Architecture des microprocesseurs (ECE_4ES01_TA)

TD/TP4 : Analyse de configurations d'architectures

Microprocesseur superscalaire / mémoires caches

Auteurs

Zhe CHEN

José Daniel CHACÓN GÓMEZ

Gustavo de MACENA BARRETOS

Lucas Emanuel BARBOSA VASCONCELOS

Encadrant / Chargé de TD : SIDEM Antoine

Responsable de la matière : HAMMAMI Omar

École Nationale des Techniques Avancées (ENSTA Paris)

Février 2026

Table des matières

1 Introduction

Dans ce TP, nous étudions l'impact de choix micro-architecturaux et de la hiérarchie mémoire sur les performances, la surface et l'énergie, en nous appuyant sur des simulations (gem5) et sur les applications `Dijkstra` et `BlowFish`.

2 Exercice 3 — A completer

2.1 Questions

2.1.1 Q1 - Configuration de cache

> **Énoncé (Q1).** Déterminez, pour chacune des deux configurations de mémoires cache, les paramètres de configuration des caches à utiliser dans gem5 et complétez la Table 1.

Dans cette question, nous allons comparer les performances de deux configurations de cache différentes, C1 et C2. Les caractéristiques de chaque configuration sont présentées dans le tableau ci-dessous.

Listing 1 – Extrait du fichier `se_cache.py`

```
def apply_cache_conf(args, system):
    """
    Cree I$, D$, L2 selon C1/C2 ou selon parametres custom.
    """
    system.cache_line_size = args.line_size

    # Selection config
    if args.conf == "C1":
        l1i_size, l1i_assoc = "4kB", 1
        l1d_size, l1d_assoc = "4kB", 1
        l2_size,  l2_assoc  = "32kB", 1
    elif args.conf == "C2":
        l1i_size, l1i_assoc = "4kB", 1
        l1d_size, l1d_assoc = "4kB", 2
        l2_size,  l2_assoc  = "32kB", 4

    # L1
    system.cpu.icache = L1ICache()
    system.cpu.dcache = L1DCache()
    system.cpu.icache.size = l1i_size
    system.cpu.icache.assoc = l1i_assoc
    system.cpu.dcache.size = l1d_size
    system.cpu.dcache.assoc = l1d_assoc

    # L2
    system.l2cache = L2Cache()
    #(...)
    # Mem bus
    system.membus = SystemXBar()
    #(...)
```

Dans l'extrait présenté dans le Listing 1, nous pouvons voir les configurations de cache C1 et C2 qui sont appliquées. La configuration C2 se distingue par une associativité plus élevée pour le cache de données (DL1) et le cache de niveau 2 (UL2).

Configuration	IL1	DL1	UL2
C1	size=4kB, assoc=1	size=4kB, assoc=1	size=32kB, assoc=1
C2	size=4kB, assoc=1	size=4kB, assoc=2	size=32kB, assoc=4

Table 1 – Paramètres de cache pour les configurations C1 et C2

2.1.2 Q2 - Comparaison de performances

Énoncé (Q2). Complétez les tableaux 9, 10 et 11. Pour cela vous devez simuler les différentes configurations et collecter les informations necessaires pour IMR, DMR et UCR.

Programme	Conf. Cache - C1	Conf. Cache - C2
P1(normale)	0.000119	0.000119
P2(pointer)	0.000089	0.000089
P3(tempo)	0.000123	0.000123
P4(unrol)	0.000141	0.000141

Table 2 – Resultats de Instructions Miss Rate (IMR)

Dans la Table 2, nous pouvons observer que les taux de miss pour les instructions (IMR) sont identiques pour les deux configurations de cache, C1 et C2. Cela suggère que les modifications apportées à la configuration de cache n'ont pas eu d'impact significatif sur le taux de miss pour les instructions.

Programme	Conf. Cache - C1	Conf. Cache - C2
P1(normale)	0.298317	0.306918
P2(pointer)	0.299730	0.308453
P3(tempo)	0.299726	0.308447
P4(unrol)	0.300091	0.305469

Table 3 – Resultats de Data Cache Miss Rate (DMR)

En ce qui concerne les taux de miss pour les données (DMR), nous pouvons observer une valeur pareille dans la configuration C2 par rapport à la configuration C1. Cela demontre que l'augmentation de l'associativité du cache de données (DL2) et du cache de niveau 2 (UL2) dans la configuration C2 n'a pas eu d'impact significatif sur le taux de miss pour les données.

En ce qui concerne les taux de miss pour le cache unifié (UCR), nous pouvons observer une légère amélioration dans la configuration C2 par rapport à la configuration C1. Cela suggère que l'augmentation de l'associativité du cache de données (DL2) et du cache de niveau 2 (UL2) dans la configuration C2 a eu un impact positif sur le taux de miss pour le cache unifié, bien que l'amélioration soit relativement modeste.

Programme	Conf. Cache - C1	Conf. Cache - C2
P1(normale)	0.437203	0.423355
P2(pointer)	0.437227	0.423262
P3(tempo)	0.437230	0.423259
P4(unrol)	0.434496	0.425145

TABLE 4 – Resultats de Unified Cache Rate (UCR)

2.1.3 Q3 - Localité de références

> **Énoncé (Q3).** Les 4 algorithmes de multiplication de matrices présentent -ils une bonne localité de références pour le code ? Pourquoi ?

Oui. On observe une excellente localité de référence côté instructions : le taux de défaut du cache d'instructions est quasi nul (environ 0,00%) pour l'ensemble des programmes et ce, quelle que soit la configuration de cache (voir Table 2). Cela indique que le flot d'instructions tient dans le cache d'instructions et que les boucles de multiplication réutilisent en permanence le même petit segment de code, ce qui met bien en évidence la localité spatiale et temporelle.

3 Exercice 4 — Architecture big.LITTLE : Dijkstra & BlowFish

3.1 1. Profiling de l'application

3.1.1 Q1 — Pourcentage par classe d'instructions

> **Énoncé (Q1).** Générez le pourcentage de chaque classe d'instructions de `dijkstra` et `BlowFish` (en utilisant `gem5`) et reportez les valeurs dans un tableau.

Objectif. Identifier la *répartition* des classes d'instructions (contrôle, entiers, mémoire, flottants, etc.) afin d'anticiper quelles unités fonctionnelles (ALU, LSU, unités de branchement, etc.) sont les plus sollicitées.

Méthodologie.

