TP3 - Évaluation des Performances de Mémoires Caches

Architecture des Microprocesseurs - ES201

Votre Nom

February 15, 2026

Contents

1 Introduction

Ce travail pratique a pour objectif d'évaluer les performances de différentes configurations de mémoires caches (instructions et données) pour 4 algorithmes de multiplication de matrices. La multiplication de matrices est une opération critique dont les performances dépendent fortement de l'organisation de la hiérarchie mémoire.

1.1 Contexte

Le taux de défauts de cache (*miss rate*), défini comme le rapport entre le nombre d'accès qui se traduisent par un défaut de cache et le nombre total d'accès au cache, est le paramètre principal pour évaluer ces performances.

1.2 Outils utilisés

- **Simulateur :** gem5 version 23.0.0.1 (architecture RISC-V 64 bits)
- **Scripts :** Bash pour l'automatisation des simulations
- **Programmes testés :** 4 algorithmes de multiplication de matrices (128×128)
  - P1 : `normale.riscv` - Multiplication classique (i-j-k)
  - P2 : `pointer.riscv` - Accès via pointeurs
  - P3 : `tempo.riscv` - Cache blocking (tiling)
  - P4 : `unrol.riscv` - Loop unrolling

2 Question 1 : Paramètres gem5

2.1 Configurations de caches

Nous considérons deux configurations de caches décrites dans le Tableau 1.

Table 1: Configurations de caches pour 2 processeurs

Config	I-cache	D-cache	L2 cache	Block size
C1	4KB direct	4KB direct	32KB direct	32 bytes
C2	4KB direct	4KB 2-way	32KB 4-way	32 bytes

2.2 Paramètres gem5

Le Tableau 2 présente les paramètres gem5 correspondant à chaque configuration.

Table 2: Paramètres de cache pour chaque configuration (Tableau 8)

Configuration	IL1	DL1	UL2
C1	4kB, 1-way, LRU, 32B	4kB, 1-way, LRU, 32B	32kB, 1-way, LRU, 32B
C2	4kB, 1-way, LRU, 32B	4kB, 2-way, LRU, 32B	32kB, 4-way, LRU, 32B

2.3 Implémentation dans gem5

Le fichier de configuration `se_A7.py` implémente ces paramètres comme suit :

Listing 1: Extrait de se_A7.py

```
# Configuration de base (C1)
system.cache_line_size = 32
system.cpu.icache = L1ICache()   # 4kB, assoc=1
system.cpu.dcache = L1DCache()   # 4kB, assoc=1
system.l2cache = L2Cache()       # 32kB, assoc=1

# Passage en configuration C2
if args.cacheconfig == "C2":
    system.cpu.dcache.assoc = 2   # DL1: 2-way
    system.l2cache.assoc = 4      # UL2: 4-way
```

Notes importantes :

- Le cache d'instructions (IL1) reste direct-mapped dans les deux configurations
- L'algorithme de remplacement LRU est utilisé par défaut dans gem5
- La taille de bloc (block size) est uniforme : 32 bytes

3 Question 2 : Méthodologie et Résultats

3.1 Méthodologie de simulation

3.1.1 Script de lancement des simulations

Pour automatiser les 8 simulations (4 programmes $\times$ 2 configurations), nous avons développé le script `simulations.sh` :

Listing 2: simulations.sh - Script de lancement

```
#!/bin/bash
# Simule les 8 cas : 4 programmes    2 configs

PROGS=("normale.riscv" "pointer.riscv" "tempo.riscv" "unrol.riscv")
NAMES=("normale" "pointer" "tempo" "unrol")
CONFIGS=("C1" "C2")
GEM5="/root/gem5/build/RISCV/gem5.opt"

```

```
echo "Lancement des 8 simulations..."
start=$(date +%s)

for i in "${!PROGS[@]}"; do
    for cfg in "${CONFIGS[@]}"; do
        out="m5out_${NAMES[$i]}_${cfg}"
        echo "-> ${NAMES[$i]} - $cfg"
        rm -rf "$out"
        $GEM5 --outdir="$out" --redirect-stdout --redirect-stderr \
              se_A7.py --cmd "exo3/${PROGS[$i]}" --cacheconfig "
                 $cfg"
        [ -f "$out/stats.txt" ] && echo "  OK" || echo "  ERREUR"
    done
done

echo "Termine en $(($(date +%s)-start))s"
```

Ce script génère 8 dossiers de sortie : `m5out_normale_C1`, `m5out_normale_C2`, etc., chacun contenant un fichier `stats.txt` avec les statistiques de simulation.

