérifiée. Les deux simulateurs utilisent les fichiers suivants : *main.c*, *syscall.c*, *memory.c*, *regs.c*, *loader.c*, *machine.c*, *endian.c*, *misc.c*. De plus, sim-safe utilise *dlite.c*. [**11**]

### 2.5.2 Simulation de cache : *sim-cache* et *sim-cheetah*

SimpleScalar est distribué avec deux versions gérant les caches : sim-cache et sim-cheetah. Ces deux simulateurs utilisent le fichier cache.c et ils utilisent respectivement les fichiers sim-cache.c et sim-cheetah.c.

Il est possible de configurer :

- la taille des deux niveaux de mémoire cache d'instructions et de données
- La taille du TLB (Translation Lockaside Buffer – Tampon de traduction anticipée) d'instructions et de données

- Le comportement lors d'un appel système .

Il est de plus possible de générer un fichier de statistiques à l'aide de l'option –pcstat.

Le format de la configuration <config> est le suivant :
<name> :<nsets> :<bsize> :<assoc> :<repl>

- <name> : nom du cache
- <nsets> : nombre d'ensembles
- <bsize> : taille de bloc
- <assoc> : associativité
- <repl> : politique de remplacement

La taille de bloc devant être remplacée par la taille d'une page pour le TLB.

L'associativité doit être une puissance de 2, et la politique de remplacement soit :

- l : LRU (Last Recently Used),
- f : FIFO (First In First Out),
- r : Random (Aléatoire). [**11**]

Notre travail consiste à explorer l'ensemble des paramètres du cache qui sont nsets, bsize et assoc pour déterminer la configuration du cache qui répond au mieux aux besoins d'une application en terme de performances.

### 2.5.3 Génération de statistiques : *sim-profile*

Le simulateur sim-profile permet de produire des informations concernant l'exécution d'un programme. Il peut générer des statistiques détaillées sur les adresses des instructions, les accès mémoires, les branchements, etc.[**11**]

### 2.5.4 L'exécution des instructions dans le désordre : *sim-outorder*

C'est le simulateur le plus complet et le plus compliqué (il fait plus de 4000 lignes). Ce simulateur est capable de lancer et d'exécuter des instructions dans le désordre. Pour cela, il utilise le RUU (Register Update Unit ). Le RUU utilise un buffer de réordonnancement pour renommer les registres automatiquement. [**11**]

#### 2.5.4.1 Le pipeline du Simplescalar

L'exécution en pipeline de `sim-outorder` est simulée la boucle principale du simulateur, qui se trouve dans la fonction `sim_main ()`, est définie ainsi :

```
ruu_init();
for (;;) {
ruu_commit();
ruu_release_fu();
ruu_writeback();
lsq_refresh();
ruu_issue();
ruu_dispatch();
ruu_fetch();
}
```

Cette boucle est exécutée une fois par cycle d'horloge. On pourra remarquer que le pipeline est utilisé à l'envers. En effet, cette technique permet de synchroniser le passage de données d'une étape à une autre.

L ' étape ***ruu_fetch()*** est la partie du pipeline qui se charge de charger les instructions. Cette unité extrait les données suivantes : le compteur de programme PC et la détection d'une mauvaise prédiction de branchement provenant de l'unité d'exécution. À chaque cycle, les instructions sont lues à partir d'une seule ligne du cache d'instructions. Après que les instructions ont été lues, elles sont placées dans une liste qui sera utilisée par ***ruu_dispatch***.

La partie du simulateur qui se charge de répartir les instructions se situe dans la fonction ***ruu_dispatch()***. C'est ici que le décodage et le renommage des registres est effectué. Les instructions proviennent de la liste créée par l'étape "extraction des instructions" (c'est-à-dire l'étape précédente). Une fois par cycle, cette unité prend le maximum possible d'instructions afin de les placer dans une liste d'ordonnancement.

Les étapes qui lancent les instructions se trouvent dans les fonctions ***ruu_issue()*** et ***lsq_refresh()***. Ces fonctions s'occupent de lancer les instructions dans les unités fonctionnelles.

La fonction lsq_refresh() se charge en particulier de repérer des éventuelles dépendances dues aux registres ou à la mémoire. À chaque cycle, elle localise les instructions qui utilisent des registres qui sont prêts à être utilisés.

L'étape se chargeant de l'exécution est également géré par ***ruu_issue***. À chaque cycle, la fonction obtient le plus d'instructions prêtes à être exécutées. Ces instructions proviennent de la liste d'ordonnancement. La disponibilité des unités fonctionnelles est vérifiée : si certaines d'entres elles sont disponibles, des instructions sont alors lancées. Les latences des unités fonctionnelles sont stockées dans le tableau fu_config[].

L'écriture des résultats est implémentée dans la fonction ***ruu_writeback*** (). À chaque cycle d'horloge, cette fonction parcourt la liste des instructions qui sont achevées. Quand une instruction qui se termine est trouvée, la fonction cherche une autre instruction qui dépend de celle qui vient de se terminer. Si une telle instruction est trouvée alors elle est

prête à être lancée et ainsi exécutée.

La fonction *ruu_commit* reçoit les instructions provenant de l'étape *ruu_writeback()*. Elle remet dans l'ordre les instructions et met à jour le cache de données. [**10**]

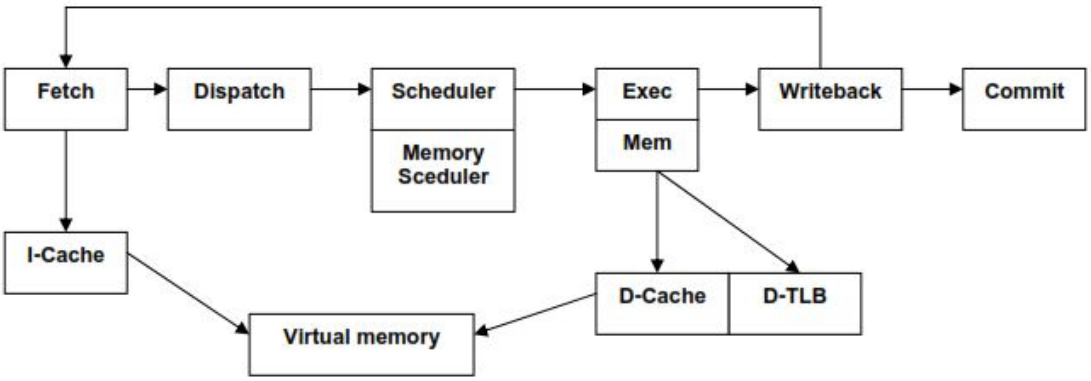



FIGURE 2.3 – Exécution en pipeline de sim-outorder

## 2.6 Gestion de la mémoire

La gestion de la mémoire dans SimpleScalar est implémentée dans les fichiers memory. h et memory. c.

Cette mémoire est partagée en plusieurs sous-blocs (figure 2.4)

1. La pile (Stack Segment) : cette partie contient les différentes variables d'environnement telles que argv, envp et argc. La taille de la pile est limitée à 16384 octets (définie dans le fichier machine. h dans la constante MD_MAX_ENVIRON),
2. Les données (Data Segment) : cette partie est elle-même découpée en deux sous-parties : les données qui sont initialisées (données statiques ou init) et celles qui ne le sont pas (données dynamiques ou bss). La partie dynamique contient les données qui sont créées au cours de l'exécution du programme, par exemple en utilisant la fonction malloc du langage C,
3. Le code (Text Segment) : cette partie contient les instructions du programme. [**11**]

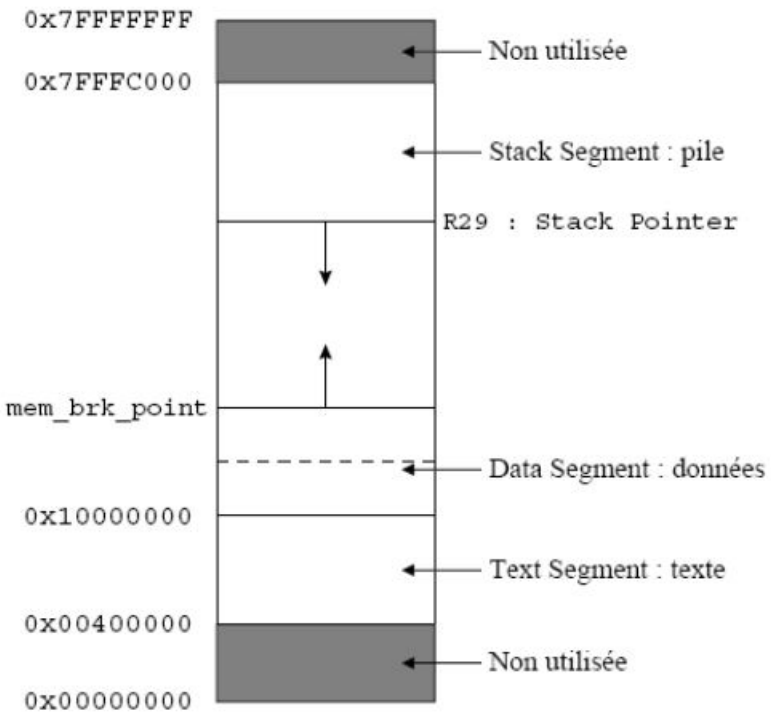



FIGURE 2.4 – Détail de la mémoire de SimpleScalar

## 2.7 Le cache dans SimpleScalar

Le cache est implémenté dans SimpleScalar dans les fichiers cache.h et cache.c. Pour effectuer la création du cache, il est nécessaire de fournir le nombre d'ensemble, la taille d'une ligne et l'associativité comme paramètres. Il faut également indiquer la méthode à utiliser pour accéder à ce cache. La fonction ainsi définie indique la latence d'accès aux lignes lorsque le cache crée un "miss ". Il existe plusieurs politiques de remplacement dans le cache. Ces différentes méthodes sont définies ainsi :

enum cache_policy {
LRU, /* Remplacele dernier bloc utilisé*/
Random, /* Remplace un bloc de manière aléatoire*/
FIFO /* Remplace le plus vieux bloc qui se trouve dans l'ensemble*/} ; **[11]**

## 2.8 Exemple de simulation sous simplescalar

le programe suivants :

```
#include <stdio.h>
#define N 100000
int tab [N],i;
```

```
int main()
{
for(i=0;i<N;i++)
tab[i]=i/37;
return 0;
}
```

l'execution de ce programme vas se passé par deux étapes :

1.compiler le programe **.c** avec le compilateur croisé en utilisant cette commande :

*arm-linux-gcc -static prog.c -o prog*

2.simulation de l'exécutable prog avec le sim-cache en utilisant cette commande :

*sim-cache prog*

voilà les résultat de la simulation :

```
lounas@kali:~/amin$ sim-cache prog
sim-cache: SimpleScalar/ARM Tool Set version 3.0 of November, 2000.
Copyright (c) 1994-2000 by Todd M. Austin.  All Rights Reserved.
This version of SimpleScalar is licensed for academic non-commercial use only.

sim: command line: sim-cache prog

sim: simulation started @ Sat Sep 26 11:15:41 2015, options follow:

sim-cache: This simulator implements a functional cache simulator.  Cache
statistics are generated for a user-selected cache and TLB configuration,
which may include up to two levels of instruction and data cache (with any
levels unified), and one level of instruction and data TLBs.  No timing
information is generated.

# -config                     # load configuration from a file
# -dumpconfig                 # dump configuration to a file
# -h                    false # print help message
# -v                    false # verbose operation
# -d                    false # enable debug message
# -i                    false # start in Dlite debugger
-seed                       1 # random number generator seed (0 for timer seed)
# -q                    false # initialize and terminate immediately
# -chkpt                <null> # restore EIO trace execution from <fname>
# -redir:sim            <null> # redirect simulator output to file (non-interacti
ve only)
# -redir:prog           <null> # redirect simulated program output to file
-nice                       0 # simulator scheduling priority
-max:inst                   0 # maximum number of inst's to execute
-cache:dl1       dl1:256:32:1:l # l1 data cache config, i.e., {<config>|none}
-cache:dl2       ul2:1024:64:4:l # l2 data cache config, i.e., {<config>|none}
-cache:il1       il1:256:32:1:l # l1 inst cache config, i.e., {<config>|dl1|dl2|
none}
-cache:il2               dl2 # l2 instruction cache config, i.e., {<config>|dl2
|none}
-tlb:itlb        itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none
}
-tlb:dtlb        dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-flush                 false # flush caches on system calls
-cache:icompress       false # convert 64-bit inst addresses to 32-bit inst equ
ivalents
# -pcstat               <null> # profile stat(s) against text addr's (mult uses o
k)
```

```
  The cache config parameter <config> has the following format:

    <name>:<nsets>:<bsize>:<assoc>:<repl>

    <name>   - name of the cache being defined
    <nsets>  - number of sets in the cache
    <bsize>  - block size of the cache
    <assoc>  - associativity of the cache
    <repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

    Examples:   -cache:dl1 dl1:4096:32:1:l
                -dtlb dtlb:128:4096:32:r

  Cache levels can be unified by pointing a level of the instruction cache
  hierarchy at the data cache hiearchy using the "dl1" and "dl2" cache
  configuration arguments.  Most sensible combinations are supported, e.g.,

    A unified l2 cache (il2 is pointed at dl2):
      -cache:il1 il1:128:64:1:l -cache:il2 dl2
      -cache:dl1 dl1:256:32:1:l -cache:dl2 ul2:1024:64:2:l

    Or, a fully unified cache hierarchy (il1 pointed at dl1):
      -cache:il1 dl1
      -cache:dl1 ul1:256:32:1:l -cache:dl2 ul2:1024:64:2:l


sim: ** starting functional simulation w/ caches **
warning: unsupported ioctl call: ioctl(-4294945791, ...)
Taille du vecetur: 2 Mo
warning: invalid/unimplemented syscall 91, PC=0x02005cf0, winging it


sim: ** simulation statistics **
sim_num_insn                9969641 # total number of instructions executed
sim_num_uops               15743176 # total number of UOPs executed
sim_avg_flowlen               1.5791 # uops per instruction
sim_num_refs                5771438 # total number of loads and stores executed
sim_elapsed_time                  7 # total simulation time in seconds
sim_inst_rate          1424234.4286 # simulation speed (in insts/sec)
il1.accesses                9969641 # total number of accesses
il1.hits                    9969216 # total number of hits
il1.misses                      425 # total number of misses
il1.replacements                220 # total number of replacements
il1.writebacks                    0 # total number of writebacks
il1.invalidations                 0 # total number of invalidations
il1.miss_rate                0.0000 # miss rate (i.e., misses/ref)
il1.repl_rate                0.0000 # replacement rate (i.e., repls/ref)
il1.wb_rate                  0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
dl1.accesses                5771514 # total number of accesses
dl1.hits                    5694237 # total number of hits
dl1.misses                    77277 # total number of misses
dl1.replacements              77021 # total number of replacements
dl1.writebacks                 4144 # total number of writebacks
dl1.invalidations                 0 # total number of invalidations
dl1.miss_rate                0.0134 # miss rate (i.e., misses/ref)
dl1.repl_rate                0.0133 # replacement rate (i.e., repls/ref)
dl1.wb_rate                  0.0007 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ul2.accesses                  81846 # total number of accesses
ul2.hits                      48747 # total number of hits
ul2.misses                    33099 # total number of misses
ul2.replacements              29003 # total number of replacements
ul2.writebacks                   22 # total number of writebacks
ul2.invalidations                 0 # total number of invalidations
ul2.miss_rate                0.4044 # miss rate (i.e., misses/ref)
ul2.repl_rate                0.3544 # replacement rate (i.e., repls/ref)
ul2.wb_rate                  0.0003 # writeback rate (i.e., wrbks/ref)
ul2.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
```

FIGURE 2.5 – Exemple de simulation avec sim-cache

## 2.9 présentation des benchmarks

L'utilisation des benchmarks est une technique connue pour évaluer les performances d'un système. Nous avons choisi les applications de la famille Mediabench et MiBench pour valider notre démarche de configuration de sous système caches.