— Compilation RISC-V (conforme au sujet) :

```
REPO_ROOT=/path/to/ES201-TP
cd "$REPO_ROOT"
make -C TP4/Projet/dijkstra clean all
make -C TP4/Projet/blowfish clean all
```

— Simulation gem5 en mode SE (CPU OoO) — Architecture A7 :

```
GEM5=/path/to/gem5/build/RISCV/gem5.opt
```

```
"$GEM5" \
  -d TP4/Projet/q1_m5out/m5out_q1_a7_dijkstra TP4/se_A7.py \
  --cmd TP4/Projet/dijkstra/dijkstra_large.riscv \
  --options TP4/Projet/dijkstra/input.dat

"$GEM5" \
  -d TP4/Projet/q1_m5out/m5out_q1_a7_blowfish TP4/se_A7.py \
  --cmd TP4/Projet/blowfish/bf.riscv \
  --options e TP4/Projet/blowfish/input_large.asc \
  TP4/Projet/q1_m5out/output_q1_a7.enc 0123456789ABCDEF
```

— Extraction des pourcentages (tableau final Q1) :

— Extraction des compteurs par classe d'instructions depuis `m5out/stats.txt` (ou script de parsing), puis calcul :

$$\%(\text{classe } i) = 100 \times \frac{N_i}{\sum_j N_j}.$$

TABLE 5 – Pourcentage d'instructions par classe (gem5) — Dijkstra vs BlowFish (A7 et A15)

	A7		**A15**	
Classe	**Dijkstra**	**BlowFish**	**Dijkstra**	**BlowFish**
Entiers (IntAlu + IntMult + IntDiv)	68.225 357	64.424 663	68.225 357	64.424 663
Load (MemRead)	22.241 504	22.855 417	22.241 504	22.855 417
Store (MemWrite)	9.533 089	12.719 215	9.533 089	12.719 215
Contrôle (branch/jump)	0.000 000	0.000 000	0.000 000	0.000 000
Flottants (Float*)	0.000 006	0.000 091	0.000 006	0.000 091
Autres (No_OpClass, etc.)	0.000 045	0.000 614	0.000 045	0.000 614
Total	100.000 001	100.000 000	100.000 001	100.000 000

Résultats.

— **Dijkstra :** Majorité d'instructions **entiers** ($\sim$ 68%) + **mémoire** surtout en lecture (load $\sim$ 22%, store $\sim$ 9,5%).

— **BlowFish :** Entiers dominants ($\sim$ 64%) + **mémoire** plus marquée, avec **plus d'écritures** (store $\sim$ 12,7%) et load $\sim$ 22,9%.

— **Point clé :** Les deux sont dominés par **entiers + accès mémoire** ; BlowFish met davantage de pression sur la mémoire (stores).

3.1.2 Q2 — Catégorie d'instructions à optimiser

> **Énoncé (Q2).** Quelle catégorie d'instructions nécessiterait une amélioration de performances ? Expliquez en quelques lignes (max 5 lignes).

Réponse. Categorie a ameliorer en priorite : Load/Store (memoire). Les entiers dominent en nombre (Table 5), mais ils sont generalement peu couteux ; a l'inverse, les acces

memoire subissent la penalite des **misses** de cache, qui peuvent bloquer le pipeline (stalls) et degrader fortement l'IPC. Il faut donc viser une meilleure hierarchie memoire (latence/taux de miss L1D/L2, bande passante, prefetch), et `BlowFish` est encore plus sensible cote ecritures (stores).

3.1.3 Q3 — Similitudes/divergences comportementales

> **Énoncé (Q3).** Au regard des résultats obtenus lors du TP2, pouvez-vous justifier d'éventuelles similitudes/divergences comportementales entre `dijkstra`, `BlowFish`, `SSCA2-BCS`, `SHA-1` et le produit de polynômes ?

Réponse. L'analyse comparative des profils d'exécution entre `dijkstra`, `BlowFish`, `SSCA2-BCS`, `SHA-1` et le produit de polynômes met en évidence les tendances suivantes :

Convergences.

— **Intensité de calcul :** Les algorithmes cryptographiques (`BlowFish` et `SHA-1`) sollicitent fortement le processeur, principalement via de l'arithmétique entière.

— **Intensité des accès mémoire :** La manipulation de structures de données complexes par `dijkstra` et `SSCA2-BCS` se traduit par un volume important d'opérations de lecture et d'écriture en mémoire.

Divergences.

— Contrairement aux flux d'exécution linéaires de `BlowFish` et `SHA-1`, l'algorithme de `dijkstra` se singularise par une forte densité de sauts conditionnels nécessaires à la détermination des chemins.

— Le produit de polynômes présente un profil hybride, alternant calculs et accès mémoire fréquents, ce qui rend critique l'optimisation des transferts de données.

Compléments d'analyse.

— **Impact des misses :** Lorsque le CPI diminue, les défauts de cache deviennent plus pénalisants ; la réduction des misses (L1/L2) peut alors apporter un gain notable.

— **Irrégularité mémoire et préfetching :** Pour `dijkstra`/`SSCA2-BCS`, les accès irréguliers (*pointer chasing*) rendent le préchargement moins efficace et augmentent la sensibilité à la hiérarchie mémoire et à l'exécution OoO.

3.2 2. Evaluation des performances

3.2.1 Q4 — Cortex A7 : impact de la taille de L1 (L1I = L1D)

> **Énoncé (Q4).** Générez les figures de performances détaillées (performance générale, IPC, hiérarchie mémoire, prédiction de branchement, etc.) en fonction de la taille du cache L1 pour les configurations testées. Analysez les résultats. Quelle configuration de L1 donne les meilleures performances pour le Cortex A7 pour `dijkstra` ? et pour `BlowFish` ?
> **N.B. : Mentionnez les paramètres d'exécution de Gem5 que vous avez utilisé.**

Paramètres d'exécution (gem5).

— Gem5 (mode SE, ISA RISC-V) : `build/RISCV/gem5.opt`

— Script de configuration : `TP4/se_A7.py` (CPU OoO)

— Paramètres CLI utilisés : `-d <outdir>`, `-cmd <binary>`, `-options <args>`, `-l1i-size <NkB>`, `-l1d-size <NkB>`

— Balayage : `-l1i-size = -l1d-size` $\in \{1, 2, 4, 8, 16\}$ kB

— L2 fixé à 512kB.

TABLE 6 – Q4 (Cortex A7) — `dijkstra_large` : métriques vs taille L1 (L1I=L1D, L2=512kB)

L1 (kB)	IPC	CPI	Cycles (M)	I$ miss	D$ miss	L2 miss	Mispred
1	0.231858	4.312987	879.477	0.046418	0.235547	0.000280	0.010266
2	0.239406	4.177012	851.750	0.037714	0.196325	0.000333	0.010268
4	0.249901	4.001583	815.977	0.017283	0.152747	0.000445	0.010266
8	0.271426	3.684245	751.268	0.002980	0.079314	0.000903	0.010263
16	0.278733	3.587658	731.573	0.001091	0.056496	0.001294	0.010263

TABLE 7 – Q4 (Cortex A7) — `blowfish_large` : métriques vs taille L1 (L1I=L1D, L2=512kB)

L1 (kB)	IPC	CPI	Cycles (M)	I$ miss	D$ miss	L2 miss	Mispred
1	0.251957	3.968937	52.348	0.298339	0.181680	0.002570	0.024673
2	0.257932	3.876992	51.136	0.128580	0.147473	0.003898	0.024617
4	0.270185	3.701164	48.817	0.024657	0.096718	0.007233	0.024615
8	0.296596	3.371592	44.470	0.000666	0.018584	0.043697	0.024614
16	0.298072	3.354891	44.249	0.000537	0.014353	0.057827	0.024614

