

Mémoires caches - Evaluation des performances de différentes configurations de mémoires caches

1st Luiz Gariglio Dos Santos
Ingénieur Degree Programme in STIC
ENSTA Paris
Paris, France
email@ensta-paris.fr

2nd Helena Guachalla De Andrade
Ingénieur Degree Programme in STIC
ENSTA Paris
Paris, France
email@ensta-paris.fr

3rd Santiago Florido Gomez
Ingénieur Degree Programme in STIC
ENSTA Paris
Paris, France
santiago.florido@ensta-paris.fr

4th Franck Ulrich Kenack Noumedem
Ingénieur Degree Programme in STIC
ENSTA Paris
Paris, France
email@ensta-paris.fr

Abstract—mamamamsmamammasaalsnoihrf eirfrf
Index Terms—hsbuaahbdwefwcwcbdwuguwcc

I. INTRODUCTION

ahdubwyfwi ceuce hyec

II. EVALUATION DES PERFORMANCES DE DIFFÉRENTES CONFIGURATIONS DE MÉMOIRES CACHES POUR 4 ALGORITHMES DE MULTIPLICATION DE MATRICES

A. Paramètres de configuration des caches utilisés dans gem5

Configuration	Instruction cache	Data cache	L2 cache	Block size
C1	4KB direct-mapped	4KB direct-mapped	32KB direct-mapped	32 bytes
C2	4KB direct-mapped	4KB 2-way set-asso	32KB 4-way set-asso	32 bytes

TABLE I

Les paramètres de configuration des caches dans le simulateur gem5 suivent le format suivant : tag:n_lignes:taille_bloc:associativité:politique.

Pour chaque cache, le nombre de lignes est obtenu en divisant la taille totale du cache par la taille d'un bloc. L'associativité correspond au nombre de voies, fixé à 1 pour un cache direct-mapped, 2 pour un cache 2-way set-associative, etc. Enfin, la politique de remplacement utilisée ici est LRU, notée 1.

Ainsi, pour la configuration C1, par exemple, le cache d'instructions possède $\frac{4096}{32} = 128$ lignes, d'où i11:128:32:1:1. En utilisant la même logique pour les autres caches et configurations, le tableau suivant est rempli.

Configuration	IL1	DL1	UL2
C1	i11:128:32:1:1	d11:128:32:1:1	ul2:1024:32:1:1
C2	i11:128:32:1:1	d11:64:32:2:1	ul2:256:32:4:1

B. Taux de défauts dans les différentes caches

- Le taux de défauts dans le cache d'instructions i11 : icode.overallMissRate
- Le taux de défauts dans le cache de données d11 : dcache.overallMissRate
- Le taux de défauts dans le cache unifié (L2) ul2 : l2cache.overallMissRate

Programmes	Configuration de caches	
	C1	C2
P1 normale	0,00%	0,00%
P2 (pointeur)	0,00%	0,00%
P3 (tempo)	0,00%	0,00%
P4 (unrol)	0,00%	0,00%

TABLE II: icode.overallMissRate

Programmes	Configuration de caches	
	C1	C2
P1 normale	30,08%	31,01%
P2 (pointeur)	30,22%	31,12%
P3 (tempo)	30,23%	31,12%
P4 (unrol)	30,06%	31,10%

TABLE III: dcache.overallMissRate

Programmes	Configuration de caches	
	C1	C2
P1 normale	43,85%	42,20%
P2 (pointeur)	43,63%	42,21%
P3 (tempo)	43,62%	42,20%
P4 (unrol)	43,65%	42,02%

TABLE IV: l2cache.overallMissRate

Nous pouvons vérifier que les quatre algorithmes présentent une bonne localité de référence pour le code. En effet, le taux de défaut du cache d'instructions est nul ($\approx 0,00\%$) pour tous les programmes et pour les deux configurations de caches (Table II). Cela signifie que le flux d'instructions tient entièrement dans le cache d'instructions et que les boucles de

multiplication de matrices réutilisent toujours le même petit bloc de code, ce qui exploite très bien la localité spatiale et temporelle du code.