Un benchmark désigne typiquement un programme qui exécute un ensemble d'opérations (ou d'autres programmes) dans un système informatique et qui détermine un certain résultat.

### 2.9.1 MediaBench

MediaBench est une suite de benchmarks composée des programmes multimédia. MediaBench se compose d'applications complètes codées dans des langages de haut niveau. MediaBench contient plusieurs applications comme des applications du traitement d'image, des applications de communications . [**12**]

La table 1.2 donne une courte description des application de la suite MediaBench :

| Programme | Description |
|---|---|
| JPEG | Compression d'imaes (Encode/ Decode) |
| MPEG | Compression vidéo (Encode/ Decode) |
| GSM | Norme européenne pour le transcodage de la parole (Encode/ Decode) |
| G721 | Compression CCITT (Encode/ Decode) |
| PGP | Chiffrage à clef publique (RSA) |
| PEGWIT | Chiffrage à clefs publique (courbe elliptique) |
| GHOSTSCRIPT | Interpréteur Post-Script |
| MESA | Clone libre d'OpenGL |
| EPIC | Compression d'image |
| ADPCM | Codage audio |
| RESTA | Un programme de reconnaissance de la parole |

TABLE 2.1 – Programmes de MediaBench

### 2.9.2 Mibench

La suite Mibench sont des benchmarks spécialement développés pour l'évaluation des systèmes embarqués. Ces benchmarks sont divisés en six suites avec chaque suite visant un secteur spécifique comme la bureautique, les réseaux, la sécurité ou les télécommunications. Tous les programmes sont disponibles en code source C. [**13**]

La table 2.2 donne un aperçu sur les différents benchmarks avec leurs descriptions.

| Programme | Description |
|---|---|
| blowfish | Algorithmme de chiffrement (cryptographie). |
| rijndael | Algorithmme de chiffrement avancé. |
| sha | Algorithmme de hachage cryptographique. |
| ispell | Vérificateur d'orthographe |
| patricia | Structure de données utilisée dans les arbres pleins de n¡uds feuilles très clairsemée . |
| dijkstra | Algorithmme de recherche du plus court chemin |

TABLE 2.2 – Programmes de MiBench

## 2.10 Conclusion

Dans ce chapitre, nous avons présenté le simulateur simplescalar serve à la simulation des performances des architectures multi-cores/multi-caches et les programmes ordinaire (benchmarks) extraits de la famille Mediabench et Mibench.

Le chapitre suivants vas être consacré à étudier les performances de cache mémoires L1 et L2 afin de déterminer une configuration satisfaisante des paramètres pour ces deux niveaux de cache .

# Chapitre 3

# Conception

## 3.1 Introduction

Après la description et la présentation du cache mémoire et les outils nécessaires pour notre travail dans les deux chapitres précédents, nous allons débuter la conception de notre projet qui consiste à appliquer plusieurs configurations de cache mémoire (L1&L2) sur l'execution d'un programme (benchmark), puis évaluer et comparer les résultats de ces configurations afin de déterminer le meilleur ajustement du cache. Pour définir l'ensemble des valeurs possibles des paramétres du cache, nous proposons une méthodologie de configuration des caches basée sur la simulation.

Donc l'objectif principal de notre travail est de trouver la meilleure configuration du cache (nsets, bsize, assoc) pour les deux niveaux (L1 et L2). l'utilisation du simulateur simplescalar a limité le travail à ces deux niveaux de cache seulement, puisque il ne dispose pas du niveaux 3 .

## 3.2 sim-cache

Le simulateur sim-cache utilise les arguments se trouvant dans la table 3.1.

La configuration du cache (<string>) est définie comme suit :

<name> :<nsets> :<bsize> :<assoc> :<repl>.

Avec :

<name> : nom du cache crée.

<nsets> : nombre d'ensembles dans le cache.

<bsize> : taille d'un bloc dans le cache .

<assoc> : associativité du cache (doit être une puissance de 2).

<repl> : politique de remplacement (l = LRU, f = FIFO, r = random).

| Argument | Description | VAL PAR DÉFAUT |
|---|---|---|
| cache :dl1 <string> | Cache de données de premier niveau | dl1 :256 :32 :1 :l |
| cache :dl2 <string> | Cache de données de deuxième niveau | ul2 :1024 :64 :4 :l |
| cache :il1 <string> | Cache d'instructions de premier niveau | il1 :256 :32 :1 :l |
| cache :il2 <string> | Cache d'instructions de deuxième niveau | dl2 :256 :32 :1 :l |
| tlb :itlb <string> | Conf. du TLB d'instructions | itlb :16 :4096 :4 :l |
| tlb :dtlb <string> | Conf. du TLB de données | itlb :16 :4096 :4 :l |
| flush : <true\|false> | Vide le cache lors d'un appel système | false |
| icompress : <true\|false> | Converti une inst. 64 bits en 32 bits | false |
| pcstat : <string liste...> | Génère un fichier profile | <NULL> |

TABLE 3.1 – Les arguments du simulateur sim-cache.

Pour avoir la taille du cache, il faut donc multiplier : <nsets>, <bsize> et <assoc>.

Il est possible d'avoir un niveau unifié en faisant pointer le cache d'instruction vers le nom du cache de données correspondant au même niveau. **[10]**

Dans ce travail, nous considérons qu'une configuration est composée de quatre arguments : la taille du cache, la taille de bloc du cache, nombre d'ensemble et l'associativité. L'ensemble des quatre arguments forme ce que nous appelons la configuration d'un cache. De là, nous avons structuré notre configuration en prenant en compte les paramètres caractérisant chaque niveau de cache.

Un bon choix de dimensionnement de cache permet à un calculateur d'atteindre de hautes performances, d'ou l'intérêt de simuler cette unité pour le bon fonctionnement du processeur et d'extraire leurs valeurs optimales.

## 3.3 Les paramètres du cache étudiés

Plusieurs paramètres du cache influencent les performances de l'architecture des systèmes.**[16]**

Un des paramètres les plus important est la taille du cache. En effet, un cache très large implique un nombre réduit d'échecs sur les accès au caches. Cependant les contraintes liés aux ressources (matériaux utilisé pour la construction) et a l'espace réservé au cache dans les système embarqué imposent une réduction de la taille du cache.

Un autre paramètre important est l'associativité du cache. L'associativité détermine combien d'accès simultanés (comparaison de clés) peuvent être effectués dans une mémoire cache pour acquérir une donnée. Dans un cache direct, le nombre de comparaisons est réduit à un, puisque chaque bloc mémoire a un bloc de cache qui lui est dédiée. Dans le cas, d'un cache associatif à 2, 4 ou 8 voies, les possibilités de rangement sont nombreuses pour la même donnée. Ainsi, plus on augmente l'associativité d'un cache et plus nous consommons de l'énergie qui est due principalement au nombre d'accès simultané. Cependant les performances sont meilleures du fait que le nombre de succès sur le cache soit élevé.

Un autre paramètre aussi important à étudier que les deux premiers cités, est la taille de bloc du cache. L'avantage de définir une taille petite pour le bloc, par exemple 16 octets permet de réduire la surcharge sur le bus système par contre augmente en revanche le taux d'échecs sur le cache compte tenu de la petite taille du cache.

le nombre d'ensemble, un autre paramètre qui est tout aussi important dans cette étude.

| Module | Nom du paramètre | Exemples valeurs possibles |
|---|---|---|
| Cache de donnée | Nombre d'ensemble (nsets) | 64,128, 256, 512 |
| | Taille de bloc (bsize) | 8, 16, 32, 64 (octets) |
| | Associativité (assoc) | 1, 2, 4, 8 |

TABLE 3.2 – Exemple de valeurs possibles pour le cache de donnée dl1

## 3.4 Les algorithmes

Dans cette section, nous allons présenter l'ensemble des algorithmes que nous avons conçu pour l'accomplissement de notre travail.

Chacun de ces algorithmes est conçu pour être exécuté sur les différents benchmarks, en faisant varier un paramètre particulier de la configuration du cache mémoire.

Nous rappelons que les paramètres pris en considération dans notre travail sont :

- le nombre d'ensemble,
- la taille de bloc, et
- l'associativité.

De ce fait, nous présentons dans la suite, trois algorithmes, dont chacun ne faisant varier que l'un des paramètres et donnant des valeurs fixes à tous les autres. Les trois algorithmes ont une forme similaire. Il s'agit dans chacun d'eux d'exécuter plusieurs itérations d'une boucle faisant varier le paramètre correspondant. Dans chaque itération, après avoir lancer la simulation avec sim-cache, on extrait les valeurs déterminantes pour notre étude (taux de succès, défauts de cache, nombre de remplacements) pour les sauvegarder dans un fichier. Après avoir exécuter toutes les itérations, nous visualisons les résultats sous forme de courbe au moyen de l'outil gnuplot.

**Algorithme 3.1** Simulation en faisant varier le nombre d'ensembles.

```
/* Définition des paramètres de configuration de test */
k=1024 /* définition d'un kilo octets */
bsize=<la taille du bloc > /* définition de nombre d'ensemble */
assoc=<associativité> /* définition de l'associativité */
repl=<remplacement> /* définition de remplacement*/
Début
```

1. pour nsets dans {32, 64, 128, 256, 512,1024,2048,4096} faire
   - conf={nsets}{bsize}{assoc}{repl})
   - taille =((bsize*nsets*assoc)/k) */* calculer la taille du cache*/*
   - écrire (configuration du cache conf, la taille du cache taille)
   - lancer la simulation avec sim-cache et envoyer les résultats dans un fichier output
   - extraire les valeurs de la simulation
   - afficher et sauvegarder les résultats de simulation dans un fichier.dot

   fait
2. visualiser les résultats enregistrés dans le fichier .dot avec l'outil gnuplot

fin

L'algorithme 3.1 permet d'étudier le première paramètre de configuration de cache mémoire, à savoir le nombre d'ensembles, dans cet algorithme nous faisons varier ce paramètre pour prendre successivement les valeurs 32, 64, 128, 256, 512,1024,2048,4096, et fixer tous les autres.

**Algorithme 3.2** Simulation en faisant varier la taille de bloc.

```
/* Définition des paramètres de configuration de test */
k=1024 /* définition d'un kilo octets */
nsets=<le nombre d'ensembles> /* définition de nombre d'ensemble */
assoc=<associativité> /* définition de l'associativité */
repl=<remplacement> /* définition de remplacement*/
Début
```

1. pour bsize dans {32, 64, 128, 256, 512} faire
   - conf={nsets}{bsize}{assoc}{repl})
   - taille =((bsize*nsets*assoc)/k) */* calculer la taille du cache*/*
   - écrire (configuration du cache conf, taille)
   - lancer la simulation avec sim-cache et envoyer les résultats dans un fichier output
   - extraire les valeurs de la simulation
   - afficher et sauvegarder les résultats de simulation dans un fichier.dot

   fait
2. visualiser les résultats enregistrés dans le fichier .dot avec l'outil gnuplot

fin

L'algorithme 3.2 permet d'étudier le second paramètre de configuration de cache mémoire, à savoir la taille de bloc, dans cet algorithme nous faisons varier ce paramètre pour prendre successivement les valeurs `32, 64, 128, 256, 512`, et fixer tous les autres.

**Algorithme 3.3** Simulation en faisant varier l'associativité

```
/* Définition des paramètres de configuration de test */
k=1024 /* définition d'un kilo octets */
bsize=<la taille du bloc > /* définition de la taille de bloc */
nsets=<le nombre d'ensembles> /* définition de nombre d'ensemble */
repl=<remplacement> /* définition de remplacement*/
Début
```

1. pour assoc dans {1, 2, 4, 8,16} faire
   - conf={nsets}{bsize}{assoc}{repl})
   - taille =((bsize*nsets*assoc)/k) /* calculer la taille du cache */
   - écrire (configuration du cache conf, la taille du cache taille)
   - lancer la simulation avec sim-cache et envoyer les résultats dans un fichier output
   - extraire les valeurs de la simulation
   - afficher et sauvegarder les résultats de simulation dans un fichier.dot

   fait
2. visualiser les résultats enregistrés dans le fichier .dot avec l'outil gnuplot

fin

L'algorithme 3.3 permet d'étudier le troisième paramètre de configuration de cache mémoire, à savoir l'associativité, dans cet algorithme nous faisons varier ce paramètre pour prendre successivement les valeurs `1, 2, 4, 8, 16`, et fixer tous les autres.

Tout les trois algorithmes seront implémentés dans la partie de réalisation avec les scripts shell bash. Afin d'obtenir une configuration satisfaisante pour le cache mémoire nous allons analyser et interpréter les représentations graphiques obtenues avec gnuplot.

## 3.5 Modes de simulation

Il existe plusieurs modes de simulation : le mode fonctionnel, le mode détaillé et le mode de simulation par échantillonnage (figure 3.1).

La simulation fonctionnelle est la plus rapide en termes d'exécution, cependant ne donne aucune information sur les mesures de performances, elle sert uniquement à vérifier la bonne exécution du programme sur l'architecture définit.

La simulation par cycle détaillée permet d'avoir une trace complète de l'exécution du système et donc elle donne plus de précision que toute autre technique de simulation.

La solution compromise entre les deux est la simulation par échantillon.Le processus d'échantillonnage accélère l'exploration de l'espace de conception en réduisant le temps de

simulation d'une exécution d'un programme donné. Au lieu de mesurer la performance de l'ensemble du programme en mode simulation détaillée. Le processus d'échantillonnage, comme son nom l'indique, mesure les performances uniquement des extraits choisis. Et donc la performance de l'ensemble du programme est reconstituée à partir des performances mesurées des "échantillons".[**16**]

les modes du simulateur simplescalar sont le mode fonctionnel et le mode détaillé.

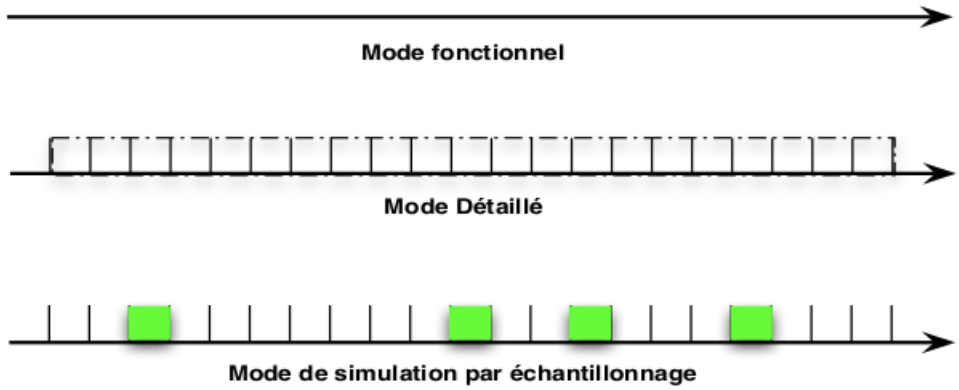



FIGURE 3.1 – Différents modes de simulation.

## 3.6 Conclusion

Après avoir déterminé notre objectif, identifié nos besoins, et conçu tout un projet qui nous permettra d'atteindre notre but. Nous sommes maintenant près à la mise en ¡uvre de notre projet, la prochaine étape sera la plus importante, nous procéderont l'implémentation des algorithmes présenter ci-dessus et nous ferons les différents analyse des résultats qui seront produit par les différents simulation.

# Chapitre 4

## Réalisation

## 4.1 Introduction

Dans ce chapitre, nous présentons les outils de développement qu'on a utilisé ainsi que les résultats expérimentaux de notre solution pour résoudre le problème de configuration des caches mémoires dans une architecture multi-cores/multi-niveaux. Nous avons validé notre démarche de configuration des caches en utilisant 15 benchmarks de la famille Mediabench et MiBench.