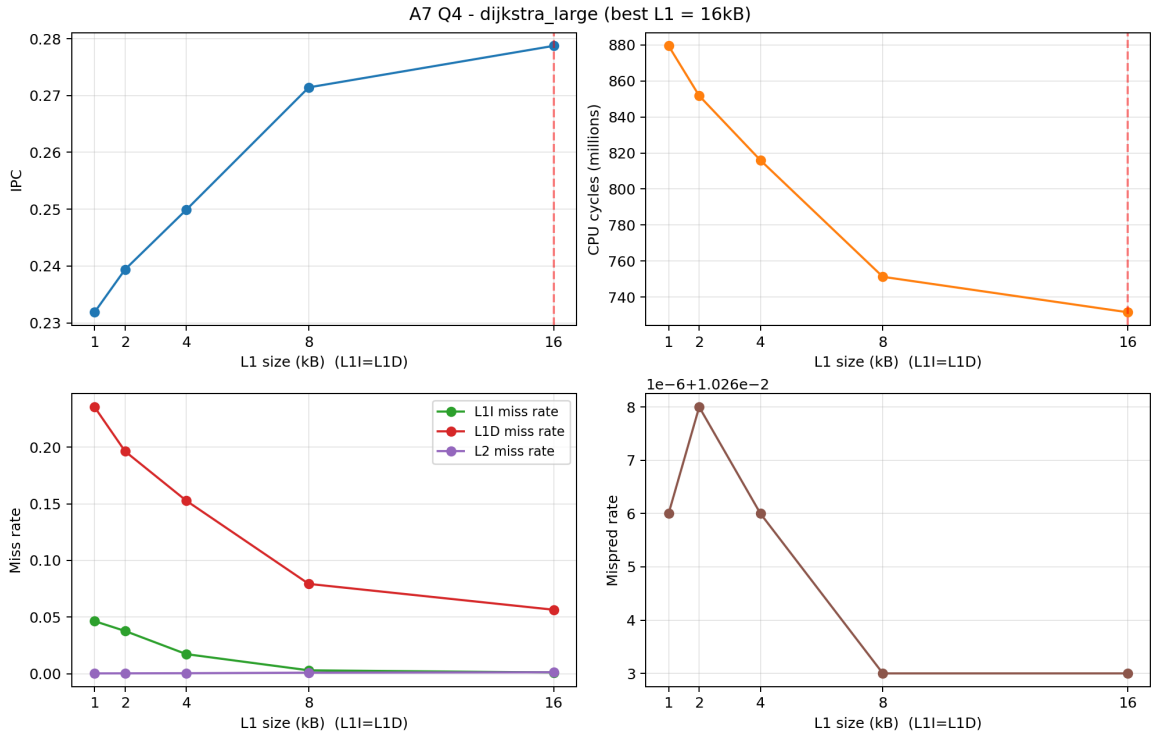

FIGURE 1 – Q4 (A7) — `dijkstra_large` : IPC, cycles, hiérarchie mémoire et prédiction de branchement vs taille L1.

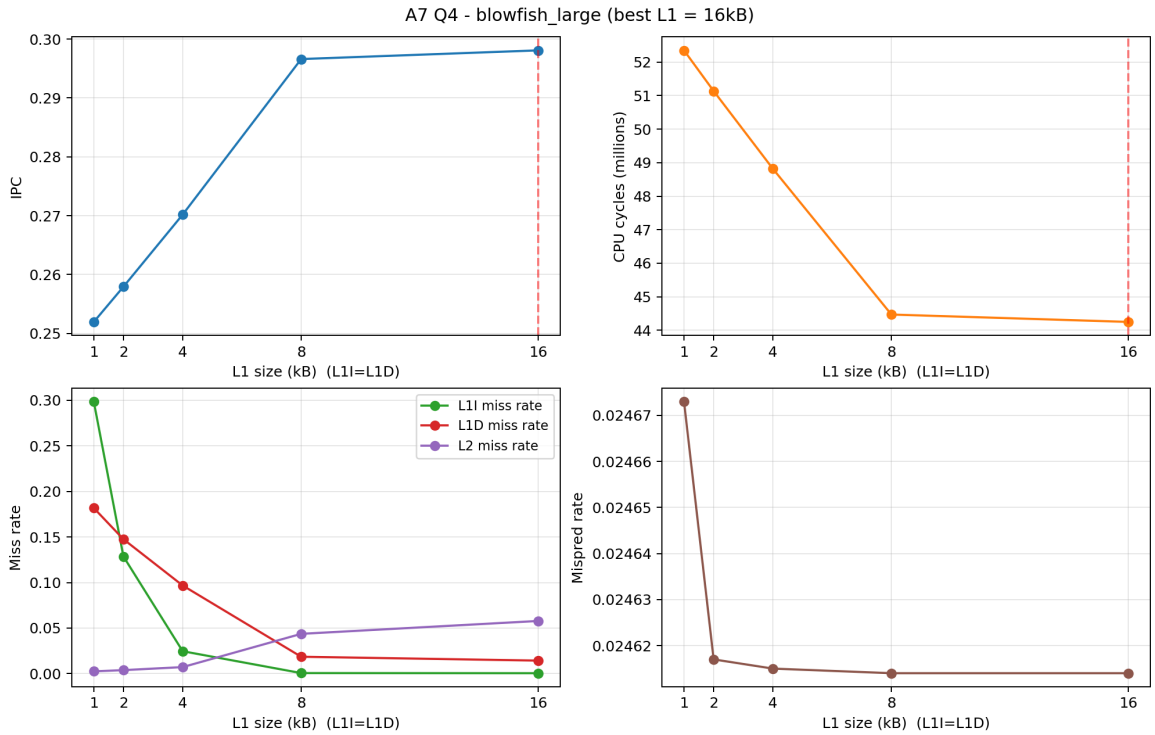

FIGURE 2 – Q4 (A7) — `blowfish_large` : IPC, cycles, hiérarchie mémoire et prédiction de branchement vs taille L1.

Analyse On observe une tendance très claire : quand on augmente la taille de L1 (avec **L1I=L1D**), les **misses I$ et D$ diminuent fortement** et, en parallèle, **l'IPC augmente** tandis que **le nombre de cycles baisse**. Cela a du sens, car un miss de cache force le processeur à attendre (L2/mémoire), ce qui crée des *stalls* et réduit le débit global.

Pour `dijkstra_large`, on voit que la baisse des misses est progressive et continue jusqu'à 16kB, ce qui se traduit par une amélioration régulière des performances : plus de données et d'instructions tiennent en L1, donc moins d'allers-retours vers les niveaux inférieurs.

Pour `blowfish_large`, l'effet est encore plus spectaculaire sur le cache d'instructions : **I$ miss chute très rapidement** (dès 2–4 kB), et **D$ miss** baisse aussi fortement. C'est cohérent avec le fait que BlowFish exécute des boucles intensives : dès que le code et les tables de travail tiennent en cache, on réduit fortement les attentes mémoire.

Enfin, on constate que **le gain se tasse à partir de 8kB** : passer de 8 à 16 kB apporte peu de bénéfice supplémentaire. Cela suggère qu'à 8kB une grande partie du *working set* critique tient déjà dans L1, et que le reste des coûts provient d'autres goulots (accès L2/mémoire résiduels, limites du pipeline, etc.). De plus, **la misprediction reste quasi constante**, ce qui confirme que l'évolution est principalement pilotée par la hiérarchie mémoire.

Meilleure configuration (A7). Sur les points testés, la meilleure performance est obtenue avec **L1I=L1D=16kB** pour `dijkstra` et `BlowFish` (min cycles / max IPC), même si 8kB apparaît déjà comme un bon compromis au vu des gains marginaux.

3.2.2 Q5 — Cortex A15 : impact de la taille de L1 (L1I = L1D)

Énoncé (Q5). Générez les figures de performances détaillées (performance générale, IPC, hiérarchie mémoire, prédiction de branchement, etc.) en fonction de la taille du cache L1 pour les configurations testées. Analysez les résultats. Quelle configuration de L1 donne les meilleures performances pour le Cortex A15 pour `dijkstra` ? et pour `BlowFish` ?
N.B. : Mentionnez les paramètres d'exécution de Gem5 que vous avez utilisé.

