TP3 - Évaluation des Performances de Mémoires Caches

Architecture des Microprocesseurs - ES201

ELADEB Mohamed, SAFI Ahmed, KLILA Mohamed, ABID Mahdi, BOUGUERBA Nazih

15 février 2026

1 Introduction

Ce TP évalue les performances de 2 configurations de mémoires caches (C1 : direct-mapped, C2 : set-associative) pour 4 algorithmes de multiplication de matrices (128×128). Le paramètre analysé est le **miss rate** : rapport entre défauts de cache et accès totaux.

Outils : gem5 v23.0.0.1 (RISC-V 64 bits), scripts Bash pour automatisation.

Programmes testés :

— P1 (`normale`) : Multiplication classique (i-j-k)
— P2 (`pointer`) : Accès via pointeurs
— P3 (`tempo`) : Cache blocking
— P4 (`unrol`) : Loop unrolling

2 Question 1 : Paramètres gem5

2.1 Configurations de caches

Table 1 – Configurations de caches testées

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

2.2 Tableau 8 - Paramètres gem5

Table 2 – Paramètres de cache pour chaque configuration

Config	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

Implémentation (se_A7.py) :

```
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

if args.cacheconfig == "C2":
    system.cpu.dcache.assoc = 2  # 2-way
    system.l2cache.assoc = 4     # 4-way
```

3 Question 2 : Méthodologie et Résultats

3.1 Méthodologie

Script simulations.sh (lance les 8 simulations) :

```
#!/bin/bash
PROGS=("normale.riscv" "pointer.riscv" "tempo.riscv" "unrol.riscv")
CONFIGS=("C1" "C2")
GEM5="/root/gem5/build/RISCV/gem5.opt"

for prog in "${PROGS[@]}"; do
    for cfg in "${CONFIGS[@]}"; do
        out="m5out_${prog}_${cfg}"
        $GEM5 --outdir="$out" se_A7.py --cmd "$prog" --cacheconfig "$cfg"
    done
done
```

Script resultats.sh (extrait les miss rates) :

```
get_stat() { grep "^$2" "$1/stats.txt" | awk '{print $2}' }
percent() { awk -v m=$1 -v a=$2 'BEGIN{printf "%.2f",m/a*100}' }

# Exemple I-cache
m=$(get_stat "m5out_prog_C1" "system.cpu.icache.overallMisses::total")
a=$(get_stat "m5out_prog_C1" "system.cpu.icache.overallAccesses::total")
echo "Miss rate: $(percent $m $a)%"
```

3.2 Résultats

```
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 IL1, DL1 et UL2

3.2.1 Tableau 9 - Instruction Cache (IL1)

TABLE 3 – Instruction Cache Miss Rate (%)

Programme	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 (DL1)

TABLE 4 – Data Cache Miss Rate (%)

Programme	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 - L2 Cache (UL2)

TABLE 5 – L2 Cache Miss Rate (%)

Programme	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 des Résultats

4.1 Cache d'instructions (Tableau 9)

Observations :

— Miss rates très faibles : 0.07% à 2.33%

— Aucun gain C1→C2 (IL1 reste direct-mapped)

— `pointer` meilleur (0.07%) : code réutilisé 1500× (1364 misses / 2M accès)

Explication : Le code (∼500B-1kB) tient entièrement dans IL1 (4kB). Localité temporelle exceptionnelle : boucles exécutées $N^3 \approx 2M$ fois.

4.2 Cache de données (Tableau 10)

Observations :

— Miss rates élevés : 44%-52% (thrashing)

— Gain C1→C2 : 5.6% à 7.5% (`unrol` meilleur)

— Performances similaires entre algorithmes

Explication : Problème de capacité. 3 matrices = $3 \times 128 \times 128 \times 8 = 384$ kB >> DL1 (4 kB). Ratio 96 :1. La 2-way associativité réduit les conflict misses mais ne résout pas le problème fondamental.

4.3 L2 Cache (Tableau 11)

Observations :
— Miss rates élevés : 44%-52% (similaire au DL1)
— Gain C1→C2 modéré : 5.3% à 6.4%
Explication : Même problème de capacité. 384 kB >> L2 (32 kB), ratio 12 :1. La majorité des accès L2 vont en RAM.

5 Question 3 : Localité du Code

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

5.1 Réponse : OUI, excellente localité

Justifications :

1. **Contraste instructions vs données** (Tableau 6) :

Table 6 – Contraste I-cache vs D-cache

Programme	I-cache	D-cache	Facteur
normale	2.16%	51.43%	×24
pointer	0.07%	51.47%	×735
tempo	2.33%	50.99%	×22
unrol	0.17%	51.53%	×303

Le cache d'instructions est **22 à 735 fois plus efficace**.

2. **Localité temporelle :** Boucles exécutées $N^3 = 2.1M$ fois. Pour `pointer` : code chargé une fois, réutilisé 1500 fois.
3. **Taille code ≪ cache :** Code actif (500B-1kB) << IL1 (4kB). Aucun capacity miss.
4. **Pas de gain C1→C2 :** Code trop petit, aucun conflict miss. L'associativité est inutile (déjà optimal).

5.2 Conclusion : Localité asymétrique

Instructions		Données
Excellente (0.07-2.33%)	vs	Médiocre (44-52%)

Typique du calcul scientifique : petit noyau de code réutilisé massivement sur gros volume de données. Le bottleneck est la **mémoire données**, pas le code.

6 Recommandations

Pour améliorer les performances :
— **Augmenter L2 à 512kB-1MB** (contenir les 3 matrices)
— **Augmenter DL1 à 32kB** (optimiser cache blocking)
— **Ajouter L3 de 2-4MB**
— **L2 8-16-way associative**
Avec ces configs, `tempo` (blocking) devrait montrer son avantage (miss rate réduit à 20-30%).

7 Conclusion

Ce TP démontre que :

1. Le code a une **excellente localité** (miss rate <2.5%)
2. Les données ont une **mauvaise localité** (miss rate >44%)
3. Le facteur limitant est la **capacité**, pas l'organisation
4. L'associativité apporte un gain modéré (5-7%) mais insuffisant

Pour les applications de calcul intensif sur grandes données, la **capacité des caches** est critique.

Annexes

Code source : https://github.com/medm3alem/TP3
Fichiers :
— `simulations.sh` : Lancement des 8 simulations
— `resultats.sh` : Extraction des miss rates
— `se_A7.py` : Configuration gem5
— 8 dossiers `m5out_*` : Statistiques complètes (1400+ lignes/fichier)