## 4.2 Procédé à suivre

- Installer le simulateur simplescalar 2.0 sous Linux
- Télécharger des benchmark pour évaluer les performances des différentes configuration parmi ceux qui existent nous avons choisi un ensemble de programme de test de la famille MiBench et Mediabench
- Installer le compilateur croisé arm-linux-gcc pour, éventuellement, compiler les benchmarks fournis sous forme source
- Le Script shell bash pour les différents configurations de mémoire cache (L1&L2) qui sont sauvgarder dans des fichies se termine par (.sh)
- Exécuter chaque benchmark (compilé) avec chacune des configurations et relever les valeurs des qualités les plus significatifs pour l'étude
- un simulateur de cache "sim-cache" permettant de simuler des programmes exécutables.
- Pour visualiser les résultat des configuration du cache avec les paramètres appliqués, on utilise l'outil gnuplot 4.6
- Analyser et interpréter les représentations graphique obtenu avec gnuplot, et en déduire la meilleure configuration
- Dans notre cas la performance du cache est définie par le taux de succès (qui est le rapport du nombre de succès de cache sur le nombre d'accès mémoire), le nombre de remplacement et le nombre de défauts de cache.

## 4.3 Outils de développement

Pour réaliser cette étude nous avons utilisé les outils suivants :

### 4.3.1 Le simulateur simplescalar sous linux

simplescalar est un simulateur d'architecture très flexible, portable, extensible et performante, se compose de plusieurs simulateurs parmi eux les deux simulateurs sim-cache

et sim-cheetah, ces deux simulateurs permettent d'obtenir des statistiques sur l'utilisation des hiérarchies mémoires à un et à deux niveaux. Sim-cache est un simulateur idéal pour la simulation rapide des caches (si l'effet des performances du cache sur le temps d'exécution est pas nécessaire), idéal pour réaliser des études de cache de haut niveau qui ne prennent pas le temps d'accès aux caches en compte. [**10**]

### 4.3.2 Un compilateur croisé pour ARM

La cross compilation est la possibilité sur une machine avec un matériel spécifique (architecture) avec un système d'exploitation donné, de compiler des programmes pour une autre architecture, ou pour un autre système d'exploitation.

Cela peut être utilisé par exemple pour compiler un programme sur votre ordinateur de tous les jours (sous Gnu/Linux, avec une architecture i386) à destination de votre téléphone mobile, qui lui est sous Symbian avec un processeur ARM.[**15**]

### 4.3.3 Un ensemble de benchmarks

Tous les résultats présentés dans cette étude sont issus des benchmarks MediaBench et MiBench .Chaque programme a été compilé avec un compilateur GCC gnu pour Simplescalar. [**13**]

### 4.3.4 Le shell bash

Le shell n'est pas qu'un simple interpréteur de commandes, mais dispose d'un véritable langage de programmation avec notamment une gestion des variables, des tests et des boucles, des opérations sur les variables, des fonctions...

Toutes les instructions et commandes sont regroupées au sein d'un script. Lors de son exécution, chaque ligne sera lue une à une et exécutée. Une ligne peut se composer de commandes internes ou externes, de commentaires ou être vide. Plusieurs instructions par lignes sont possibles, séparées par le `";"` ou liées conditionnellement par `"&&"` ou `"||"`. Le point virgule `";"` est l'équivalent d'un saut de ligne. Par convention les noms des shell scripts se terminent généralement (pas obligatoirement) par ".sh" pour le `Bourne Shell` et le `Bourne Again Shell`, par ".ksh " pour le `Korn Shell` et par ".csh " pour le `C Shell`.

Dans notre cas le script shell sert à implémenter les différents algorithmes présenter dans le chapitre précèdent qui nous permettons de définir les différents configurations de cache pour simuler les tests des benchmarks (Mediabench, Mibench) et tracer les courbes de différents simulation sous l'outil gnuplot.[**15**]

### 4.3.5 L'outil gnuplot

Gnuplot est un logiciel qui sert à produire des représentations graphiques en deux ou trois dimensions de fonctions numériques ou de données. Le programme fonctionne sur de nombreux ordinateurs et systèmes d'exploitation (Linux, Windows,...) et peut envoyer les

graphiques à l'écran ou dans des fichiers dans de nombreux formats. Dans notre travail on utilise des script shell pour définir les résultats de simulation des benchmarks étudie sous forme des courbes. [**14**]

### 4.3.6 Édition de document LyX

LyX est un logiciel libre d'édition WYSIWYM (What You See Is What You Mean) de document basé sur LaTeX. Doté d'une interface graphique facile à prendre en main, il permet d'avoir en direct une visualisation grossière de ce qu'on veut dire, et pour voir le résultat il suffit de demander à visualiser le document. Une aide très bien fournie et en français favorise la prise en main du logiciel. Le rendu des documents n'a rien à envier à ce que peut produire LaTeX, et le logiciel est configurable aux petits oignons. La compilation en LaTeX et l'exportation en formats imprimables sont grandement facilités. [**2**]

## 4.4 Les configuration de caches proposé

Un processeur SuperScalar comporte deux types de caches : cache instructions (I-cache) et cache données (D-cache).Nous utilisons le paramétrage qui est modifié selon plusieurs paramètres de cache (table 4.1) en vue d'améliorer la performance de système cache (nombre de défauts, taux de succès, nombre de remplacements) .

| Module | Nom de paramètres | Exemples de valeurs possible |
|---|---|---|
| dl1/il1 | nombre d'ensemble(**nsets**) | **32,64,128,256,512,1024,2048,4096** |
| | taille de bloc(**bsize**) | **32,64,128,256,512** |
| | associativité(**assoc**) | **1,2,4,8,12,16** |
| | remplacement(**repl**) | **l,f,r** |
| ul2 | nombre d'ensemble(**nsets**) | **1024,2048,4096,8192,16384,32768,65536** |
| | taille de bloc(**bsize**) | **32,64,128,256,512** |
| | associativité(**assoc**) | **1,2,4,8,12,16** |
| | remplacement(**repl**) | **l,f,r** |

TABLE 4.1 – paramétrage de cache

## 4.5 Les résultats de la simulation :

Nous avons utilisé dans ce travail des benchmarks mediabench et mibench avec les différents paramétres de cache. Dans cette expérimentation, nous analysons l'impact des paramètres nombre d'ensemble (nsets), la taille de bloc (bsize) et l'associativité (assoc) du cache de premier niveau L1 et deuxième niveau L2 sur le nombre de défauts, nombre de remplacement et le taux de succès. Nous procédons en faisant varier le nombre d'ensemble, la taille de bloc et l'associativité du cache L1 et L 2, La politique de remplacement utilisé durant cette étude est LRU (Least Recently Used).

Les programme de test utilisé pour valider les résultats est le benchmark jpeg pour la suite mediabench et dijkstra_small pour la suite mibench.

### 4.5.1 Mediabench

Les programmes de tests utilisés pour évaluer les performances sont extraits de la suite des programmes Mediabench.

#### 4.5.1.1 Nombre de misses (nombre de défauts)

Chaque simulation est faite a partir des scripts shell où on as définit les configurations des différents paramétres de caches (nsets, bsize, assoc).

Exemple d'un script shell qui est défini dans la figure 4.1 suivante :

```
#!/bin/bash

K=1024
nsets_dl1=256
nsets_il1=256
bsize_dl1=32
bsize_il1=32
repl=1
rm -f data1.dot
rm -f data2.dot
 echo -n "Quel est l'executable a executer ? " ; read program
for assoc in 1 2 4 8 16
do
        //configuration de cache de données dl1

  dl1_conf="-cache:dl1 dl1:${nsets_dl1}:${bsize_dl1}:${assoc}:${repl}"

        //calculer la taille de cache dl1

  dl1_size=$((nsets_dl1*bsize_dl1*assoc/K))

       //simuler les programme éxecutable avec le simulateur sim-cahe

  echo "Configuration ${dl1_conf}     dl1 size:${dl1_size} Ko"
  sim-cache ${dl1_conf}  $program > output 2> /tmp/.sim-cache
  sed -e 's/ \+/ /g' /tmp/.sim-cache > /tmp/.sim-cache.new
  mv /tmp/.sim-cache.new /tmp/.sim-cache

       //etraits le nombre de misse dans le cache dl1

  dl1_misses=`grep -E "dl1.misses" /tmp/.sim-cache | cut -d ' ' -f 2`

       //affichage de nombre de misse

  echo "dl1_misses: ${dl1_misses}"

      //affichage de nombre de misse en fonction de l'associativité' de cache dl1
      //enregistré les resultats dans le fichier dat1.dat

  echo "${assoc} ${dl1_misses}" >> data1.dot
  echo -e "\n\n"
done

for assoc in 1 2 4 8 16
do

        //configuration de cache de données il1

  il1_conf="-cache:il1 il1:${nsets_il1}:${bsize_il1}:${assoc}:${repl}"

        //calculer la taille de cache il1

  il1_size=$((nsets_il1*bsize_il1*assoc/K))
  echo "Configuration ${il1_conf}     il1 size:${il1_size} Ko"

       //simuler les programme éxecutable avec le simulateur sim-cahe

  sim-cache ${il1_conf}   $program > output 2> /tmp/.sim-cache
  sed -e 's/ \+/ /g' /tmp/.sim-cache > /tmp/.sim-cache.new
  mv /tmp/.sim-cache.new /tmp/.sim-cache

       //affichage de nombre de misse

  il1_misses=`grep -E "il1.misses" /tmp/.sim-cache | cut -d ' ' -f 2`

      //affichage de nombre de misse en fonction de l'associativité' de cache il1
      //enregistré les resultats dans le fichier dat1.dat

  echo "il1_misses: ${il1_misses}"
  echo "${assoc} ${il1_misses}" >> data2.dot
  echo -e "\n\n"
```

```
77
78          //tracer les courbes des résultats sous l'outils gnuplot' avec l'extension png'
79
80  export GNUTERM=png
81
82         //spécifié certains paramétres pour tracer les graphes
83
84  echo "set output 'dl1_misses_assoc.png'; set xlabel textcolor rgb 'blue' 'dl1_assoc';
85  set ylabel textcolor rgb 'blue' 'misses_dl1';plot 'data1.dot' title 'dl1:misses/dl1_assoc'
86   with linespoints lc rgb'red';
87  set output 'il1_misses_assoc.png';set xlabel textcolor rgb 'blue' 'il1_assoc'
88  ;set ylabel textcolor rgb 'blue' 'misses_il1';plot 'data2.dot'
89  title 'il1:misses/il1_assoc' with linespoints lc rgb'red'; "| gnuplot  '-'
90  /usr/bin/display-im6 dl1_misses_assoc.png
91  /usr/bin/display-im6 il1_misses_assoc.png
92
93
94  exit 0;
95
```

FIGURE 4.1 – Le script shell correspondant aux configuration de cache L1 en cas de l'associativité

Le première paramètre que nous allons étudier est le nombre de défauts de cache en fonction du nombre d'ensemble (nsets), la taille de bloc (bsize) et l'associativité (assoc) du cache L1 et L2.

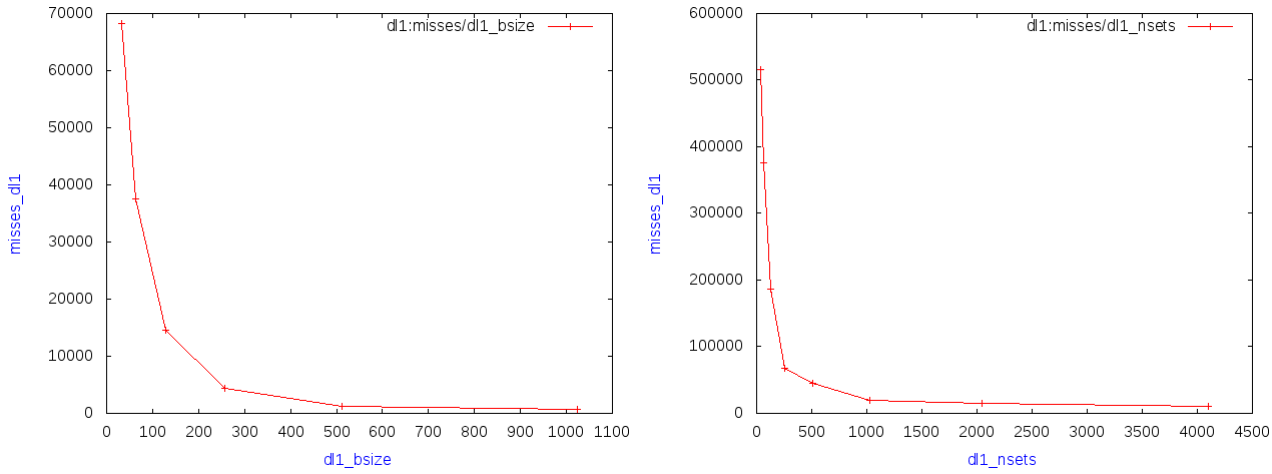



FIGURE 4.2 – Évolution du nombre de misses de cache dl1 en fonction de nsets et bsize

Les résultats représentées dans la figure 4.2 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache de données dl1 (première graphe "dl1_bsize") avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est varié.

-Cache de données dl1 (deuxième graphe "dl1_nsets" ) avec la taille de bloc de 32octets, l'associativité 1, politique de remplacement LRU, le nombre d'ensemble est varié.

-Cache d'instructions il1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est varié.

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU.

On remarque que l'évolution du nombre de défauts de cache dl1 en fonction de la taille de bloc est globalement importants, il donne une résultat satisfaisante qui abaisse le nombre de défauts (nombre de misses). D'après ces résultats la taille de bloc pour un cache de données dl1 est 512 octets.

On remarque aussi que l'évolution du nombre de défauts de cache dl1 en fonction de nombre d'ensemble est aussi très importants, il donne une résultat qui abaisse le nombre de défauts de cache (nombre de misses). D'après ces résultats le nombre d'ensemble du cache de données dl1 est 2048 octets.

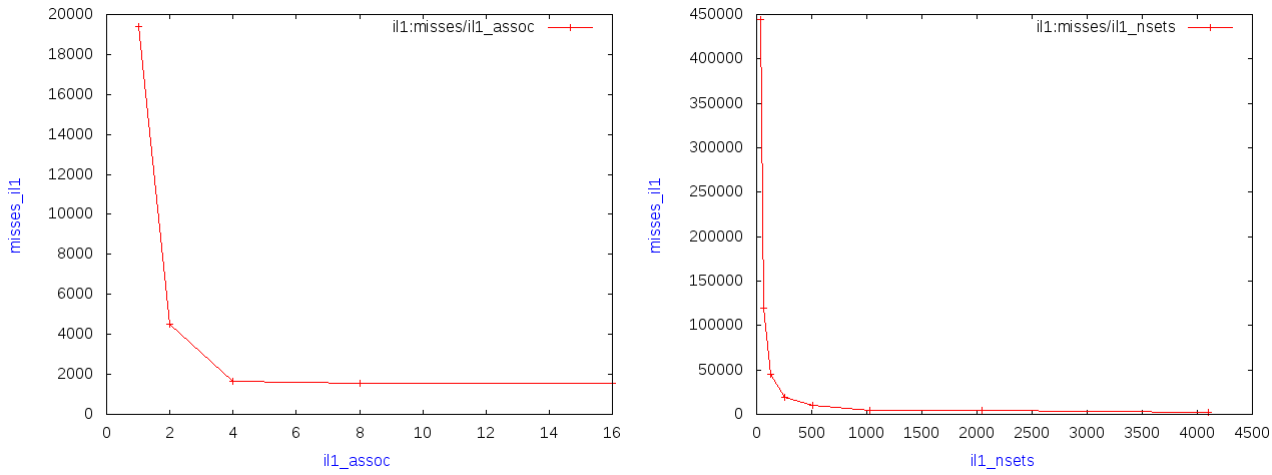



FIGURE 4.3 – Évolution du nombre de misses de cache il1 en fonction de nsets et assoc

Les résultats représentées dans la figure 4.3 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache d'instructions il1 (première graphe "il1_assoc") avec nombre d'ensemble de 256 octets, la taille de bloc 32 octets, politique de remplacement LRU, l'associativité est varié.

-Cache d'instructions il1 (deuxième graphe "il1_nsets") avec la taille de bloc de 32 octets, l'associativité 1, politique de remplacement LRU, le nombre d'ensemble est varié.

-Cache de données dl1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est 32 octets.