Paramètres d'exécution (gem5).

— Gem5 (mode SE, ISA RISC-V) : `build/RISCV/gem5.opt`

— Script de configuration : `TP4/se_A15.py` (CPU OoO)

— Paramètres CLI utilisés : `-d <outdir>`, `-cmd <binary>`, `-options <args>`, `-l1i-size <NkB>`, `-l1d-size <NkB>`

— Balayage : `-l1i-size = -l1d-size` $\in \{2, 4, 8, 16, 32\}$ kB

— L2 fixé à 512kB dans le script.

TABLE 8 – Q5 (Cortex A15) — `dijkstra_large` : métriques vs taille L1 (L1I=L1D, L2=512kB)

L1 (kB)	IPC	CPI	Cycles (M)	I$ miss	D$ miss	L2 miss	Mispred
2	0.654236	1.528500	311.682	0.052157	0.185632	0.000220	0.011167
4	0.716546	1.395584	284.579	0.026767	0.143118	0.000290	0.011140
8	0.901382	1.109407	226.223	0.005627	0.077506	0.000574	0.011522
16	0.981770	1.018568	207.700	0.002244	0.051970	0.000871	0.011546
32	1.141754	0.875845	178.597	0.001183	0.017590	0.002893	0.011588

TABLE 9 – Q5 (Cortex A15) — `blowfish_large` : métriques vs taille L1 (L1I=L1D, L2=512kB)

L1 (kB)	IPC	CPI	Cycles (M)	I$ miss	D$ miss	L2 miss	Mispred
2	1.060149	0.943264	12.441	0.110876	0.176653	0.002642	0.125675
4	1.134343	0.881568	11.627	0.019981	0.124223	0.004438	0.124880
8	1.359661	0.735477	9.701	0.000680	0.034137	0.022683	0.124722
16	1.390119	0.719363	9.488	0.000507	0.029321	0.030426	0.124713
32	1.497467	0.667794	8.808	0.000448	0.001067	0.798194	0.124786

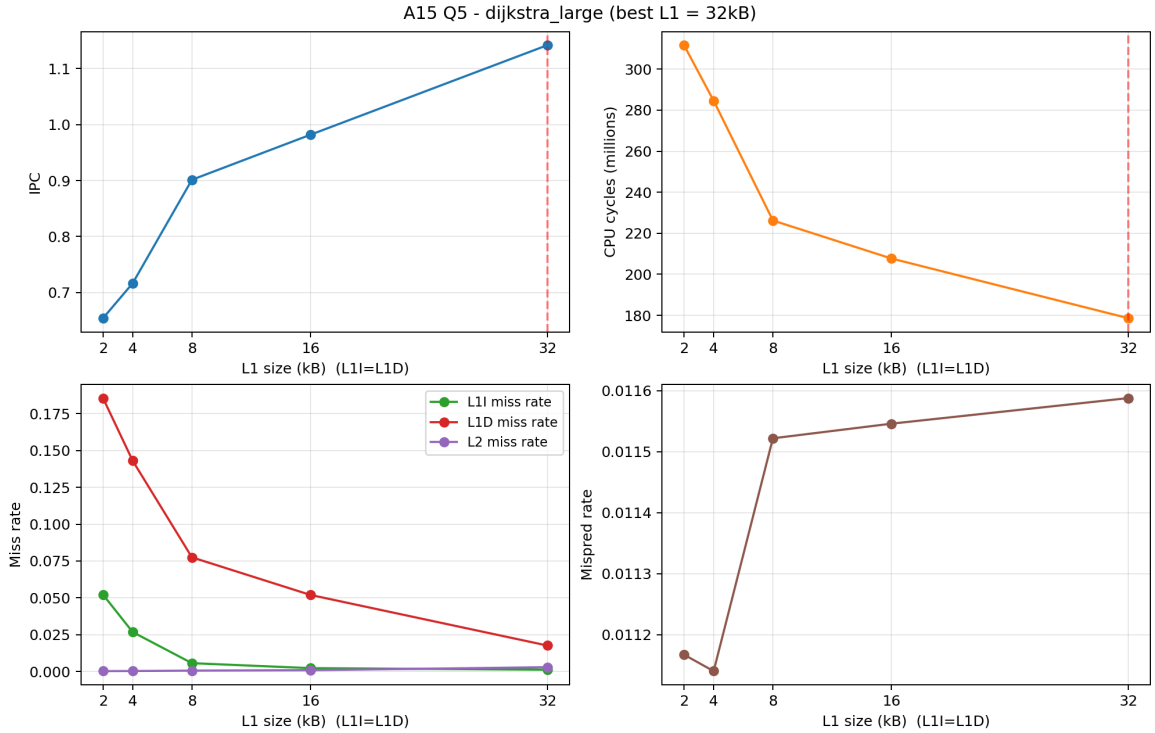

FIGURE 3 – Q5 (A15) — `dijkstra_large` : IPC, cycles, hiérarchie mémoire et prédiction de branchement vs taille L1.

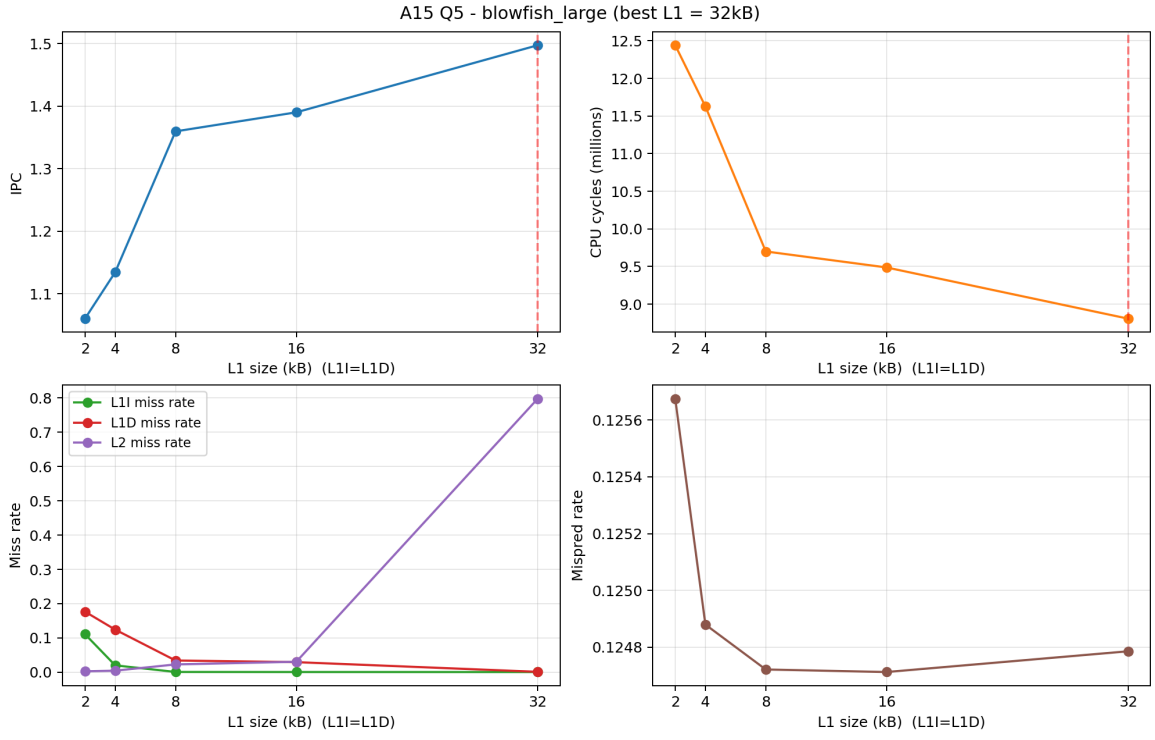

FIGURE 4 – Q5 (A15) — `blowfish_large` : IPC, cycles, hiérarchie mémoire et prédiction de branchement vs taille L1.