III. PROFILING DE L'APPLICATION

Pour procéder à l'évaluation des configurations de cache pour chacun des algorithmes proposés et analyser leurs performances, il est proposé de réaliser un *profiling* de l'application à l'aide du simulateur gem5. Le *profiling* est essentiel, car il permet de quantifier des éléments du comportement du programme afin de prendre des décisions d'optimisation et de microconception architecturale fondées sur des données, principalement issues de la simulation [1]. Il permet également d'identifier des *hotspots* sur lesquels concentrer la conception et l'optimisation, c'est-à-dire de cibler en priorité les composantes qui contribuent le plus au temps d'exécution. Enfin, il fournit une première approximation de la caractérisation de la charge de travail (*workload*) d'un programme, ce qui facilite l'orientation des choix de conception.

Classe	Dijkstra large (A7)	Dijkstra large (A15)
Lecture (Load)	45 516 963 [28.4]	45 905 506 [28.5]
Écriture (Store)	19 439 553 [12.1]	19 593 718 [12.1]
Branchement	43 904 570 [21.5]	44 122 872 [21.5]
Calcul entier (Int)	95 334 242 [59.5]	95 780 142 [59.4]
Calcul flottant (Fp)	0 [0.0]	0 [0.0]
Total d'instructions exécutées	204 195 328	205 402 238

TABLE V: Dijkstra large (Cortex-A7 vs Cortex-A15).

Classe	Dijkstra small (A7)	Dijkstra small (A15)
Lecture (Load)	10 313 882 [28.5]	10 474 419 [28.5]
Écriture (Store)	4 759 916 [13.2]	4 850 175 [13.2]
Branchement	9 823 729 [21.4]	9 978 854 [21.4]
Calcul entier (Int)	21 106 947 [58.3]	21 363 899 [58.2]
Calcul flottant (Fp)	0 [0.0]	0 [0.0]
Total d'instructions exécutées	46 004 474	46 667 347

TABLE VI: Dijkstra small (Cortex-A7 vs Cortex-A15).

Classe	Blowfish (A7)	Blowfish (A15)
Lecture (Load)	19 141 [21.8]	19 769 [21.3]
Écriture (Store)	5 516 [6.3]	5 671 [6.1]
Branchement	29 760 [25.3]	30 060 [24.4]
Calcul entier (Int)	63 342 [72.0]	67 560 [72.6]
Calcul flottant (Fp)	0 [0.0]	0 [0.0]
Total d'instructions exécutées	117 759	123 060

TABLE VII: Blowfish (Cortex-A7 vs Cortex-A15).

Le *profiling* indique que la classe dominante dans les deux programmes est le calcul entier ($\approx 60\%$ pour Dijkstra et $\approx 72\%$ pour Blowfish) ; ainsi, une micro-architecture disposant de davantage d'ALU entières ou d'un parallélisme accru sur l'entier peut apporter le gain le plus important. Les branchements représentent également une part notable : restructurer les boucles ou réduire les sauts peut donc limiter les pénalités de contrôle. Côté mémoire, Dijkstra est plus sensible car il combine de nombreux accès, souvent irréguliers ; des améliorations de cache et de latence ont donc un impact plus marqué. Blowfish présente généralement des accès plus

séquentiels et une meilleure localité : la hiérarchie mémoire aide, mais le goulot principal demeure le *throughput* du calcul entier.

La comparaison montre que, tant pour Dijkstra/SSCA2-BCS que pour le produit de polynômes et Blowfish, le profil d'exécution est fortement dominé par les instructions de calcul : dans Dijkstra et Blowfish, le calcul entier prédomine, tandis que, pour un produit de polynômes implémenté en flottant, on observe une fraction élevée de calcul FP. Par conséquent, une amélioration architecturale transversale consiste à accroître le parallélisme d'exécution dans la classe dominante (davantage d'ALU entières pour les applications entières et davantage d'unités FP si le noyau est flottant), car cette approche cible directement la plus grande proportion d'instructions.