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU.

On remarque que l'évolution du nombre de défauts de cache il1 en fonction de l'associativité est importants, il donne la meilleur résultat pour l'associativité d'une mémoire cache qui abaisse le nombre de défauts de cache (nombre de misses). D'après ces résultats la meilleurs associativité de cache d'instructions il1 est 8.

On remarque aussi que l'évolution du nombre de défauts de cache il1 en fonction de nombre d'ensemble est aussi très importants, On a obtenu une meilleur résultat qui abaisse le nombre de défauts de cache (nombre de misses). D'après ces résultats la meilleur nombre d'ensemble du cache d'instructions il1 est 1024 octets.

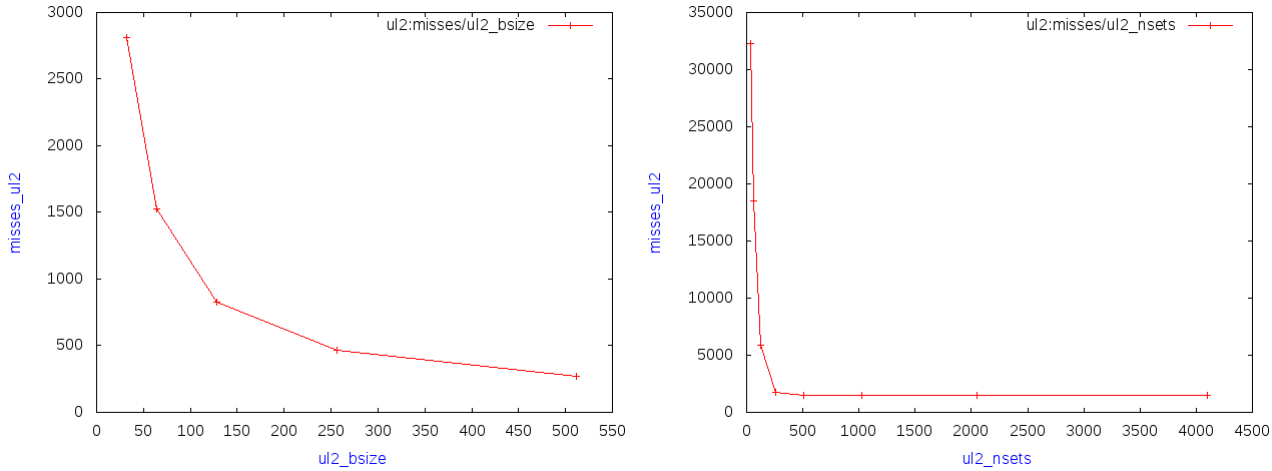



FIGURE 4.4 – Évolution du nombre de misses de cache ul2 en fonction de nsets et bsize

Les résultats représentées dans la figure 4.4 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets, l'associativité 4, politique de remplacement LRU, la taille de bloc est varié (première graphe "ul2_bsize").

-Cache L2 unifié (données+instructions) avec un taille de bloc de 64 octets, l'associativité 4, politique de remplacement LRU, nombre d'ensemble est varié (deuxième graphe "ul2_nsets").

-Cache de données dl1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est 32 octets.

-Cache d'instructions il1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc 32 octets.

On remarque que l'évolution du nombre de défauts de cache ul2 en fonction de la taille de bloc décroît quand la taille de bloc croit et d'après ces résultats on conclu que la meilleurs taille de bloc pour le cache ul2 est 256 octets.

On remarque aussi que l'évolution du nombre de défauts de cache ul2 en fonction de nombre d'ensemble est très importants, On a obtenu une meilleur résultat qui abaisse le nombre de défauts de cache (nombre de misse). D'après ces résultats la meilleurs nombre d'ensemble du cache unifié ul2 est 1024 octets.

#### 4.5.1.2 Nombre de remplacements

Le deuxième paramètre que nous allons étudier est le nombre de remplacements en fonction du nombre d'ensemble (nsets), la taille de bloc (bsize) et l'associativité (assoc) du cache de premier niveau et deuxième niveau.

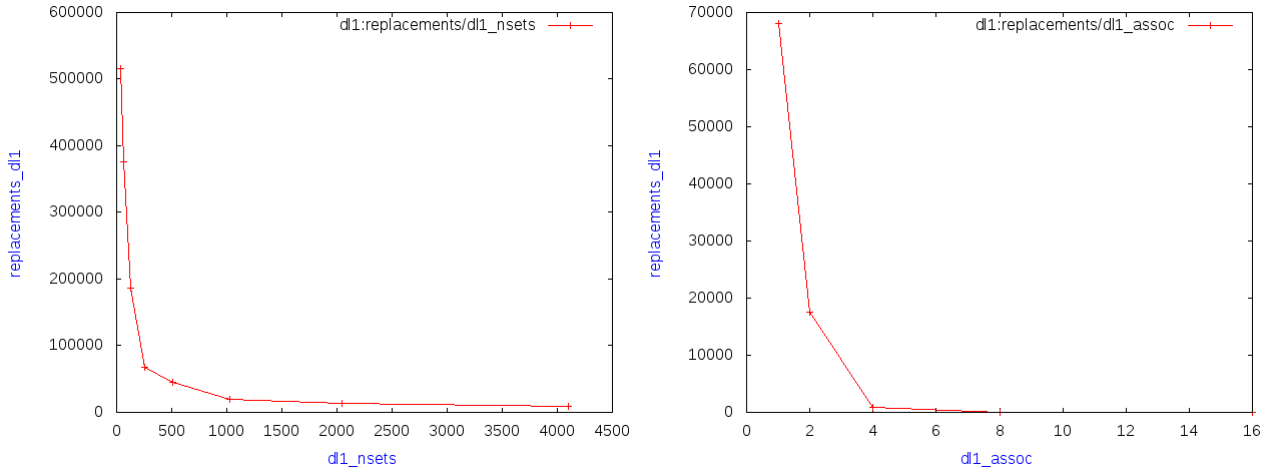



FIGURE 4.5 – Évolution du nombre de remplacement de cache dl1 en fonction de nsets et assoc

Les résultats représentées dans la figure 4.5 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache de données dl1 (première graphe "dl1_nsets") avec la taille de bloc 32 octets, l'associativité 1, politique de remplacement LRU, nombre d'ensemble est varié.

-Cache de données dl1 (deuxième graphe "dl1_assoc") avec la taille de bloc de 32octets, le nombre d'ensemble 256 octets, politique de remplacement LRU, l'associativité est varié.

-Cache d'instructions il1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est varié.

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU.

On peut remarquer qu'en même temps que la taille de bloc augmente, le nombre de remplacement diminue et par là on conclu que la meilleur résultat pour un nombre d'ensemble de cache de données dl1 est 2048 octets.

On remarque aussi que l'évolution du nombre de remplacements de cache dl1 en fonction l'associativité est aussi très importants, Et d'après ces tests on a obtenu une meilleur résultat pour l'associativité d'un cache de données dl1 est 8.

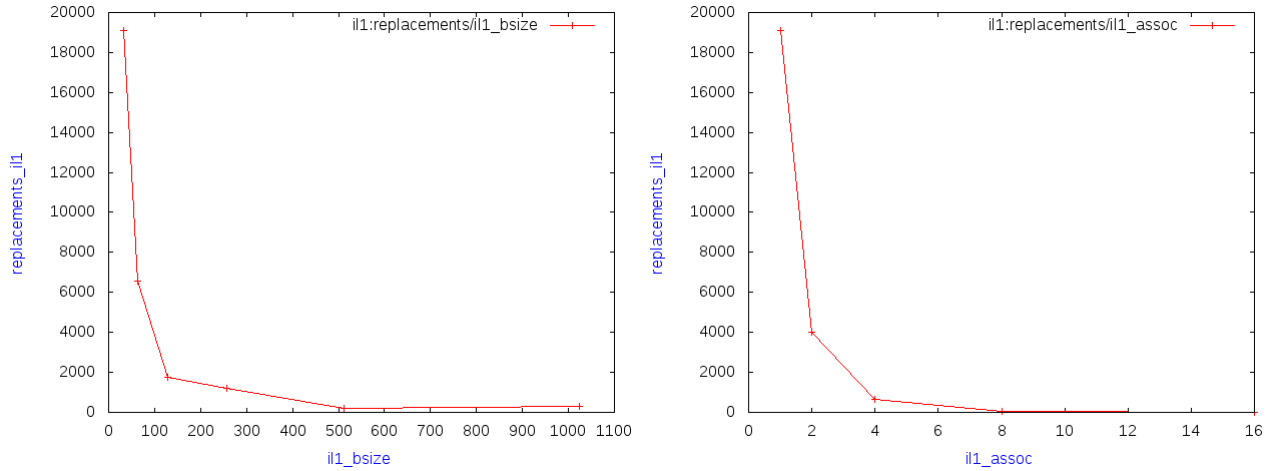



FIGURE 4.6 – Évolution du nombre de remplacement de cache il1 en fonction de bsize et assoc

Les résultats représentées dans la figure 4.6 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache de données dl avec la taille de bloc 32 octets, l'associativité 1, politique de remplacement LRU, nombre d'ensemble 256.

-Cache d'instructions il1 (première graphe "dl1_bsize") avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est varié.

-Cache de d'instructions il1 (deuxième graphe "dl1_assoc") avec la taille de bloc de 32octets, le nombre d'ensemble 256 octets, politique de remplacement LRU, l'associativité est varié.

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU.

D'après les graphes on remarque que l'évolution de nombre de remplacement de cache il1 diminue en temps que la taille de bloc augmente donc de ces résultats on conclu que la meilleur résultat pour la taille de bloc est 512 octets.

On remarque aussi que l'évolution du nombre de remplacements de cache il1 en fonction l'associativité est aussi très importants, Et d'après ces tests on a obtenu une meilleur résultat pour l'associativité d'un cache d'instructions il1 est 8.

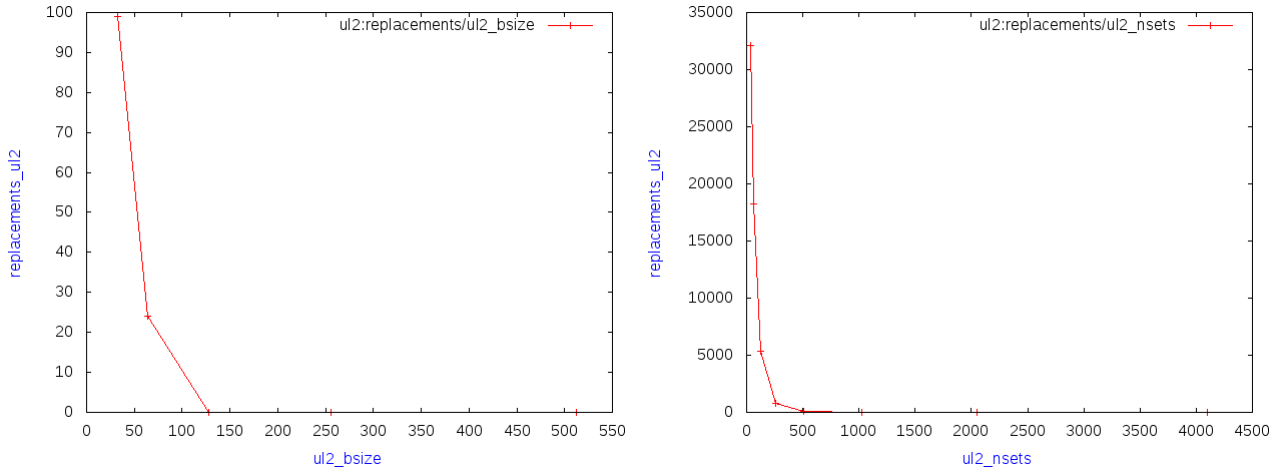



FIGURE 4.7 – Évolution du nombre de remplacement de cache ul2 en fonction de nsets et bsize

Les résultats représentées dans la figure 4.7 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets, l'associativité 4, politique de remplacement LRU, la taille de bloc est varié (première graphe "ul2_bsize").

-Cache L2 unifié (données+instructions) avec un taille de bloc de 64 octets, l'associativité 4, politique de remplacement LRU, nombre d'ensemble est varié (deuxième graphe "ul2_nsets").

-Cache de données dl1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est 32 octets.

-Cache d'instructions il1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc 32 octets.

On remarque que l'évolution du nombre de remplacement de cache ul2 en fonction de la taille de bloc décroît quand la taille de bloc croît et d'après ces résultats on conclu que la meilleur taille de bloc pour le cache ul2 est 128 octets.

On remarque aussi que l'évolution du nombre de remplacement de cache ul2 en fonction de nombre d'ensemble est très importants, On a obtenu une meilleur résultat qui abaisse le nombre de défauts de cache (nombre de misses). D'après ces résultats le meilleur nombre d'ensemble du cache unifié ul2 est 512 octets.

#### 4.5.1.3 Le taux de succès(Hit ration)

Le troisième paramètre que nous allons étudier est le taux de succès en fonction du nombre d'ensemble (nsets), la taille de bloc (bsize) et l'associativité (assoc) du cache de premier niveau et deuxième niveau.

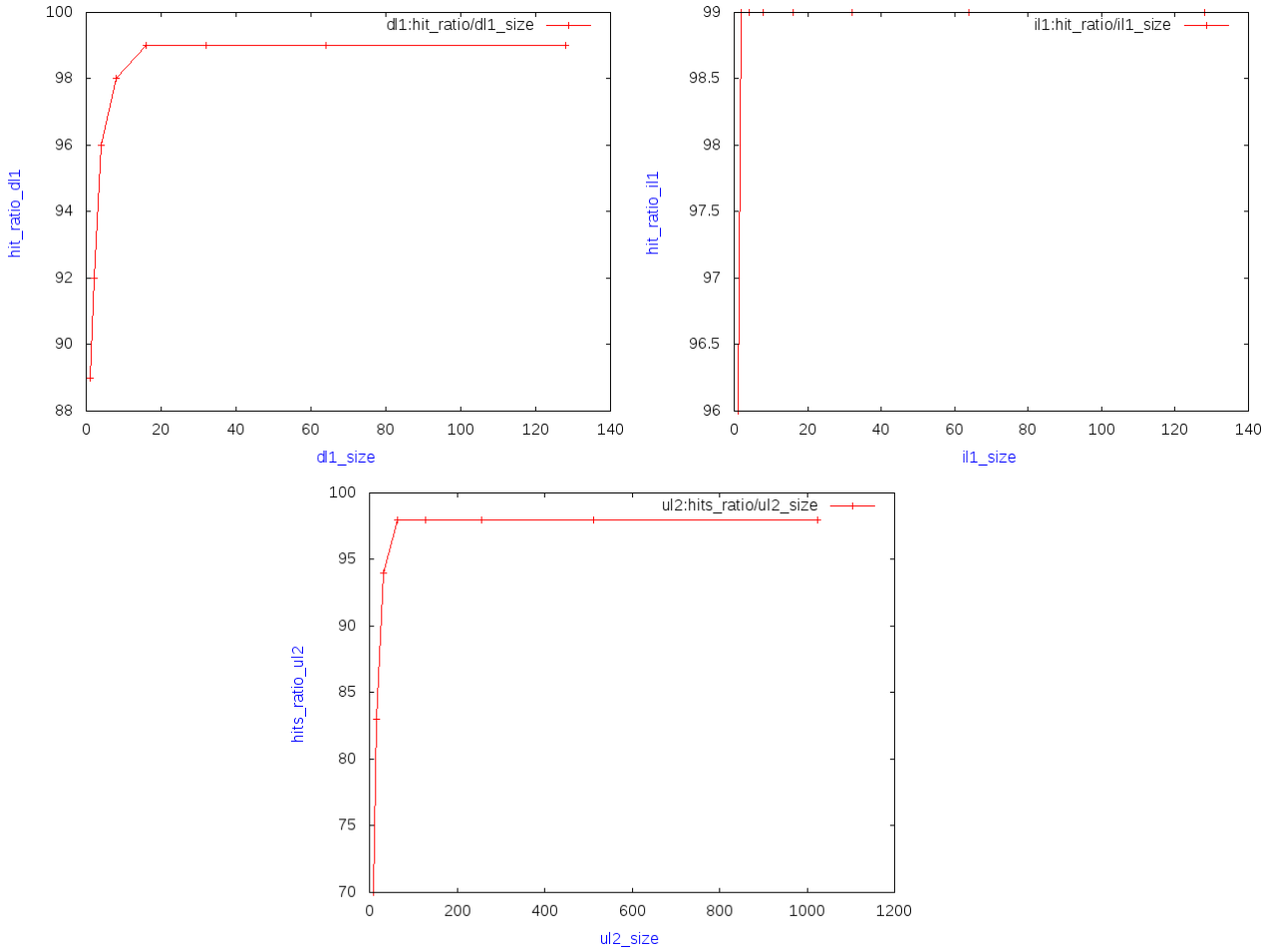



FIGURE 4.8 – Évolution du taux de succès de cache en fonction de la taille de cache L1 et L2

Nous remarquons dans la figure 4.8 que plus nous augmentons la taille du cache du L1et L2, plus le taux de succès augmente, cette augmentation est expliquée par le nombre d'accès au cache qui s'accroît avec une taille plus importante. Dans ce cas, plusieurs parcours sont nécessaires pour trouver la donnée recherchée.