Analyse Les résultats des Tables 8 et 9, confirmés par les Figures 3 et 4, montrent une tendance nette : quand la taille de L1 augmente (**L1I=L1D**), **I$ miss** et **D$ miss** chutent, ce qui se traduit

par **moins de cycles** et **un IPC plus élevé**. Cela a du sens, car réduire les misses réduit les attentes vers L2/mémoire et donc les *stalls*.

Pour `dijkstra_large` (Table 8, Figure 3), l'amélioration est régulière : on observe simultanément la baisse des misses (surtout **D$**) et la hausse de l'IPC, ce qui indique que l'application bénéficie directement d'un meilleur taux de hits en L1.

Pour `blowfish_large` (Table 9, Figure 4), le gain est surtout marqué entre 2kB et 8kB : les misses chutent très rapidement et l'IPC augmente fortement, ce qui correspond bien à une charge *bouclée* (code répétitif + tables) dont le *working set* tient progressivement en cache. Le taux de misprediction reste globalement stable (Figures 3 et 4), donc la performance est principalement pilotée par la hiérarchie mémoire.

Remarque (BlowFish, L1=32kB) : pic de miss L2. Sur la Figure 4 et la Table 9, le **miss L2** augmente fortement à 32kB. D'après `stats.txt`, cela s'explique surtout par un **effet de ratio** : seulement $m2547$ accès atteignent L2, et une grande partie sont des misses, alors même que $D\$$ miss devient quasi nul (accès majoritairement servis en L1). La tendance globale (cycles en baisse, IPC en hausse) confirme que l'effet dominant reste la chute des misses L1.

Meilleure configuration (A15). Sur les points testés, la meilleure performance est obtenue avec **L1I=L1D=32kB** pour `dijkstra` et `BlowFish` (min cycles / max IPC).

3.3 3. Efficacité surfacique

3.3.1 Q6 — Paramètres par défaut dans `cache.cfg`

> **Énoncé (Q6).** Observez le fichier de configuration `cache.cfg`. Quels sont les paramètres de cache (taille, taille de bloc, associativité) et la technologie (nm) utilisés par défaut ?

Réponse. D'après les lignes non commentées du fichier `cache.cfg`, la configuration par défaut est :

— Taille du cache : **32 kB** (`-size (bytes) 32768`)

— Taille de bloc (ligne) : **64 B** (`-block size (bytes) 64`)

— Associativité : **2-way** (`-associativity 2`)

— Technologie : **32 nm** (`-technology (u) 0.032`)

Remarque (technologie). Le sujet cible 28 nm, mais la version de CACTI utilisée dans ce TP ne permet pas d'obtenir un résultat stable à 28 nm (abandon interne lors de l'exécution). Conformément à la note de l'énoncé (32 nm si 28 nm n'est pas supporté), les estimations de surface sont donc réalisées en 32 nm (`-technology (u) 0.032`, soit 0.032 um).

3.3.2 Q7 — Surface L1, pourcentage et taille des coeurs

Énoncé (Q7). Quelle est la surface des caches L1 du Tableau 12 (instructions et données) en mm^2 ? Quel pourcentage de la surface totale des coeurs A7 et A15 est occupé par les caches L1 ? En déduire la taille des deux coeurs (hors caches L1). Donnez votre analyse.

Méthode. Nous utilisons CACTI à 32 nm (cf. Q6), avec les paramètres L1 du Tableau 12 :

— A7 : 32 kB, bloc 32 B, associativité 2

— A15 : 32 kB, bloc 64 B, associativité 2

Les fichiers `cache_L1_A7_32nm.cfg` et `cache_L1_A15_32nm.cfg` ont été créés à partir de `cache.cfg`, en modifiant uniquement les paramètres L1 utiles. Pour Q7, on utilise seulement les champs du Tableau 12 suivants : **I-L1$** et **D-L1$** (taille, taille de bloc, associativité).

Dans chaque sortie CACTI, la surface d'un cache L1 est calculée par :

$$S_{L1} = S_{\text{data}} + S_{\text{tag}}$$

Puis :

$$S_{L1,\text{total}} = S_{L1I} + S_{L1D}, \quad \%L1 = \frac{S_{L1,\text{total}}}{S_{\text{coeur+L1}}} \times 100, \quad S_{\text{coeur seul}} = S_{\text{coeur+L1}} - S_{L1,\text{total}}$$

avec $S_{\text{coeur+L1}} = 0.45$ mm^2 pour A7 et 2.0 mm^2 pour A15 (énoncé).

TABLE 10 – Q7 — Résultats de surface (CACTI 32nm)

Métrique	**A7**	**A15**
S_{data} d'un L1 (mm^2)	0.0301	0.0301
S_{tag} d'un L1 (mm^2)	0.0083	0.0044
S_{L1I} (mm^2)	0.0384	0.0346
S_{L1D} (mm^2)	0.0384	0.0346
$S_{L1,\text{total}}$ (mm^2)	0.0769	0.0691
$\%L1$ de la surface coeur+L1	17.08%	3.46%
$S_{\text{coeur seul}}$ (mm^2)	0.3731	1.9309

Analyse. Les L1 représentent une part nettement plus importante sur A7 ($\approx 17.1\%$) que sur A15 ($\approx 3.46\%$). Autrement dit, la surface "logique coeur" (hors L1) est dominante sur A15, ce qui est cohérent avec un coeur plus large et plus agressif micro-architecturalement. À paramètres L1 comparables (32 kB, 2-way), la différence principale provient ici de la taille de bloc (32 B vs 64 B), qui modifie le coût de la partie tag.

3.3.3 Q8 — Variation de la taille L1 et nouvelle surface totale

Énoncé (Q8). Faire varier la taille des caches L1 pour le Cortex A7 et le Cortex A15, puis donner (en mm^2) les surfaces L1 obtenues. Avec la configuration L2 utilisée précédemment (512 kB), en déduire pour chaque valeur les nouvelles surfaces totales (coeur + L1 + L2), et présenter les résultats sous forme de graphes.

Méthode. Pour chaque architecture, nous avons fait un balayage des tailles L1 (L1I = L1D) avec CACTI en 32 nm. Les paramètres de blocs/associativité suivent le Tableau 12 :

— A7 : taille L1 variable (1, 2, 4, 8, 16 kB), bloc 32 B, associativité 2 ; L2 = 512 kB, bloc 32 B, associativité 8

— A15 : taille L1 variable (2, 4, 8, 16, 32 kB), bloc 64 B, associativité 2 ; L2 = 512 kB, bloc 64 B, associativité 16

Les calculs utilisés sont :

$$S_{L1,\text{total}} = S_{L1I} + S_{L1D}, \qquad S_{\text{total}} = S_{\text{coeur sans L1}} + S_{L1,\text{total}} + S_{L2}$$

avec les surfaces coeur sans L1 obtenues en Q7.