3.1.2 Script d'extraction des résultats

Le script `resultats.sh` extrait automatiquement les miss rates des fichiers `stats.txt` :

Listing 3: resultats.sh - Extraction des miss rates

```
#!/bin/bash
get_stat() {
    grep "^$2" "$1/stats.txt" 2>/dev/null | awk '{print $2}'
}
percent() {
    awk -v m=$1 -v a=$2 'BEGIN {
        if(a>0) printf "%.2f",m/a*100; else print "0.00"
    }'
}

# Exemple pour I-cache
for p in "${PROGS[@]}"; do
    for c in "${CONFIGS[@]}"; do
        d="m5out_${p}_${c}"
        m=$(get_stat "$d" "system.cpu.icache.overallMisses::total")
        a=$(get_stat "$d" "system.cpu.icache.overallAccesses::total
           ")
        printf "%.2f%%" "$(percent ${m:-0} ${a:-1})"
    done
done
```

Statistiques gem5 utilisées :

- I-cache : `system.cpu.icache.overallMisses::total` et `overallAccesses::total`
- D-cache : `system.cpu.dcache.overallMisses::total` et `overallAccesses::total`
- L2 cache : `system.l2cache.overallMisses::total` et `overallAccesses::total`

3.2 Résultats obtenus

La Figure 1 présente les résultats complets des simulations.

```
root@6ef37a6d7cb4:/workspace/TP2/TP4# ./resultats.sh
TABLEAU 9 - Instruction Cache Miss Rate (%)
Programme          C1          C2
normale         2.16%       2.16%
pointer         0.07%       0.07%
tempo           2.33%       2.32%
unrol           0.17%       0.17%

TABLEAU 10 - Data Cache Miss Rate (%)
Programme          C1          C2
normale        51.43%      44.65%
pointer        51.47%      45.83%
tempo          50.99%      44.18%
unrol          51.53%      43.98%

TABLEAU 11 - L2 Cache Miss Rate (%)
Programme          C1          C2
normale        51.51%      45.66%
pointer        50.99%      44.59%
tempo          51.55%      45.92%
unrol          51.51%      46.16%
root@6ef37a6d7cb4:/workspace/TP2/TP4#
```

Figure 1: Résultats des simulations - Miss rates pour les 3 niveaux de cache

3.2.1 Tableau 9 - Instruction Cache Miss Rate

Table 3: Instruction Cache (IL1) Miss Rate (%)

Programme	**C1 (%)**	**C2 (%)**	**C1→C2**
P1 (normale)	2.16	2.16	0.00%
P2 (pointer)	0.07	0.07	0.00%
P3 (tempo)	2.33	2.32	-0.01%
P4 (unrol)	0.17	0.17	0.00%

3.2.2 Tableau 10 - Data Cache Miss Rate

Table 4: Data Cache (DL1) Miss Rate (%)

Programme	C1 (%)	C2 (%)	C1→C2
P1 (normale)	51.43	44.65	-6.78%
P2 (pointer)	51.47	45.83	-5.64%
P3 (tempo)	50.99	44.18	-6.81%
P4 (unrol)	51.53	43.98	-7.55%

3.2.3 Tableau 11 - Unified L2 Cache Miss Rate

Table 5: Unified L2 Cache (UL2) Miss Rate (%)

Programme	C1 (%)	C2 (%)	C1→C2
P1 (normale)	51.51	45.66	-5.85%
P2 (pointer)	50.99	44.59	-6.40%
P3 (tempo)	51.55	45.92	-5.63%
P4 (unrol)	51.51	46.16	-5.35%

4 Analyse et Interprétations

4.1 Analyse du cache d'instructions (Tableau 9)

4.1.1 Observations principales

- **Miss rates très faibles :** Tous les programmes affichent des taux de défaut inférieurs à 2.5%
- **Aucune amélioration C1→C2 :** Le cache d'instructions reste direct-mapped dans les deux configurations
- **Variation entre programmes :**
  - `pointer` : 0.07% (meilleur) - 1364 misses pour 2,095,967 accès
  - `unrol` : 0.17% - Code déroulé mais très réutilisé
  - `normale` : 2.16% - Structure de boucle classique
  - `tempo` : 2.33% (moins bon) - Code de blocking plus complexe

4.1.2 Interprétation

Ces résultats exceptionnels s'expliquent par :

1. **Localité temporelle exceptionnelle :** Les boucles de multiplication sont exécutées $N^3 \approx 2$ millions de fois ($N = 128$). Le même petit corps de boucle (50-200 bytes) est réutilisé massivement.