Ainsi, Augmenter la taille du cache implique également l'augmentation du taux de succès sur le cache et vice versa. Augmenter la taille du cache améliore forcément les performances de cache.

## 4.5.2 Mibench

Les programmes de tests utilisés pour évaluer les performances sont extraits de la suite de programmes de tests Mibench et le programme utilisé pour valider les résultats est le benchmark dijkstra_small.

### 4.5.2.1 Nombre de misses(nombre de défauts)

Le première paramètre que nous allons étudier est nombre de défauts de cache en fonction du nombre d'ensemble (nsets), la taille de bloc (bsize) et l'associativité (assoc)

du cache L1 et L2.

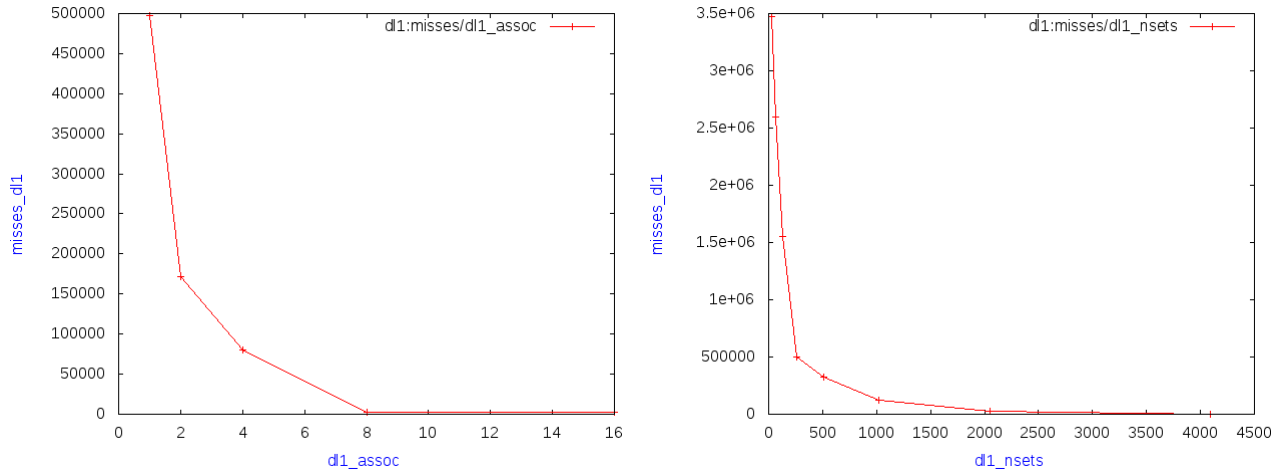



FIGURE 4.9 – Évolution du nombre de misses de cache dl1 en fonction de nsets et assoc

Les résultats représentées dans la figure 4.9 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache de données dl1 (première graphe "dl1_assoc") avec nombre d'ensemble de 256 octets, la taille de bloc est 32 octets , politique de remplacement LRU, l'associativité est varié.

-Cache de données dl1 (deuxième graphe "dl1_nsets") avec la taille de bloc de 32octets, l'associativité 1, politique de remplacement LRU, le nombre d'ensemble est varié.

-Cache d'instructions il1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est varié.

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU.

On remarque que l'évolution du nombre de défauts de cache dl1 en fonction de la taille de bloc est globalement importants, d'après les résultats obtenu dans le premier graphe de la figure 4.9 le nombre de défauts décroît en temps que l'associativité augmente jusqu'à l'arrivée a la meilleur résultats qui est 8.

On remarque aussi que l'évolution du nombre de défauts de cache dl1 en fonction de nombre d'ensemble est aussi très importants, d'après le le deuxième graphe de la figure 4.9 nous remarquons que le nombre de défauts décroît en temps que le nombre d'ensemble croît. Et d'après ces résultats la meilleur nombre d'ensemble du cache de données dl1 est 2048 octets.

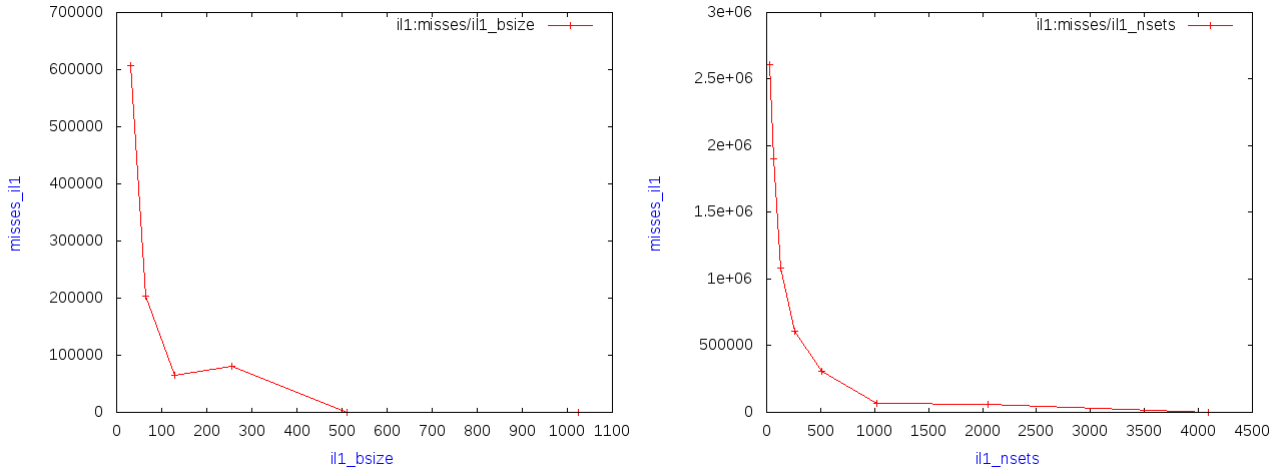



FIGURE 4.10 – Évolution du nombre de misses de cache il1 en fonction de bsize et nsets

Les résultats représentées dans la figure 4.10 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache d'instructions il1 (première graphe "il1_bsize") avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est varié.

-Cache d'instructions il1 (deuxième graphe "il1_nsets") avec la taille de bloc de 32 octets, l'associativité 1, politique de remplacement LRU, le nombre d'ensemble est varié.

-Cache de données dl1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est 32 octets.

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU.

La figure 3.11 montre l'évolution du nombre de défauts de cache il1 en fonction de l'associativité est importants, ou on remarquons que le nombre de défauts se diminue en même temps que la taille de bloc s'augmente ce que signifie plus que la taille de bloc est grand, le nombre de défauts décroît. D'après ces résultats la meilleur taille de bloc est 512 octets.

On remarque aussi que l'évolution du nombre de défauts de cache il1 en fonction de nombre d'ensemble est aussi très importants, On a obtenu une meilleur résultat qui abaisse le nombre de défauts de cache (nombre de misses). D'après ces résultats la meilleur nombre d'ensemble du cache d'instructions il1 est 2048 octets.

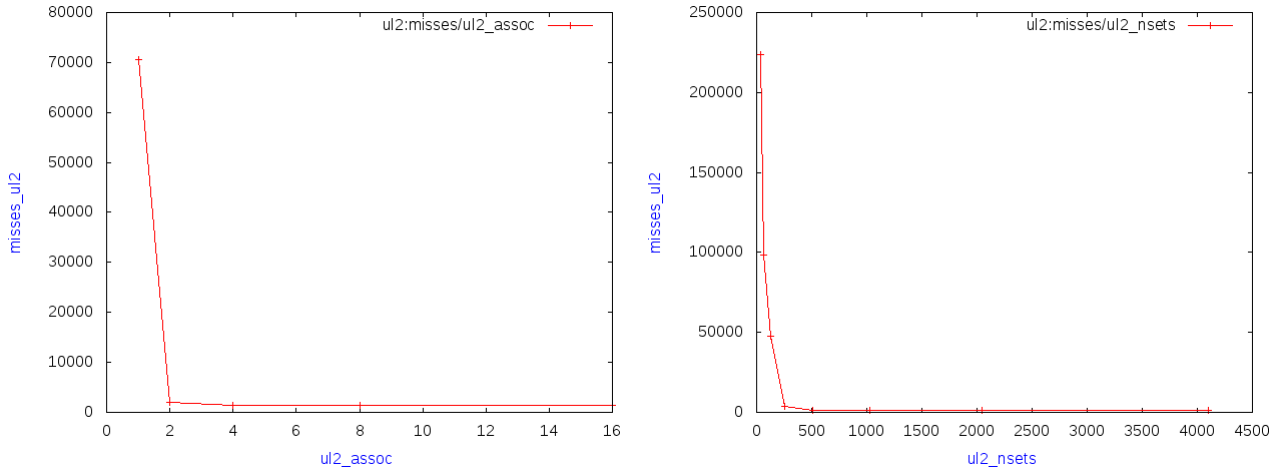



FIGURE 4.11 – Évolution du nombre de misses de cache ul2 en fonction de nsets et assoc

Les résultats représentées dans la figure 4.11 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, politique de remplacement LRU, l'associativité est varié.

-Cache L2 unifié (données+instructions) avec la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU, un nombre d'ensemble est varié.

-Cache de données dl1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est 32 octets.

-Cache d'instructions il1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc 32 octets.

On remarque que l'évolution du nombre de défauts de cache ul2 en fonction de l'associativité décroît quand l'associativité croit. Et d'après ces résultats on conclu que la meilleur associativité pour le cache ul2 est 4.

On remarque aussi que l'évolution du nombre de défauts de cache ul2 en fonction de nombre d'ensemble est très importants dans le deuxième graphe "ul2_nsets", on voit que le nombre de défauts diminue en temps que le nombre d'ensemble s'augmente. D'après ces résultats la meilleur nombre d'ensemble du cache unifié ul2 est 512 octets.

#### 4.5.2.2 Nombre de remplacements

Le deuxième paramètre que nous allons étudier est nombre de remplacements de cache en fonction du nombre d'ensemble (nsets), la taille de bloc (bsize) et l'associativité (assoc) du cache L1 et L2.

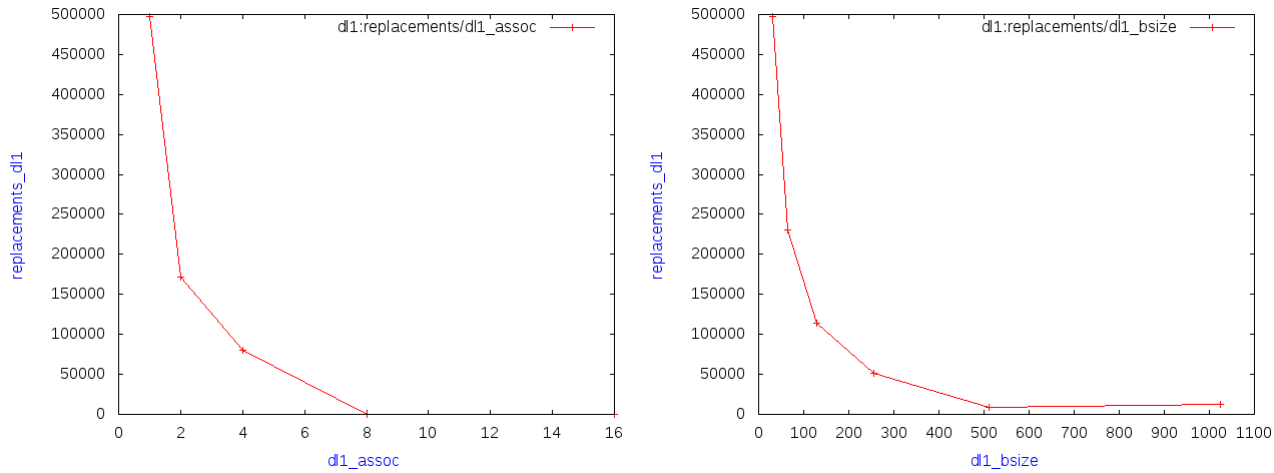



FIGURE 4.12 – Évolution du nombre de remplacement de cache dl1 en fonction de bsize et assoc

Les résultats représentées dans la figure 4.12 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache de données dl1 (première graphe "dl1_assoc") avec la taille de bloc de 32octets, le nombre d'ensemble 256 octets, politique de remplacement LRU, l'associativité est varié.

-Cache de données dl1 (deuxième graphe "dl1_bsize") avec un nombre d'ensemble 32 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est varié.

-Cache d'instructions il1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est varié.

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU.

On peut remarquer qu'en même temps que l'associativité augmente, le nombre de remplacement diminue et par là on conclu que la meilleur résultat pour un l'associativité de cache de données dl1 est 8 d'après les résultats obtenu dans le premier graphe de la figure 4.12.

On remarque aussi que l'évolution du nombre de remplacements de cache dl1 en fonction de la taille de bloc est aussi très importants, Et d'après ces tests qu'on a obtenu dans le deuxième graphe une meilleur résultat pour la taille de bloc d'un cache de données dl1 est 512 octets.

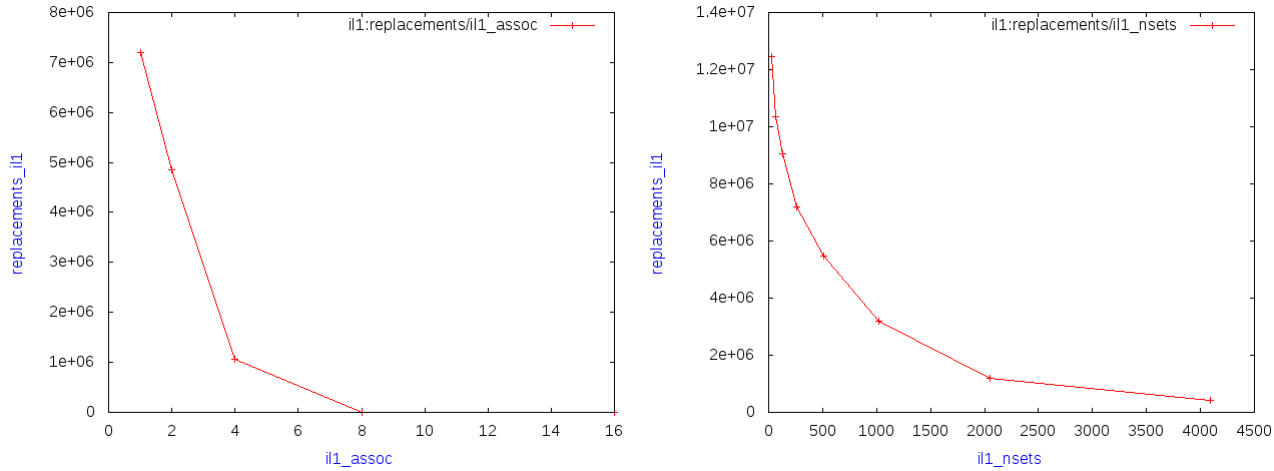



FIGURE 4.13 – Évolution du nombre de remplacement de cache il1 en fonction de nsets et assoc

Les résultats représentées dans la figure 4.13 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache de données dl1 avec la taille de bloc 32 octets, l'associativité 1, politique de remplacement LRU, nombre d'ensemble 256.