La principale divergence concerne le comportement mémoire. Dijkstra/SSCA2-BCS (graphes) exécute généralement de nombreux *loads/stores* et, surtout, avec des accès irréguliers (dépendants de structures chaînées ou de motifs non linéaires). Cela dégrade la localité, augmente les *miss* en cache et rend les performances très sensibles aux modifications de la hiérarchie mémoire (tailles/associativité des caches, latences, politiques, etc.). À l'inverse, le produit de polynômes et, dans une large mesure, Blowfish présentent des accès plus séquentiels et localisés, ce qui favorise les caches et réduit la pénalité mémoire ; dans ces cas, le goulot d'étranglement se déplace davantage vers le *throughput* de calcul.

Enfin, même si leur poids relatif varie, toutes ces applications peuvent bénéficier d'améliorations du contrôle de flux (branchements) : réduire les sauts via la restructuration des boucles et la simplification des conditions, ou améliorer la prédiction de branchement, diminue les pénalités de mauvaise prédiction et contribue à abaisser le CPI. En synthèse, le meilleur scénario combine (1) davantage de capacité de calcul dans la classe dominante, (2) une optimisation mémoire particulièrement critique pour les graphes, et (3) des améliorations de prédiction/codage apportant des gains additionnels de manière générale.

A. Impact de la taille des caches L1

Les figures suivantes présentent l'évolution des performances en fonction de la taille du cache L1. Pour Dijkstra, les deux jeux de données (small/large) sont affichés dans le même graphique. Les résultats sont regroupés par catégorie : performance générale, hiérarchie mémoire et prédiction de branchement.

a) *Performance générale (CPI, IPC, cycles).*:

b) *Hiérarchie mémoire (taux de défauts).*:

c) *Prédiction de branchement.*:

d) *Analyse synthétique.*: Quand la taille du L1 augmente, les taux de défauts I/D diminuent nettement, ce qui réduit le CPI et augmente l'IPC (amélioration plus marquée sur Dijkstra, plus sensible à la mémoire). Les métriques de prédiction de branchement restent globalement stables, ce qui est attendu car le prédicteur ne change pas entre configurations. Pour le Cortex-A7, la meilleure performance observée est obtenue

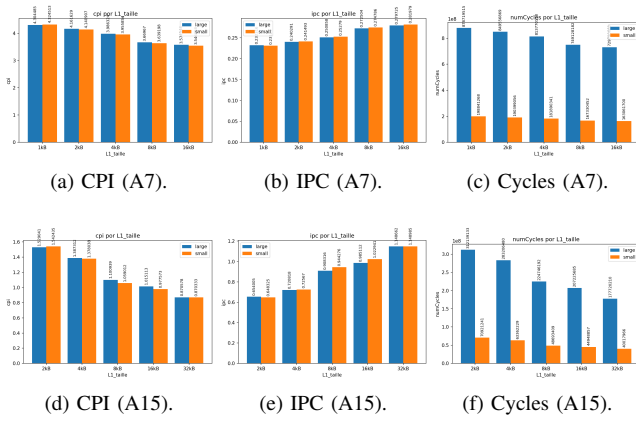


Fig. 1: Dijkstra : performance générale en fonction de la taille du cache L1 (A7 en haut, A15 en bas).

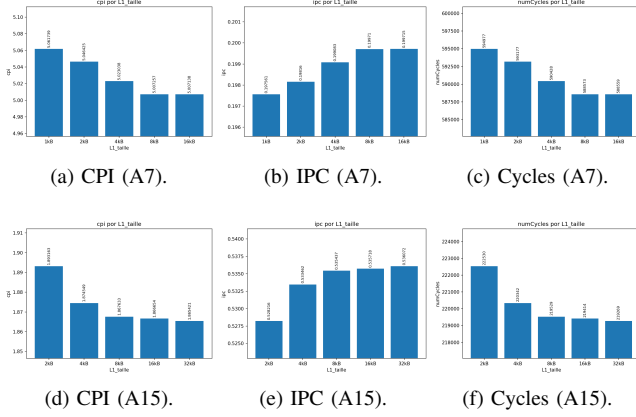


Fig. 2: Blowfish : performance générale en fonction de la taille du cache L1 (A7 en haut, A15 en bas).

avec **L1 = 16 kB** pour Dijkstra (small et large) et pour Blowfish (CPI minimal et IPC maximal, avec un gain qui se stabilise entre 8 kB et 16 kB).