2. **Taille du code $\ll$ Taille du cache :** Le code actif (500 bytes à 1 kB) tient entièrement dans le cache d'instructions de 4 kB. Aucun *capacity miss*.

3. **Localité spatiale efficace :** Avec un block size de 32 bytes, chaque chargement apporte plusieurs instructions consécutives (prefetching naturel).

4.2 Analyse du cache de données (Tableau 10)

4.2.1 Observations principales

- **Miss rates très élevés :** Entre 44% et 52%, indiquant un *thrashing* sévère
- **Amélioration significative C1→C2 :** De 5.64% à 7.55% de réduction
- **Meilleur gain :** `unrol` avec -7.55%
- **Performances similaires :** Tous les algorithmes sont limités par la capacité

4.2.2 Interprétation

Les miss rates élevés s'expliquent par un problème de **capacité** :

- **Taille des données :** 3 matrices de 128×128 doubles $= 3 \times 128 \times 128 \times 8 = 384$ kB
- **Taille du cache :** DL1 = 4 kB seulement
- **Ratio :** $384/4 = 96$: les données dépassent le cache d'un facteur 96

L'amélioration C1→C2 provient de :

- **Réduction des conflict misses :** La 2-way associativité du DL1 réduit les collisions
- **Meilleure répartition :** Les accès mémoire se répartissent mieux dans les ensembles

Résultat surprenant : `tempo` (cache blocking) n'obtient **pas** de gain majeur. Le blocking est optimisé pour des caches plus grands (>32 kB). Avec un cache de 256 kB, on s'attendrait à une réduction du miss rate à 20-30%.

4.3 Analyse du cache L2 (Tableau 11)

4.3.1 Observations principales

- **Miss rates très élevés :** Entre 44% et 52%, similaires au DL1
- **Amélioration modérée C1→C2 :** De 5.35% à 6.40%
- **Performances homogènes :** Peu de différence entre les 4 algorithmes

4.3.2 Interprétation

Le L2 de 32 kB souffre du même problème de capacité :

- **Ratio :** $384/32 = 12$: les données dépassent le L2 d'un facteur 12
- **Thrashing :** La majorité des accès L2 vont en RAM
- **4-way associativity :** Aide mais ne résout pas le problème fondamental

Conclusion : Le goulot d'étranglement est la **capacité**, pas l'organisation des caches.

5 Question 3 : Localité de références pour le code

5.1 Question posée

Les 4 algorithmes de multiplication de matrices présentent-ils une bonne localité de références pour le code ? Pourquoi ?

5.2 Réponse : OUI, excellente localité

Les 4 algorithmes présentent une **excellente localité de références** pour le code (instructions), comme en témoignent les miss rates très faibles du cache d'instructions (0.07% à 2.33%).

5.3 Justifications

5.3.1 1. Contraste instructions vs données

Le Tableau 6 illustre le contraste frappant entre les performances du cache d'instructions et du cache de données.

Table 6: Contraste entre localité des instructions et des données

Programme	I-cache (%)	D-cache (%)	Facteur
normale	2.16	51.43	$\times 24$
pointer	0.07	51.47	$\times 735$
tempo	2.33	50.99	$\times 22$
unrol	0.17	51.53	$\times 303$

Le cache d'instructions est **22 à 735 fois plus efficace** que le cache de données.

5.3.2 2. Localité temporelle exceptionnelle

- **Réutilisation massive :** Pour `pointer`, seulement 1364 misses pour 2,095,967 accès instructions
- **Calcul :** Le code est chargé une fois puis réutilisé ≈ 1500 fois !
- **Boucles imbriquées :** Exécution $N^3 = 128^3 \approx 2.1$ millions d'itérations

5.3.3 3. Taille du code négligeable

Corps de boucle :	50-200 bytes
Code total actif :	∼500 bytes - 1 kB
Cache IL1 :	4 kB
Ratio :	Code / Cache ≈ 1/4 à 1/8

Figure 2: Comparaison taille code vs cache

Le code tient **entièrement** dans le cache. Aucun *capacity miss*, seulement des *compulsory misses* initiaux.