-Cache de d'instructions il1 (premier graphe "il1_assoc") avec la taille de bloc de 32octets, le nombre d'ensemble 256 octets, politique de remplacement LRU, l'associativité est varié.

-Cache d'instructions il1 (deuxième graphe "il1_nsets") avec un taille de bloc de 32 octets, l'associativité 1, politique de remplacement LRU, nombre d'ensemble est varié.

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets,la taille de bloc 64 octets, l'associativité 4, politique de remplacement LRU.

D'après les graphes on remarque que l'évolution de nombre de remplacement de cache il1 diminue en temps que l'associativité augmente donc de ces résultats on conclu que la meilleur résultat pour l'associativité est 8 .

On remarque aussi que l'évolution du nombre de remplacements de cache il1 en fonction de l'associativité est aussi très importants, Et d'après ces tests on a obtenu une meilleur résultat pour l'associativité d'un cache d'instructions il1 est 4096 octets.

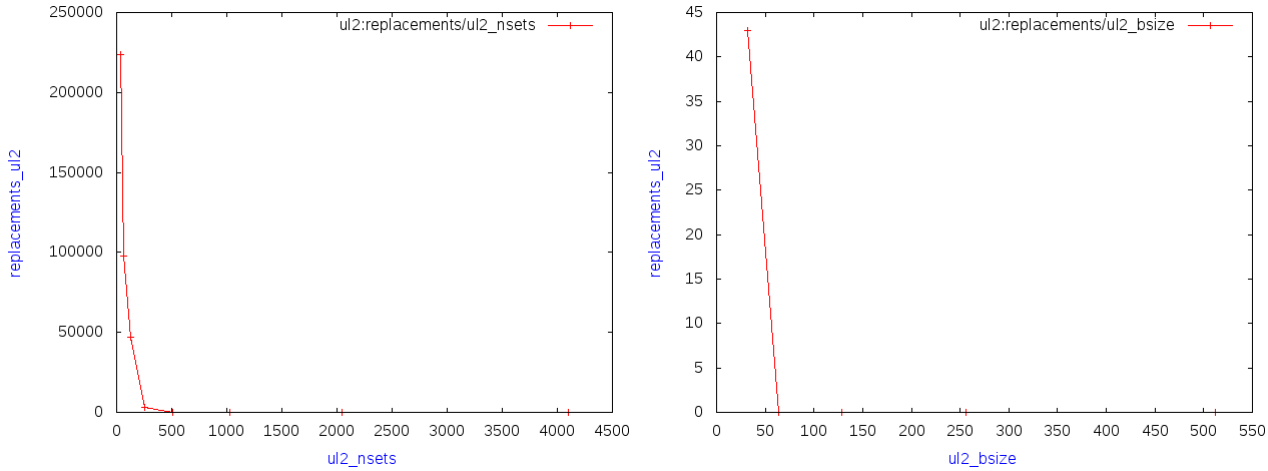



FIGURE 4.14 – Évolution du nombre de remplacement de cache ul2 en fonction de nsets et bsize

Les résultats représentées dans la figure 4.14 ont été obtenus avec la hiérarchie mémoire suivante :

-Cache L2 unifié (données+instructions) avec un taille de bloc de 64 octets, l'associativité 4, politique de remplacement LRU, nombre d'ensemble est varié (première graphe "ul2_nsets").

-Cache L2 unifié (données+instructions) avec un nombre d'ensemble de 1024 octets, l'associativité 4, politique de remplacement LRU, la taille de bloc est varié (deuxième graphe "ul2_bsize").

-Cache de données dl1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc est 32 octets.

-Cache d'instructions il1 avec nombre d'ensemble de 256 octets, l'associativité 1, politique de remplacement LRU, la taille de bloc 32 octets.

On remarque que l'évolution du nombre de remplacement de cache ul2 en fonction de nombre d'ensemble est globalement importants.Dans le première graphe on remarquons que le nombre de remplacement décroît en même temps que le nombre d'ensemble croit se que signifié plus que le nombre d'ensemble est grand plus que la donnée dans le cache et d'après les résultats le nombre d'ensemble est 512 pour le cache unifié ul2.

On remarque aussi que l'évolution du nombre de remplacement de cache ul2 en fonction de la taille de bloc est très importants, d'après le deuxième graphe de la figure 4.14 on remarquons que le nombre de remplacement se diminue en temps que la taille de bloc s'augmente. D'après ces résultats la meilleur valeur pour la taille de bloc du cache unifié ul2 est 64 octets pour ce benchmarks.

#### 4.5.2.3 Le taux de succès(Hit ration)

Le troisième paramètre que nous allons étudier est le taux de succès de cache en fonction du nombre d'ensemble (nsets), la taille de bloc (bsize) et l'associativité (assoc) du cache de premier niveau et deuxième niveau.

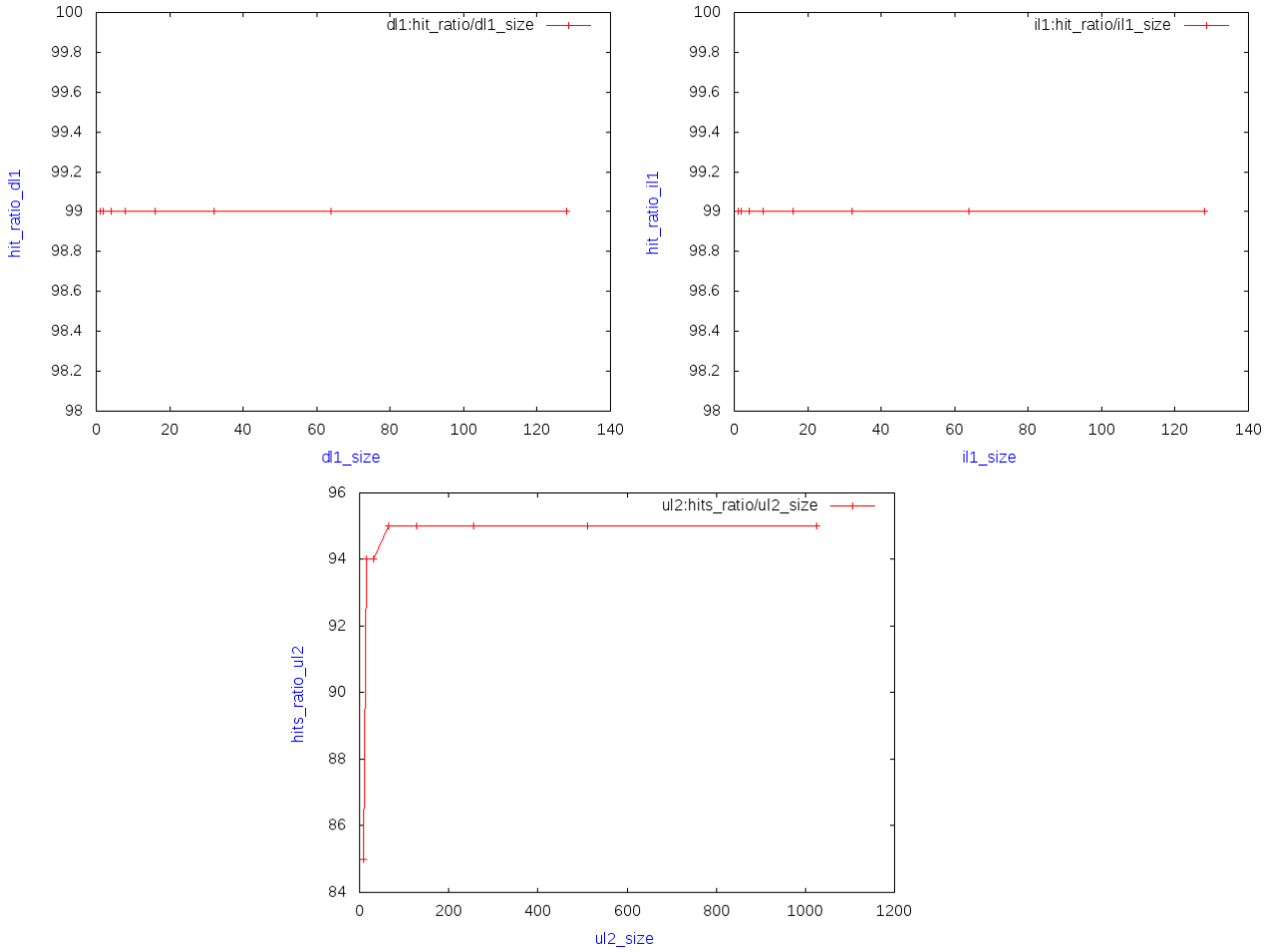



FIGURE 4.15 – Évolution du taux de succès de cache en fonction de la taille de cache L1 et L2

Nous remarquons dans la figure 4.15 que plus nous augmentons la taille du cache du L1et L2, plus le taux de succès augmente, cette augmentation est expliquée par le nombre d'accès au cache qui s'accroît avec une taille plus importante. ce qu'on peut remarquer d'après les trois graphes de la simulation c'est que globalement le taux de succès donne 99% pour le cache L1 et L2.

Ainsi, Augmenter la taille du cache implique également l'augmentation du taux de succès sur le cache et vice versa. Augmenter la taille du cache améliore forcément les performances de cache.

## 4.6 La meilleur configuration de cache

Dans cette section nous allons donner les résultats des tests que nous avons fait sur les programmes de mediabench et mibench ainsi que choisir la meilleur configuration pour les trois paramétres de cache. Cette section est subdivisée en deux parties. Dans la première nous allons voir les résultats des programmes de test utilisés pour les différents paramétres de cache L1 et L2 (nsets, bsize, assoc) et aussi la taille de cache . Dans la deuxième on

extraira les configurations optimales selon ces trois performances de cache : nombre de de défauts (nombre de misses), nombre de remplacement et le taux de succès.

### 4.6.1 Les résultats des tests

Les benchmarks testé dans notre travail sont `cjpeg, djpeg, rawcaud, rawdaud, gsmdecode, gsmencode, pegwitdecode et pegwitencode` pour la suite Mediabench et `sha, dijkstra_small, dijkstra_large, blowfish, rijndael, ispell et patricia` pour la suite Mibench.

#### 4.6.1.1 Mediabench

Les résultats des tests que nous avons fait sur les programme exécutable de la suite mediabench sont montré dans les trois tableaux suivants :

- `Nombre de misses (nombre de défauts)`

| Benchmarks/paramétres de cache | dl1 | | | il1 | | | ul2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | nsets | bsize | assoc | nsets | bsize | assoc | nsets | bsize | assoc |
| cjpeg | 2084 | 512 | 6 | 1024 | 512 | 6 | 2048 | 512 | 6 |
| djpeg | 2048 | 512 | 4 | 1024 | 256 | 4 | 512 | 512 | 4 |
| gsmdecode | 2048 | 512 | 2 | 2048 | 1024 | 4 | 512 | 512 | 2 |
| gsmencode | 2048 | 512 | 2 | 1024 | 1024 | 4 | 512 | 512 | 2 |
| pegwitdecode | 2048 | 1024 | 4 | 1024 | 1024 | 4 | 512 | 512 | 2 |
| pegwitencode | 2048 | 1024 | 4 | 2048 | 1024 | 4 | 512 | 512 | 2 |
| rawcaud | 1024 | 256 | 4 | 1024 | 1024 | 4 | 256 | 64 | 2 |
| rawdaud | 1024 | 256 | 4 | 1024 | 1024 | 4 | 256 | 64 | 2 |

TABLE 4.2 – Les résultats optimales extraits de la suite mediabench en fonction de nombre de misses

- Nombre de remplacement

| Benchmarks/paramétres de cache | dl1 | | | il1 | | | ul2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | nsets | bsize | assoc | nsets | bsize | assoc | nsets | bsize | assoc |
| cjpeg | 2048 | 512 | 6 | 2048 | 512 | 8 | 1024 | 128 | 4 |
| djpeg | 2048 | 512 | 6 | 2048 | 256 | 8 | 512 | 64 | 4 |
| gsmdecode | 2048 | 512 | 6 | 2048 | 256 | 4 | 512 | 32 | 2 |
| gsmencode | 2048 | 512 | 6 | 2048 | 256 | 4 | 512 | 32 | 2 |
| pegwitdecode | 2048 | 512 | 4 | 2048 | 512 | 4 | 512 | 256 | 2 |
| pegwitencode | 2048 | 512 | 4 | 2048 | 512 | 4 | 512 | 256 | 2 |
| rawcaud | 2048 | 128 | 4 | 4096 | 512 | 4 | 256 | 64 | 4 |
| rawdaud | 2048 | 128 | 4 | 4096 | 512 | 4 | 256 | 32 | 4 |

TABLE 4.3 – Les résultats optimales extraits de la suite mediabench en fonction de nombre d'ensemble

Les tables 4.2 et 4.3 résume les différents résultats de simulation de la suite mediabench. la première colonne liste des programme (benchmarks). Les autres colonne résume les paramétres de cache L1 et L2 .

- `Taux de succès`

| Benchmarks/taille de cache | dl1 | il1 | ul2 |
|---|---|---|---|
| cjpeg | 32 ko | 2 ko | 64 ko |
| djpeg | 32 ko | 4 ko | 64 ko |
| gsmdecode | 32 ko | 8 ko | 32 ko |
| gsmencode | 32 ko | 8 ko | 32 ko |
| pegwitdecode | 32 ko | 8 ko | 16 ko |
| pegwitencode | 32 ko | 8 ko | 16 ko |
| rawcaud | 16 ko | 8 ko | 16 ko |
| rawdaud | 16 ko | 8 ko | 16 ko |

TABLE 4.4 – Les résultats optimales extraits de la suite mediabench en fonction de taux de succès

La table 4.4 liste les tailles de cache L1 et L2 obtenus après la simulation des programmes mediabench.

#### 4.6.1.2 Mibench

Les résultats des tests que nous avons fait sur les programme exécutable de la suite mibench sont montré dans les trois tableaux suivants :

- `Nombre de misses (nombre de défauts)`

| Benchmarks/paramétres de cache | dl1 | | | il1 | | | ul2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | nsets | bsize | assoc | nsets | bsize | assoc | nsets | bsize | assoc |
| blowfish | 512 | 1024 | 2 | 512 | 1024 | 4 | 256 | 512 | 2 |
| dijkstra_small | 2048 | 512 | 8 | 2048 | 512 | 4 | 512 | 512 | 4 |
| dijksta_large | 2048 | 512 | 8 | 2048 | 1024 | 4 | 1024 | 256 | 16 |
| ispell | 2048 | 1024 | 4 | 2048 | 1024 | 2 | 512 | 64 | 2 |
| patricia | 2048 | 1024 | 4 | 2048 | 512 | 8 | 512 | 512 | 4 |
| rijnadael | 1024 | 512 | 4 | 1024 | 512 | 4 | 512 | 256 | 2 |
| sha | 1024 | 512 | 4 | 512 | 512 | 4 | 512 | 256 | 2 |

TABLE 4.5 – Les résultats optimales extraits de la suite mibench en fonction de nombre de misses

- `Nombre de remplacement`

| Benchmarks/paramétres de cache | dl1 | | | il1 | | | ul2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | nsets | bsize | assoc | nsets | bsize | assoc | nsets | bsize | assoc |
| blowfish | 2048 | 512 | 8 | 2048 | 512 | 4 | 256 | 256 | 4 |
| dijkstra_small | 2048 | 512 | 8 | 2048 | 512 | 4 | 512 | 64 | 4 |
| dijksta_large | 2048 | 512 | 8 | 2048 | 512 | 4 | 512 | 256 | 6 |
| ispell | 2048 | 1024 | 4 | 2048 | 512 | 6 | 512 | 64 | 2 |
| patricia | 2048 | 512 | 8 | 2048 | 512 | 6 | 1024 | 256 | 6 |
| rijnadael | 2048 | 128 | 4 | 2048 | 128 | 4 | 256 | 32 | 2 |
| sha | 1024 | 512 | 4 | 512 | 256 | 6 | 256 | 32 | 2 |

TABLE 4.6 – Les résultats optimales extraits de la suite mibench en fonction de nombre d'ensemble

Les tables 4.5 et 4.6 résume les différents résultats de simulation de la suite mibench. la première colonne liste des programme (benchmarks). Les autres colonne résume les paramétres de cache L1 et L2 .