TABLE 11 – Q8 — A7 : surface L1 totale et surface totale (coeur + L1 + L2)

L1 (kB)	$S_{L1,\text{total}}$ **(mm²)**	S_{total} **(mm²)**
1	0.0085	0.8274
2	0.0209	0.8397
4	0.0114	0.8302
8	0.0257	0.8445
16	0.0324	0.8513

TABLE 12 – Q8 — A15 : surface L1 totale et surface totale (coeur + L1 + L2)

L1 (kB)	$S_{L1,\text{total}}$ **(mm²)**	S_{total} **(mm²)**
2	0.0202	2.3506
4	0.0102	2.3406
8	0.0236	2.3539
16	0.0282	2.3586
32	0.0691	2.3995

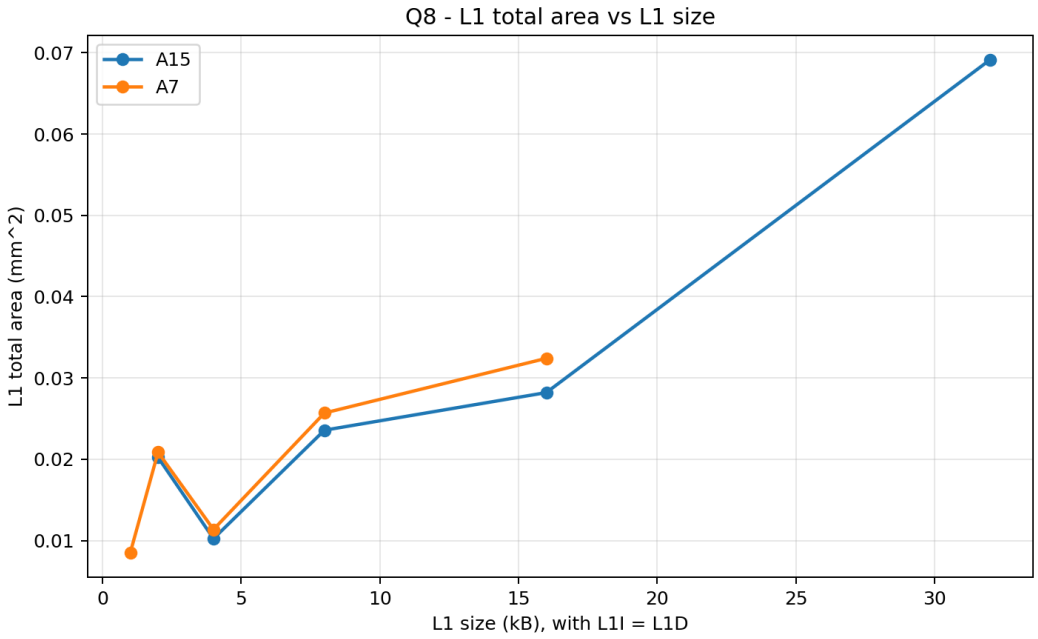

FIGURE 5 – Q8 — Surface L1 totale en fonction de la taille L1 (A7 et A15), en valeurs absolues (mm²), pas en pourcentage.

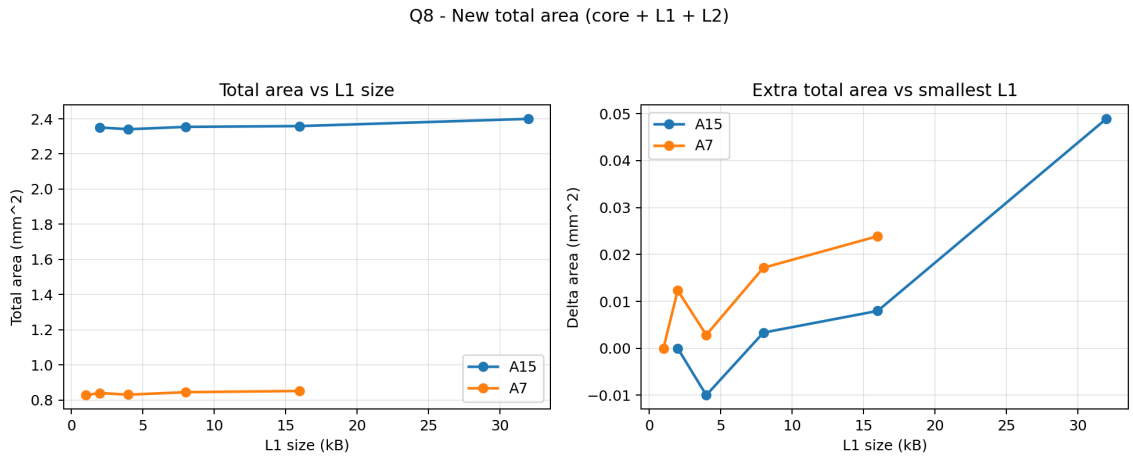

FIGURE 6 – Q8 — Surface totale en fonction de la taille L1 (gauche) et variation de surface par rapport a la plus petite L1 testée (droite).

Analyse. Les courbes de la Figure 5 montrent que la surface des caches L1 augmente globalement quand la taille L1 augmente. Ce résultat est logique : plus de capacité implique plus de cellules mémoire et donc plus de surface occupée. Important : cette figure est une comparaison en mm^2 absolus (et non une comparaison en %). Donc, si deux points paraissent proches, cela ne veut pas dire que les deux architectures ont la même taille globale ; cela veut seulement dire que la surface L1 estimée par CACTI est proche pour ces configurations.

3.3.4 Q9 — Efficacite surfacique (IPC / mm^2) selon la taille de L1

Énoncé (Q9). En prenant en compte les deux dimensions (performance et surface) pour les deux processeurs considérés, donnez pour chaque configuration de L1 l'efficacite surfacique de chaque processeur.
N.B. : Efficacite surfacique = IPC / surface(mm^2).

Méthode. Pour la performance, nous utilisons l'IPC mesuré sous gem5 :
— A7 : IPC issus de Q4 (tailles L1 = 1, 2, 4, 8, 16 kB)
— A15 : IPC issus de Q5 (tailles L1 = 2, 4, 8, 16, 32 kB)
Pour la surface, nous utilisons les surfaces totales estimées en Q8 avec CACTI : coeur sans L1 + L1I + L1D + L2 (avec L1I = L1D dans notre balayage). L2 est fixe a 512 kB (technologie 32 nm). Pour chaque point :

$$\text{efficacite surfacique} = \frac{\text{IPC}}{\text{surface totale } (mm^2)}$$

TABLE 13 – Q9 — A7 : efficacite surfacique (IPC / mm^2) selon L1

L1 (kB)	Surface totale (mm^2)	IPC (dijkstra)	IPC/mm^2 (dijkstra)	IPC (blowfish)	IPC/mm^2 (blowfish)
1	0.8274	0.2319	0.2802	0.2520	0.3045
2	0.8397	0.2394	0.2851	0.2579	0.3072
4	0.8302	0.2499	0.3010	0.2702	0.3254
8	0.8445	0.2714	0.3214	0.2966	0.3512
16	0.8513	0.2787	0.3274	0.2981	0.3502

TABLE 14 – Q9 — A15 : efficacite surfacique (IPC / mm^2) selon L1

L1 (kB)	Surface totale (mm^2)	IPC (dijkstra)	IPC/mm^2 (dijkstra)	IPC (blowfish)	IPC/mm^2 (blowfish)
2	2.3506	0.6542	0.2783	1.0601	0.4510
4	2.3406	0.7165	0.3061	1.1343	0.4846
8	2.3539	0.9014	0.3829	1.3597	0.5776
16	2.3586	0.9818	0.4163	1.3901	0.5894
32	2.3995	1.1418	0.4758	1.4975	0.6241