REFERENCES

- [1] M. J. P. (University of York), “Profiling,” *Lecture Notes (4th Year HPC), University of York*. [Online]. Available: https://www-users.york.ac.uk/~mijp1/teaching/4th_year_HPC/lecture_notes/Profiling.pdf. Accessed: Feb. 9, 2026.

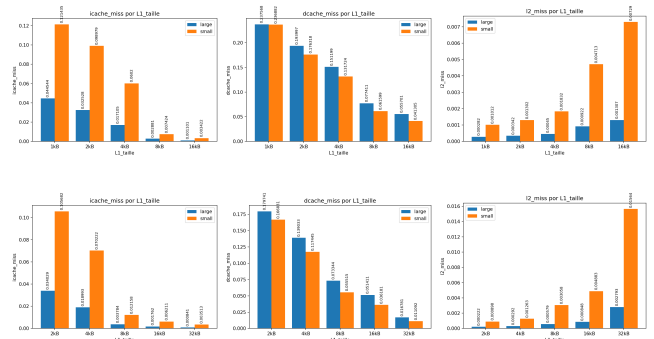


Fig. 3: Dijkstra : taux de défauts I-Cache, D-Cache et L2 en fonction de la taille du cache L1 (A7 en haut, A15 en bas).

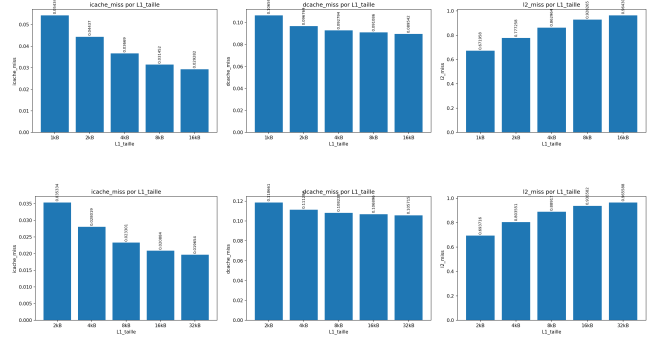


Fig. 4: Blowfish : taux de défauts I-Cache, D-Cache et L2 en fonction de la taille du cache L1 (A7 en haut, A15 en bas).

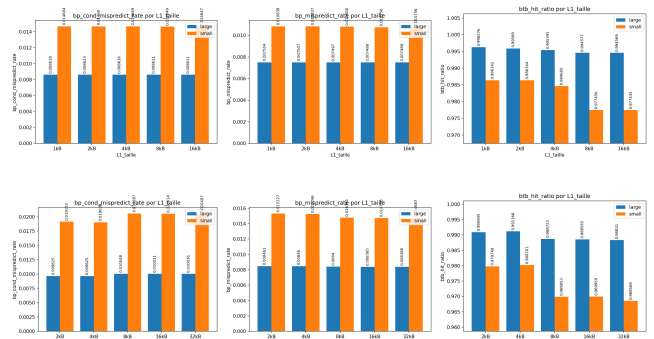


Fig. 5: Dijkstra : métriques de prédiction de branchement en fonction de la taille du cache L1 (A7 en haut, A15 en bas).

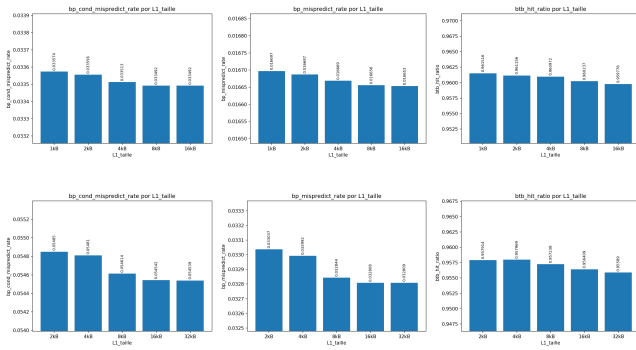


Fig. 6: Blowfish : métriques de prédiction de branchement en fonction de la taille du cache L1 (A7 en haut, A15 en bas).