- `Taux de succès`

| Benchmarks/ taille de cache | dl1 | il1 | ul2 |
|---|---|---|---|
| blowfish | 16 ko | 16 ko | 16 ko |
| dijkstra_small | 8 ko | 4 ko | 64 ko |
| dijksta_large | 8 ko | 4 ko | 64 ko |
| ispell | 16 ko | 16 ko | 16 ko |
| patricia | 16 ko | 128 ko | 128 ko |
| rijnadael | 16 ko | 16 ko | 16 ko |
| sha | 32 ko | 16 ko | 16 ko |

TABLE 4.7 – Les résultats optimales extraits de la suite mediabench en fonction de taux de succès

La table 4.7 liste les tailles de cache L1 et L2 obtenus après la simulation des programmes mibench.

### 4.6.2 La configuration optimale retenue

La table 4.8 donne la meilleur configuration obtenus pour les trois paramétres et la taille de cache L1 et L2 : le première ligne représente le nombre d'ensemble, le seconde la taille de bloc, la troisième pour l'associativité et le quatrième colonne précise la taille de cache extraits à partir des résultats des programme exécute.

| | cache de données dl1 | cache d'instruction il1 | cache unifié ul2 |
|---|---|---|---|
| nsets | 2048 octets | 2048 octets | 512 octets |
| bsize | 512 octets | 512 octets | 256 octets |
| assoc | 4 | 4 | 4 |
| size | 32 ko | 8 ko | 32 ko |

TABLE 4.8 – la configuration optimale extraits de la suite mediabench et mibench pour le cache L1 et L2

Le script shell correspondant à la configuration optimale des caches L1 et L2 est illustré dans la figure 4.8 suivante :

```
#!/bin/bash

K=1024
bsize_dl1=512
bsize_il1=512
bsize_dl2=256
assoc_dl1=4
assoc_il1=4
assoc_dl2=4
nsets_dl1=2048
nsets_il1=2048
nsets_dl2=512
repl=l
 echo -n "Quel est l'executable a executer ? " ; read program

       //configuration de cache de données dl1

  dl1_conf="-cache:dl1 dl1:${nsets_dl1}:${bsize_dl1}:${assoc_dl1}:${repl}"

       //calculer la taille de cache dl1

  dl1_size=$((nsets_dl1*bsize_dl1*assoc_dl1/K))
  echo "Configuration ${dl1_conf}     dl1 size:${dl1_size} Ko"

      //simuler les programme éxecutable avec le simulateur sim-cahe

  sim-cache ${dl1_conf}  $program > output 2> /tmp/.sim-cache
  sed -e 's/ \+/ /g' /tmp/.sim-cache > /tmp/.sim-cache.new
  mv /tmp/.sim-cache.new /tmp/.sim-cache
       //calculer le taux de succés dans le cache il1

  dl1_accesses=`grep -E "dl1.accesses" /tmp/.sim-cache | cut -d ' ' -f 2`
  dl1_hit=`grep -E "dl1.hit" /tmp/.sim-cache | cut -d ' ' -f 2`
  hit_ratio_dl1=$((dl1_hit*100/dl1_accesses))

       //affichage de taux de succés

  echo "Hit ratio_dl1: ${hit_ratio_dl1}"

      //affichage de taux de succés en fonction de la taille de cache il1
      //enregistré les resultats dans le fichier dat3.dat

  echo "${dl1_size} ${hit_ratio_dl1}" >> data1.dot
  echo -e "\n\n"

     //configuration de cache dinstructions il1

  il1_conf="-cache:il1 il1:${nsets_il1}:${bsize_il1}:${assoc_il1}:${repl}"

       //calculer la taille de cache il1

  il1_size=$((nsets_il1*bsize_il1*assoc_il1/K))

       //affichage de la taille de cache il1 calculé

  echo "Configuration ${il1_conf}     il1 size:${il1_size} Ko"

       //simuler les programme éxecutable avec le simulateur sim-cahe

  sim-cache ${il1_conf}   $program > output 2> /tmp/.sim-cache
  sed -e 's/ \+/ /g' /tmp/.sim-cache > /tmp/.sim-cache.new
  mv /tmp/.sim-cache.new /tmp/.sim-cache

       //calculer le taux de succés dans le cache il1

  il1_accesses=`grep -E "il1.accesses" /tmp/.sim-cache | cut -d ' ' -f 2`
  il1_hit=`grep -E "il1.hit" /tmp/.sim-cache | cut -d ' ' -f 2`
  hit_ratio_il1=$((il1_hit*100/il1_accesses))

       //affichage de taux de succés

  echo "Hit ratio_il1: ${hit_ratio_il1}"

```

```
106         //affichage de taux de succés
107
108     echo "Hits ratio_ul2: ${hits_ratio_ul2}"
109
110         //affichage de taux de succés en fonction de la taille de cache dl2
111         //enregistré les resultats dans le fichier dat3.dat
112
113     echo "${dl2_size} ${hits_ratio_ul2}" >> data3.dot
114     echo -e "\n\n"
115
116
117          //tracer les courbes des résultats sous l'outils gnuplot' avec l'extension png'
118
119 export GNUTERM=png
120
121         //spécifié certains paramétres pour tracer les graphes
122
123 echo "set output 'dl1_hit_ratio_size.png'; set xlabel textcolor rgb 'blue' 'dl1_size';
124 set ylabel textcolor rgb 'blue' 'hit_ratio_dl1';plot 'data1.dot' title 'dl1:hit_ratio/
    dl1_size'
125 with linespoints lc rgb'red';
126 set output 'il1_hit_ratio_size.png';set xlabel textcolor rgb 'blue' 'il1_size';
127 set ylabel textcolor rgb 'blue' 'hit_ratio_il1';
128 plot 'data2.dot' title 'il1:hit_ratio/il1_size' with linespoints lc rgb'red';
129 set output 'ul2_hits_ratio_size.png'; set xlabel textcolor rgb 'blue' 'ul2_size';
130 set ylabel textcolor rgb 'blue' 'hits_ratio_ul2';
131 plot 'data3.dot'title 'ul2:hits_ratio/ul2_size' with linespoints lc rgb'red'"| gnuplot  '-'
132
133 /usr/bin/display-im6 dl1_hit_ratio_nsets.png
134 /usr/bin/display-im6 il1_hit_ratio_nsets.png
135 /usr/bin/display-im6 ul2_misses_ratio_nsets.png
136
137 exit 0;
```

FIGURE 4.16 – Le script de configuration finale de cache L1 et L2

### 4.6.3 Les résultats de la simulation de mediabench aprés la configuration de cache

La figure 4.17 montrent le fichier de données ".dat" qui contient les résultats de taux de succès de la simulation des programmes de la suite mediabench.

| 1 | benchmarks | taux_succés_dl1 | taux_succés_il1 | taux_succés_ul2 |
|---|---|---|---|---|
| 2 | cjpeg | 99% | 99% | 99% |
| 3 | rawcaud | 98% | 99% | 79% |
| 4 | gsmdecode | 98% | 99% | 80% |
| 5 | gsmencode | 98% | 99% | 80% |
| 6 | pegwitdecode | 98% | 99% | 78% |
| 7 | djpeg | 99% | 99% | 99% |
| 8 | pegwitencode | 98% | 99% | 78% |
| 9 | rawdaud | 98% | 99% | 79% |

FIGURE 4.17 – Le fichier mediabench.dat

La figure 4.18 montrent le script ".gnu" qui consiste à spécifié les paramétres par la quel on peut tracer les courbes des résultats de la suite mediabench aprés la configuration

de cache L1 et L2 sous gnuplot aprés avoir taper la commande load ”histogram.gnu” sur le terminal.

```
clear
reset
unset key
#titre de l'image
set output 'histogram2.png'
#taille de bloc de l'histogramme
set boxwidth 0.7 absolute
set style fill   solid 1.00 border lt -1
#difinir les clés de l'histogramme a droit de la figure
set key inside right top vertical Right noreverse noenhanced autotitle nobox
#difinir les couleurs des blocs de l'histogramme
set style histogram clustered gap 1 title textcolor lt -1
set datafile missing '-'
set xtics rotate out
#titre des axes X et Y
set xlabel textcolor rgb 'blue' 'la suite mediabench'
set ylabel textcolor rgb 'blue' 'taux de succés'
#titre de l'histogramme
set title "taux de succés des programmes mediabench aprés la configuration de cache L1 et L2 "
#difinire les données par un histogramme
set style data histogram
set style fill solid border
#difinir les blocs par des sous ensemble dans l'histogramme
set style histogram clustered
difinir les parametres des axes X et Y
set yrange [ 0.000 : 130. ] noreverse nowriteback
#charger les données a partir de fichier 'mediabench.dat'
plot for [COL=2:4] 'mediabench.dat' using COL:xticlabels(1) title columnheader
#utilisé les données de fichier 'mediabench.dat' a partir de la coloune 3
plot 'media.dat' using 3:xticlabels(1)
```

FIGURE 4.18 – Le script histogram.gnu pour mediabench

La figure 4.19 montre les résultats de taux de succès aprés avoir extraire la configuration optimale de cache L1 et L2 en utilisant 8 applications différentes de la suite mediabench. L'axe des X représente les différents benchmarks, et l'axe des Y représente le taux de succès. Nous pouvons constater que dans 80% des cas la configuration optimale est trouvée . Dans deux cas (cjpeg, djpeg) les résultats les plus haute est atteint en terme de taux de succès .

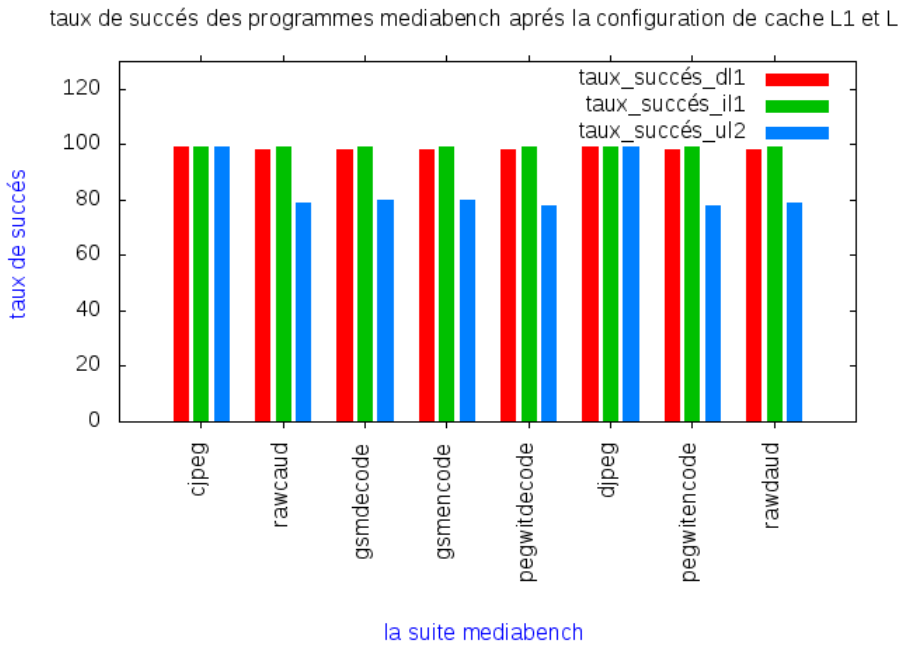



FIGURE 4.19 – taux de succès de la suite mediabench aprés la configuration de cache L1 et L2

### 4.6.4 Les résultats de la simulation de mibench aprés la configuration de cache

La figure 4.20 montrent le fichier de données ".dat" qui contient les résultats de taux de succès de la simulation des programmes de la suite mibench.

```
1  benchmarks        taux_succés_dl1     taux_succés_il1     taux_succés_ul2
2   dijkstra_small       99%                 99%                 99%
3   blowfish             98%                 99%                 79%
4
5   dijkstra_large       99%                 99%                 99%
6   ispell               98%                 99%                 76%
7   patricia             99%                 99%                 99%
8   rijndael             98%                 99%                 80%
9   sha                  99%                 99%                 98%
10
```

FIGURE 4.20 – Le fichier mibench.dat

La figure 4.21 montrent le script ".gnu" qui consiste a spécifié les paramétres par la quel on peut tracer les courbes des résultats de la suite mibench aprés la configuration de cache L1 et L2 sous gnuplot aprés avoir taper la commande load "histogram.gnu" sur le terminal.

```
clear
reset
unset key
#titre de l'image
set output 'histogram2.png'
#taille de bloc de l'histogramme
set boxwidth 0.7 absolute
set style fill   solid 1.00 border lt -1
#difinir les clés de l'histogramme a droit de la figure
set key inside right top vertical Right noreverse noenhanced autotitle nobox
#difinir les couleurs des blocs de l'histogramme
set style histogram clustered gap 1 title textcolor lt -1
set datafile missing '-'
set xtics rotate out
#titre des axes X et Y
set xlabel textcolor rgb 'blue' 'la suite mediabench'
set ylabel textcolor rgb 'blue' 'taux de succés'
#titre de l'histogramme
set title "taux de succés des programmes mibench aprés la configuration de cache L1 et L2 "
#difinire les données par un histogramme
set style data histogram
set style fill solid border
#difinir les blocs par des sous ensemble dans l'histogramme
set style histogram clustered
difinir les parametres des axes X et Y
set yrange [ 0.000 : 130. ] noreverse nowriteback
#charger les données a partir de fichier 'mibench.dat'
plot for [COL=2:4] 'mediabench.dat' using COL:xticlabels(1) title columnheader
#utilisé les données de fichier 'mibench.dat' a partir de la coloune 3
plot 'v.dat' using 3:xticlabels(1)
```

FIGURE 4.21 – taux de succès de la suite mediabench aprés la configuration de cache L1 et L2

La figure 4.22 montre les résultats de taux de succès aprés extraire la configuration optimale de cache L1 et L2 en utilisant 7 applications différentes de la suite mibench. L'axe des X représente les différents benchmarks, et l'axe des Y représente le taux de succès. Nous pouvons constater que dans 80% des cas la configuration optimale est trouvée . Dans trois cas (dijksta, sha, patricia) les résultats les plus haute est atteint en terme de taux de succès .

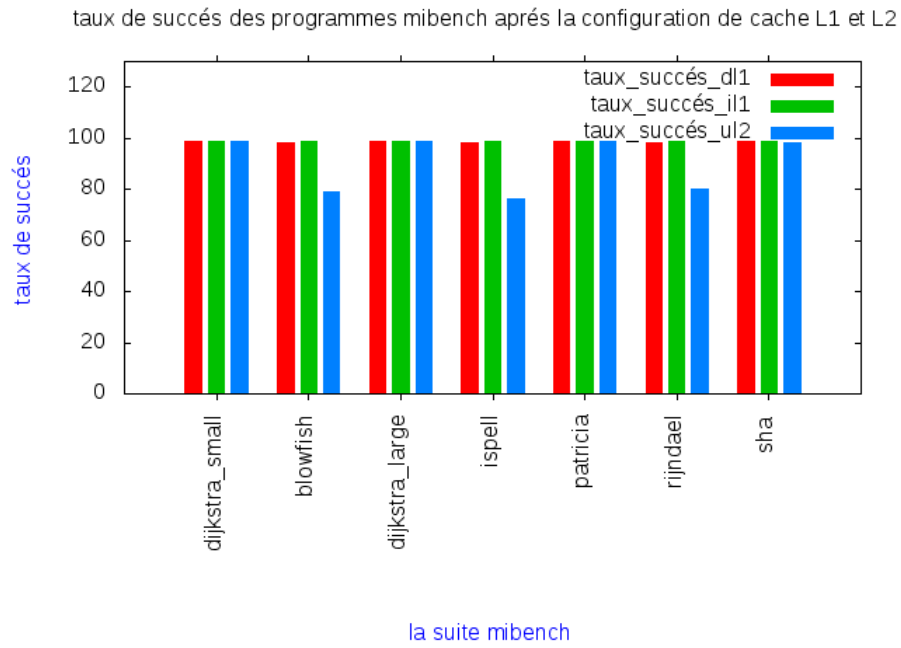



FIGURE 4.22 – taux de succès de la suite mediabench aprés la configuration de cache L1 et L2

## 4.7 Conclusion

Dans ce chapitre nous avons étudié l'évaluation expérimentale de l'influence de la configuration de cache mémoire L1 et L2 sur l'execution des programmes de la suite Mediabench et Mibench qui consiste à choisir une meilleur configuration qui nous permettons d'améliorer les performances de cache mémoire. Nous avons aussi extraits les résultats optimale aprés avoir fait plusieurs tests sur les benchmarks en utilisant le simulateur sim-cache de simplescalar . En fin nous avons présenté les résultats des évaluations sous forme des tableaux et des graphes sous l'outil gnuplot. Les résultats obtenu montrent que la configuration de cache mémoire à une influence importante sur l'exécution d'un programme.