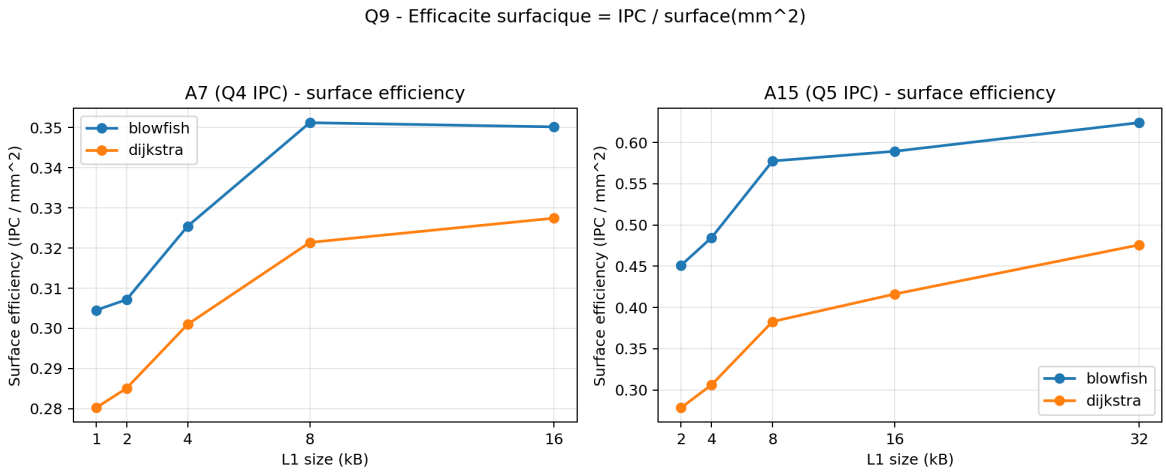

FIGURE 7 – Q9 — Efficacite surfacique (IPC / mm^2) en fonction de la taille L1, pour A7 (gauche) et A15 (droite).

Analyse. La Figure 7 et les Tables 13 et 14 montrent que l'efficacite surfacique augmente globalement quand on augmente L1. C'est un resultat logique ici, car l'augmentation de surface totale entre deux points reste relativement faible (la surface est dominée par le coeur sans L1 et par L2), alors que l'IPC beneficie directement de la baisse des misses quand L1 grandit.

Pour A7, on observe clairement un effet de rendements decroissants : entre 4 kB et 8 kB, l'efficacite surfacique progresse nettement (environ +6.8% sur `dijkstra` et +7.9% sur `blowfish`), alors qu'entre 8 kB et 16 kB le gain est faible (environ +1.9% sur `dijkstra` et quasi nul sur `blowfish`). Cela indique qu'a partir de 8 kB, une grande partie du *working set* critique tient deja dans L1 : augmenter davantage L1 apporte peu en IPC, donc peu en IPC/mm^2. Ainsi, meme si 16 kB est le maximum pour `dijkstra`, 8 kB apparait comme un choix tres proche et plus ëquilibre.

Pour A15, la tendance est plus favorable a de grandes L1 : le passage de 16 kB a 32 kB augmente l'efficacite surfacique de maniere plus significative (environ +14.3% sur `dijkstra` et +5.9% sur `blowfish`), ce qui justifie que 32 kB soit la meilleure configuration sur les points testes.

3.4 4. Efficacité énergétique

3.4.1 Q10 — Puissance consommee a la frequence maximale

Énoncé (Q10). Quelle puissance en mW consomme chaque processeur a la frequence maximale ?

Réponse. Les donnees du sujet indiquent une consommation de 0.10 mW/MHz (A7) et 0.20 mW/MHz (A15), avec des frequences maximales de 1.0 GHz (A7) et 2.5 GHz (A15). On convertit en MHz (1.0 GHz = 1000 MHz, 2.5 GHz = 2500 MHz) puis :

$$P = (\text{mW/MHz}) \times f_{\max}(\text{MHz})$$

— A7 : $0.10 \times 1000 = 100$ mW

— A15 : $0.20 \times 2500 = 500$ mW

3.4.2 Q11 — Efficacite energetique (IPC / mW) selon la taille de L1

Énoncé (Q11). Avec le meme protocole que precedemment, et en prenant en compte les deux dimensions (energie et surface) pour les deux processeurs consideres, donnez pour chaque configuration de L1 l'efficacite energetique de chaque processeur (a frequence maximale).
N.B. : Efficacite energetique = IPC / consommation energie (mW).

Méthode. Nous utilisons l'IPC mesure sous gem5 (A7 : Q4, A15 : Q5) et la puissance a frequence maximale calculee en Q10. Ici, la puissance est supposee constante pour un coeur donne (d'apres les donnees du sujet), donc :

$$\text{efficacite energetique} = \frac{\text{IPC}}{P_{\max}(\text{mW})}$$

avec $P_{\max} = 100$ mW pour A7 et $P_{\max} = 500$ mW pour A15.

Table 15 – Q11 — A7 : efficacite energetique (IPC / mW) selon L1

L1 (kB)	IPC (dijkstra)	IPC/mW (dijkstra)	IPC (blowfish)	IPC/mW (blowfish)
1	0.2319	0.002319	0.2520	0.002520
2	0.2394	0.002394	0.2579	0.002579
4	0.2499	0.002499	0.2702	0.002702
8	0.2714	0.002714	0.2966	0.002966
16	0.2787	0.002787	0.2981	0.002981

Table 16 – Q11 — A15 : efficacite energetique (IPC / mW) selon L1

L1 (kB)	IPC (dijkstra)	IPC/mW (dijkstra)	IPC (blowfish)	IPC/mW (blowfish)
2	0.6542	0.001308	1.0601	0.002120
4	0.7165	0.001433	1.1343	0.002269
8	0.9014	0.001803	1.3597	0.002719
16	0.9818	0.001964	1.3901	0.002780
32	1.1418	0.002284	1.4975	0.002995

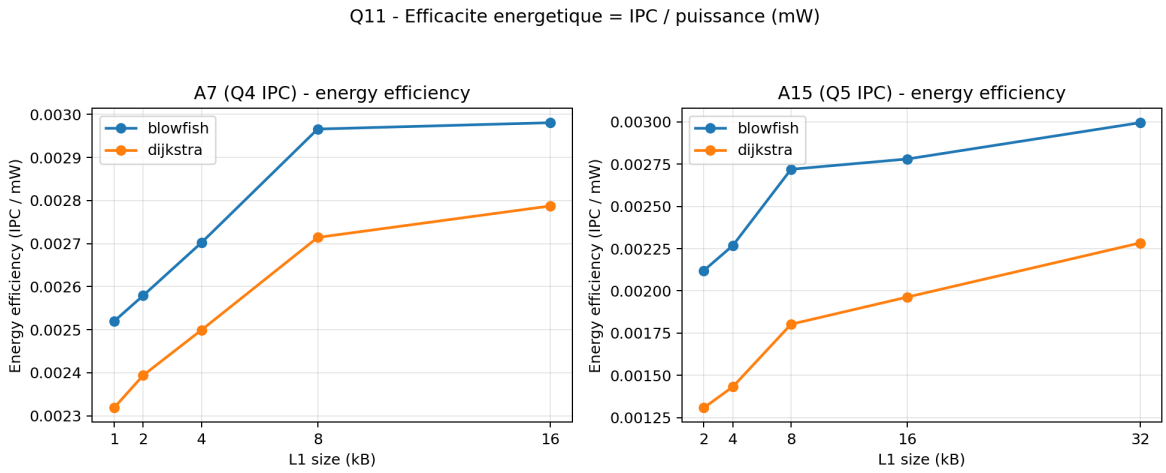

FIGURE 8 – Q11 — Efficacite energetique (IPC / mW) en fonction de la taille L1, pour A7 (gauche) et A15 (droite).