5.3.4 4. Inefficacité de l'associativité pour IL1

- **Observation :** C1 = C2 pour tous les programmes (Tableau 9)
- **Explication :** IL1 reste direct-mapped dans les deux configurations
- **Conclusion :** Avec un code aussi petit, il n'y a **aucun conflict miss**
- **Implication :** Augmenter l'associativité de IL1 serait inutile (déjà optimal)

5.4 Asymétrie fondamentale

La multiplication de matrices exhibe une **localité asymétrique** :

Instructions		**Données**
Excellente	vs	Médiocre
0.07% - 2.33%		44% - 52%
Petit code réutilisé N^3 fois		Gros volume » capacité

Cette asymétrie est typique des algorithmes de **calcul scientifique** :

- Un petit noyau de calcul (quelques dizaines d'instructions)
- Appliqué massivement sur un gros volume de données (centaines de kB)

5.5 Conséquence pour l'optimisation

Le facteur limitant des performances est la **hiérarchie mémoire pour les données**, pas le code. C'est pourquoi les optimisations comme :

- `tempo` (cache blocking) : Améliore la localité des **données**
- `unrol` (loop unrolling) : Réduit les instructions de contrôle mais cible les **données**

se concentrent sur l'amélioration de la localité des accès mémoire aux **données**, pas aux instructions.

6 Recommandations

6.1 Configuration matérielle optimale

Pour améliorer significativement les performances, il faudrait :

1. **Augmenter le L2 à 512 kB - 1 MB**
   - Permettrait de contenir les 3 matrices (384 kB)
   - Réduction du miss rate de 50% à $<10\%$
2. **Ajouter un cache L3 de 2-4 MB**
   - Marge pour working sets plus grands
   - Réduit les accès à la RAM
3. **Augmenter l'associativité du L2 à 8-way ou 16-way**
   - Réduit davantage les conflict misses
   - Synergie avec capacité augmentée
4. **Augmenter le DL1 à 32 kB minimum**
   - Permettrait au blocking de `tempo` d'être efficace
   - Réduction significative pour les tuiles 32×32

6.2 Optimisations algorithmiques

Avec la configuration actuelle (caches petits), tous les algorithmes sont équivalents. Pour des caches plus grands :

- **Cache blocking** (`tempo`) : Tuiles de 32×32 ou 64×64 adaptées au L1
- **Combinaison blocking + unrolling** : Gain maximal
- **Éviter pointer** : Indirections nuisent à la localité

7 Conclusion

Ce TP a permis d'évaluer quantitativement l'impact de l'organisation des caches sur les performances de la multiplication de matrices. Les résultats mettent en évidence :

1. **Excellente localité du code** : Miss rates $<2.5\%$ pour les instructions
2. **Mauvaise localité des données** : Miss rates $>44\%$ dus à un problème de capacité
3. **Gain modéré de l'associativité** : 5-7% d'amélioration C1→C2, mais insuffisant
4. **Nécessité de caches plus grands** : Le facteur limitant est la capacité, pas l'organisation

L'analyse démontre que pour des applications à forte intensité de calcul sur de grandes structures de données, la **capacité des caches** est le paramètre critique, bien plus que leur associativité ou leur latence.

Les outils développés (scripts d'automatisation) permettent d'étendre cette étude à d'autres configurations de caches ou d'autres algorithmes, facilitant l'exploration de l'espace de design des architectures de processeurs.

Annexes

A. Code source complet

Les scripts développés sont disponibles sur GitHub :

https://github.com/votre-username/tp3-caches

B. Fichiers générés

- `simulations.sh` : Lancement automatique des 8 simulations
- `resultats.sh` : Extraction et affichage des miss rates
- `se_A7.py` : Configuration gem5 pour les deux architectures
- 8 dossiers `m5out_*` : Résultats complets des simulations

C. Statistiques détaillées

Pour consultation complète, les fichiers `stats.txt` de chaque simulation contiennent plus de 1400 lignes de statistiques détaillées incluant :

- Nombre total d'instructions exécutées
- Temps de simulation en cycles
- Statistiques de latence pour chaque niveau de cache
- Distribution des types d'accès (lecture/écriture)
- Statistiques MSHR (Miss Status Handling Registers)