# Conclusion générale

A la fin de projet on a pu concrétiser l'objectif fixé au départ à savoir, la détermination de la meilleur configuration de cache, comme le démontre les résultats des tests.

Cette étude nous a permis l'acquisition de notions très importantes voir indispensables pour l'accomplissement de notre formation. Nous avons appris beaucoup de connaissances sur le fonctionnement de cache, de telle sorte que, lorsque des données doivent être lues depuis la RAM, le matériel du système vérifie d'abord si les données requises sont en cache. Si tel est le cas, les données sont rapidement obtenues et utilisées par le CPU. En revanche, si les données ne sont pas en cache, elles sont lues depuis la mémoire vive (ou RAM) et, tout en étant transmises au CPU, sont également mises en cache (juste au cas où elles seraient nécessaires ultérieurement). Pour le CPU, tout le processus est effectué de manière transparente si bien que la seule différence entre l'accès aux données en cache et l'accès aux données en mémoire vive réside au niveau de la durée nécessaire à l'obtention des données souhaitées.

La taille de cache, le nombre d'ensembles, la taille de bloc et l'associativité sont des paramétres du cache qui détermine sa performance, plus ces paramétres atteignent un certain valeurs plus le cache est performant.

En effet nous avons acquis de plus amples connaissances avec le système d'exploitation kali linux, qui offre des outils de simulation (simplescalar) de programmation (script shell bash) et de visualisation (gnuplot), servant à mettre en évidence les performance du cache en terme d'accès à la mémoire donc au taux de succès de l'exécution d'un programme.

D'après les résultats obtenus, nous sommes dans la bonne voix sur les deux collections de tests Mediabench et Mibench, nous nous sommes limité pour ces tests à seulement deux niveaux de cache, en raison des contraintes imposé par le simulateur simplescalar v2.0 qui ne supporte que deux niveaux de cache.

Cela ouvre le champs à des perspectives, notamment l'implémentation d'un 3 ème niveau de cache au niveau du simulateur simplescalar.

Au delà des connaissances acquises cet expérience nous a permi de s'ouvrir à de nouveaux concepts sur le plan sociale et comportemental.

Le travail de groupe, l'organisation, la détermination et la communication sont des valeurs importants pour l'accomplissement intellectuelle, un atout décisif dans la vie professionnelle.

# Annexes

Cette annexe présente l'installation du simulateur simplescalar et le compilateur croisé GCC pour ARM ainsi que l'outil gnuplot utilisé pour tracer des courbes dans notre travail.

## A. Installation de Simplescalar

- a partir de http://www.simplescalar.com/ Télécharger les outils :

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

- installer le simplesim-arm en suivants les étapes ci dessous :

créez le répertoire "simplescalar" sous votre répertoire et copiez le simplesim-arm-0.2.tar.gz dans ce répertoire :

```
mkdir simplescalar
mv simplesim-arm-0.2.tar.gz simplescalar
cd simplescalar
tar xvfz simplesim-arm-0.2.tar.gz
```

Pour la Compilation et l'installation du simulateur SimpleScalar, exécutez :

```
cd simplesim-arm
make config-arm
make
```

Le répertoire simplesim-arm contient plusieurs répertoire aprés l'installation ,parmi ces répertoire `test-arm` qui contient plusieurs programmes qui sont des benchmarks de développent de simulateurs : `anagram, bresenham.1,bresenham.2 ,test-args ,bzip2 ,cc1.`

Quelque exemples de simulation d'exécution des programme de `tests-arm` avec simplescalar

```
simplesim$ sim-cache anagram words < input.txt (anagram).
simplesim$ sim-cache bresenham.1 100 (line drawing).
simplesim$ sim-cache bresenham.2 100 (another alg).
```

```
simplesim$ sim-cache bzip2 10 (compressor).
simplesim$ sim-cache cc1 -O 1stmt.i (GNU GCC) .
```

- Installation de compilateur croisé pour arm ,télécharger les outils suivants depuis http://www.simplescalar.com/ :

http://www.eecs.umich.edu/~taustin/code/arm-cross/gcc-2.95.2.tar.gz
http://www.eecs.umich.edu/~taustin/code/arm-cross/binutils-2.10.tar.gz
http://www.eecs.umich.edu/~taustin/code/arm-cross/glibc-2.1.3.tar.gz
http://www.eecs.umich.edu/~taustin/code/arm-cross/glibc-2.1.3.tar.gz

créez le répertoire "**cross**" sous votre répertoire et copiez touts ces outils dans ce répertoire :

```
$ cd simplescalar
$ mv binutils-2.10.tar.gz cross
$ mv glibc-2.1.3.tar.gz cross
$ mv gcc-2.95.2.tar.gz cross
$ cd cross
$ tar xvfz binutils-2.10.tar.gz
$ tar xvfz glibc-2.1.3.tar.gz
$ tar xvfz gcc-2.95.2.tar.gz
```

Soit le répertoire `/home/simplescalar/cross`

1.installation de binutils

```
$ cd binutils-2.10
$ ../configure --target=arm-linux --prefix=/home/simplescalar/cross
$ make
$ make install
cd ..
export PATH=$PATH:/home/simplescalar/cross/bin
```

2. install *gcc-2.95.2* : Compilation et installation du compilateur croisé GNU GCC, exécutez :

```
$ cd gcc-2.95.2
$ ../cofigure --prefix=/home/simplescalar/cross --target=arm-linux --with-as=
/home/simplescalar/cross/bin/aram-linux-as
$ make LANGUAGES=c
$ make LANGUAGES=c install
$ cd ..
```

`$ vi lib/gcc-lib/arm-linux/2.95.2/specs`

remplacer toutes les occurrences de *"elf32arm"* par *"armelf_linux"*, cela corrige une incompatibilité entre la version plus récent de CCG et glibc bibliothèques.

pour compiler sur simplescalar arm avec la commande suivants :

`arm-linux -gcc prog -static -c prog.c`

# B.l'outil gnuplot

Certains langages ne permettent pas directement de dessiner des courbes et de tracer des plots. Il faut pour cela stocker les résultats dans des fichiers, et, dans un deuxième temps utiliser un autre logiciel qui va lire ces fichiers et tracer les courbes correspondantes. Il existe de nombreux logiciels qui permettent de s'occuper de ces taches (grâce, xmgrace, gnuplot...).

Gnuplot se lance depuis un terminal avec la commande :

`gnuplot`

On peut lancer gnuplot en lui demandant d'exécuter un ensemble d'instructions écrite dans un fichier (par exemple `script.sh`) en utilisant la commande

`gnuplot script.sh`

- Options des graphes

Avant de tracer un graphe, il est possible de régler un certain nombre de de propriétés par défaut. On règle toutes les valeurs des paramètre en utilisant la commande set suivie du nom du paramètre et de sa valeur. On peut également revenir aux réglages initiaux grâce à unset. Voici une liste des propriétés les plus fréquentes que l'on peut être amené à régler :

`set title "Mon Titre"` spécifie la légende du graphe

`set notitle` supprime la légende (équivalent à unset key)

`set xlabel "titre x"` fixe le nom de l'axe x (idem pour y et z)

`set lmargin at screen 0.3` règle la marge à gauche à 30% de la taille totale du graphe.

La fonction **plot** admet beaucoup d'options donc certaines sont décrites ici :

`title "mon titre"` spécifie un titre pour la légende

`with dots` trace avec des points

`with points` trace avec des symbole (plus, croix...)

`with lines` trace avec des lignes

`with linespoints` trace avec des points et des lignes

`with impulses` trace avec des droites verticales

`with errorbars` trace avec des points et des barres d'erreur

`pointtype 3` utilise le symbole 3 (entre 0 et 13) quand points est specifié

`linecolor rgb "couleur"` utilise une couleur prédéfinie (show colornames liste des couleurs prédéfinies)

La commande de base pour représenter des données contenues dans un fichier est : `plot "fichier.dat"`

On peut spécifier une colonne à lire en particulier en rajoutant une option à la commande plot :

`Plot "fichier" using 1:3` trace la colonne 3 en fonction de la colonne 1

**Exemple :**

soit le fichier `LR.dat` qui contient les données de la température moyenne :

```
# mois température moyenne (°c) Ensoleillement Ch) - sources données : http ://www.infoclimat.fr
Jan 5,9 84
Fév 6,9 111
Mar 8,7 174
Avr 11,1 212
Mai 14,3 239
Jun 17,5 272
Jui 19,8 305
Aou 19,6 277
Sep 17,8 218
Oct 14,2 167
Nov 9,4 107
Déc 6,6 85
```

Le script **gnuplot3.gnu** permet de tracer une courbe selon les paramétres définit et les données de fichier **LR.dat** :

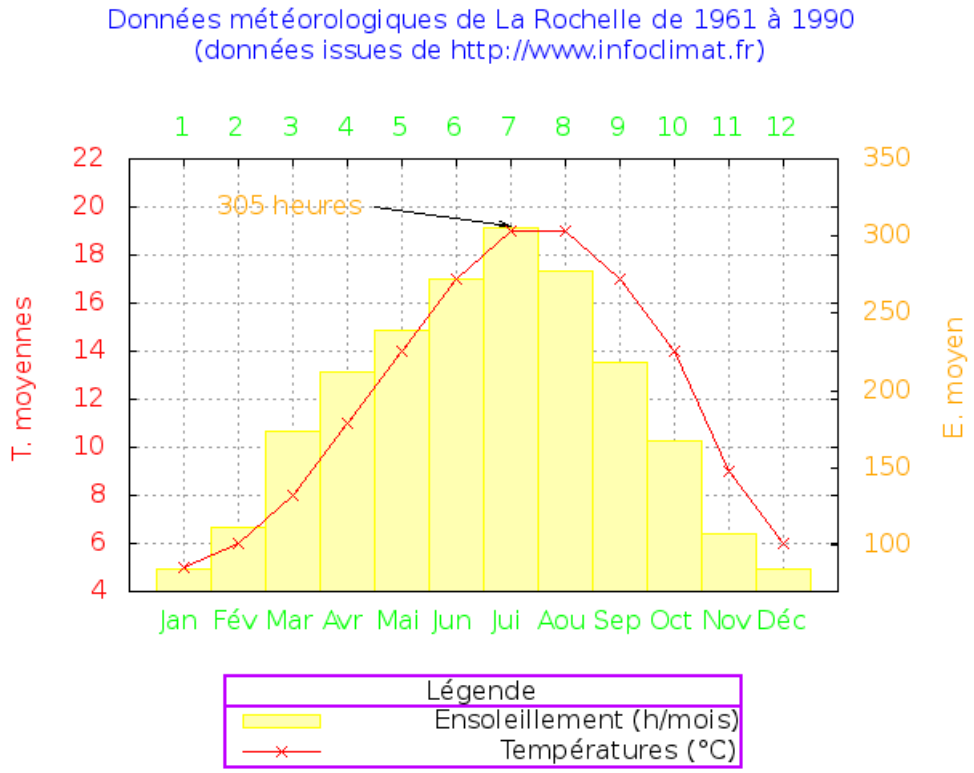
```
Reset
set terminal png
set output "La Rochelle.png"
set title textcolor rgb "blue" "Données météorologiques de La Rochelle de 1961 à 1990\n"
set key outside below; set key title "Légende"; set key box reverse; set key box lw 2
lc 4 # Légende
set grid # Grille de fond
set label 1 textcolor rgb "orange" "305 heures" at 2,20 centre; set arrow 1 from 3.5,20
to 6,19.2 # Info
set xtics textcolor rgb "green" border ("Jan" 0, "Fév" 1, "Mar" 2, "Avr" 3, "Mai" 4,"Jun"
5,"Jui" 6,
"Aou" 7,"Sep" 8,"Oct" 9,"Nov" 10,"Déc" 11) # Mois sur axe des x
set x2range [-1:12];
set x2tics textcolor rgb "green" border ("1" 0, "2" 1, "3" 2, "4" 3, "5" 4,"6" 5,"7"
6, "8" 7,"9" 8,"10" 9,
"11" 10,"12" 11) # Chiffres sur axe des x2
set ytics textcolor rgb "red"; set ylabel textcolor rgb "red" "T. moyennes" # Colori-
sation
```

```
    set y2label textcolor rgb "orange" "E. moyen"; set y2tics textcolor rgb "orange";
    set y2range [70:350]# Colorisation + échelle axe y2
    set style fill transparent solid 0.3# Colorisation plot [-1:12] [4:22] "LR.dat"
    using 3 axes x1y2 title "Ensoleillement (h/mois)"
    with boxes lc rgb "yellow",\ "LR.dat" using 2 title "Températures (°C)" with linespoints
lc rgb "red"
```

Pour tracer la corbe correspondants aux script `gnuplot3.gnu et LR.dat` en tapant les commandes

```
gnuplot
gnuplot> load "gnuplot3.gnu"
```

Et voilà la corbe correspondante :



# Bibliographie


[1] www.info.univ-angers.fr. Mémoires.

[2] https://fr.wikipedia.org

[3] imss-www.upmf-grenoble.fr /6-GestionMemoire-4dpp.

[4] http://sfavreau.chez-alice.fr. Mémoire

[5] https://www.openclassrooms.com

[6] http://www.simplescalar.com

[7] http://www.trimaran.org

[8] Alan Jai Smith "cache memories", ACM Computing surveys 14(3) :473-5330 Septembre 1982.

[9] Barry Wilkinson"Computer architecture-Design and Performance", Prentice Hall, 1996.

[10] Todd M.Austin "SimpleScalar Hacker's Guide to the SimpleScalar Architectural Research Tool Set". Intel MicroComputer Research labs, January,1997.

[11] Doug Borger, Todd M.Austin "The Simplescalar Tool Set, Version 2.0", 1997.

[12] Chunho Lee and Miodrag Potkonjak and William H. Mangione-Smith, "MediaBench : A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems". Pp 330-335, 1997.

[13] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst. MiBench : A free, commercially representative embedded benchmark suite. In IEEE 4th Annual Workshop on Workload Characterization, Austin, 2001, pp.1-12.

[14] Horatio Quadjovie,ă"Outils interactifs libres d'aide à la création scientifique", avril 2009, Département de mathématiques, (FDS/UL, Togo), BP. 1515 Lomé.

[15] Sébastien ROHAUT , "LINUX, Maîtrisez l'administration du système", [2ième édition] par Eni.

[16] Thèse de doctorat "Reconfiguration Dynamique des Mémoires Caches Dans un MP-SoC à Architecture Homogène" Présentée par Asmaa BENGUEDDACH, Université d'Oran, 2013/2014,

[17] Magistère en Informatique : "Amélioration de la performance des processeurs généralistes par les réduction du nombre d'instructions exécutées" présenter par M. Mohamed GOUDJIL, Université M'hamed Bougara Boumerdès, 2006-2007.