Analyse. La Figure 8 et les Tables 15 et 16 donnent directement l'efficacite energetique (IPC/mW) pour chaque taille de L1. Dans ce modele, $P_{\max}$ est constant par coeur, donc l'evolution d'IPC/mW suit exactement l'evolution de l'IPC.

3.5 5. Architecture système big.LITTLE

3.5.1 Q12 — Choix de la configuration L1 pour un systeme big.LITTLE

> **Énoncé (Q12).** Avec un esprit de concepteur de systeme, et en se basant sur les resultats de Q10 et Q11, proposez la meilleure configuration du cache L1 du processeur big.LITTLE pour les applications `dijkstra` et `blowfish` individuellement.

Proposition (par application). Les gains en % ci-dessous sont des gains relatifs d'efficacite energetique (IPC/mW) entre deux tailles L1 comparees.

— **Application `dijkstra` :**

— A7 : **16 kB** (meilleur IPC/mW sur les points testes, Table 15). Entre 4 kB → 8 kB, l'efficacite energetique augmente d'environ +8.6%, alors que 8 kB → 16 kB n'apporte qu'environ +2.7% : le gain devient donc nettement plus marginal.

— A15 : **32 kB** (meilleur IPC/mW, Table 16). On observe encore des gains importants dans la zone haute : +8.9% entre 8 kB → 16 kB, puis +16.3% entre 16 kB → 32 kB, ce qui justifie de pousser jusqu'a 32 kB pour cette application.

— **Application `blowfish` :**

— A7 : **8 kB** (Table 15). Le saut 4 kB → 8 kB est d'environ +9.8% d'IPC/mW, tandis que 8 kB → 16 kB n'apporte qu'environ +0.5% : la courbe est presque en plateau a partir de 8 kB.

— A15 : **32 kB** (meilleur IPC/mW, Table 16). Entre 8 kB → 16 kB, le gain est faible (+2.2%), mais 16 kB → 32 kB apporte encore environ +7.7%, donc 32 kB reste le meilleur point mesure.

3.6 6. Facultatif

3.6.1 Q13 (facultatif) — Equivalence des choix et compromis

> **Énoncé (Q13).** Les configurations proposees sont-elles equivalentes ? Proposer eventuellement un compromis et conclure sur les applications etudiees.

Compromis retenu. Pour conserver une *seule* configuration de L1 capable de supporter correctement les deux applications sur un systeme big.LITTLE, nous retenons :

— **A7 : L1 = 8 kB**

— **A15 : L1 = 32 kB**

Justification (energie/performance). Sur A7 (Table 15), augmenter de 8 kB a 16 kB apporte un gain d'IPC/mW faible : +2.7% pour `dijkstra` et +0.5% pour `blowfish`. Sur A15 (Table 16), au contraire, le gain entre 16 kB et 32 kB reste important : +16.3% pour `dijkstra` et +7.7% pour `blowfish`, ce qui justifie le choix de 32 kB pour le coeur "big".

Justification (surface). Le surcout en surface totale reste modere : sur A7, passer de 8 kB a 16 kB augmente la surface d'environ +0.007 mm^2 ($\approx$ +0.8%, Table 11), et sur A15, passer de 16 kB a 32 kB augmente la surface d'environ +0.041 mm^2 ($\approx$ +1.7%, Table 12).

3.6.2 Q14 (facultatif) — Approche de specification pour un domaine robotique embarque

> **Énoncé (Q14).** Proposez une approche pour la specification d'une architecture avec plusieurs applications dans un domaine specifique.

Domaine vise. Nous nous placons dans le contexte **robotique embarquee** (robot mobile, drone, bras robotique) : systeme autonome avec capteurs (camera, IMU, LiDAR, etc.), boucles de controle temps reel, et traitement local (*edge*) sous contraintes d'energie et de surface.

Contraintes (hypotheses realistes).

— **Temps reel :** certaines taches (commande moteurs, stabilisation, securite) ont des deadlines et une contrainte de jitter.

— **Energie/autonomie :** pour un robot mobile/donne sur batterie, minimiser l'energie par tache (ou maximiser IPC/mW) pour preserver l'autonomie.

— **Thermique :** souvent fanless ; privilegier des gains *soutenables* (eviter une architecture qui throttle rapidement).

— **Surface/cout :** les caches et les coeurs "big" coutent en mm^2 (et donc en cout), il faut justifier chaque augmentation.

— **Memoire :** la DRAM est lente et energivore ; limiter les misses (MPKI) et la bande passante est cle (surtout pour vision/SLAM).

— **Surete :** degrader gracieusement (toujours garantir le controle/safety), meme si les taches lourdes (perception) saturent.

Ensemble representatif d'applications. Pour specifier une architecture, on choisit un ensemble de cas d'usage du domaine (avec des poids si besoin). Exemple typique **robotique embarquee** :

— **Planification de trajectoire :** `dijkstra` / A* (graphes, acces memoire moins reguliers).

— **Perception :** traitement image (feature extraction) ou inference legere (CNN), charge importante et parfois vectorisable.

— **Localisation/SLAM :** estimation d'etat (EKF/graph-SLAM), souvent plus flottant et sensible a la memoire.

— **Controle :** boucles PID/commande moteurs, charge faible mais deadlines strictes.

— **Communication securisee :** `blowfish` (chiffrement symetrique, entiers + memoire).

Caracterisation des applications (ce qu'il faut mesurer). L'idee est d'identifier *le goulot d'etranglement* de chaque application :

— **Mix d'instructions :** p. ex. Table 5 montre que `dijkstra` et `blowfish` sont majoritairement **entiers** (∼64–68%) et **memoire** (loads+stores ∼32%). En robotique, il faut aussi identifier les taches plus **flottantes/vectorielles** (vision, SLAM), potentiellement candidates a DSP/-SIMD/NPU.

— **Intensite memoire :** MPKI L1/L2, taux de misses, bande passante ; distingue "memory-bound" vs "compute-bound".

— **Sensibilite a la taille de cache :** faire varier L1 (comme en Q4–Q5) et observer le point ou les gains deviennent marginaux. Dans nos resultats, `blowfish` sature vite (plateau), alors que `dijkstra` profite davantage d'une L1 plus grande.

— **Contraintes temps reel :** latence *pire-cas* et jitter pour les boucles de controle/safety (a garantir meme sous charge).

Metrices et objectif d'optimisation. En robotique embarquee, on cherche rarement le maximum de performance brute ; on optimise plutot un compromis sous contraintes temps reel :

— **Contraintes "dures" :** deadlines/jitter pour controle et safety (verification sur le pire-cas).

— **Objectif "souple" :** maximiser un score ponderé (perf/W, perf/mm^2) pour perception/planification.

— Concretement : utiliser les metriques de Q9 (IPC/mm^2) et Q11 (IPC/mW), et conserver des configurations proches du front de Pareto.

Espace de conception (parametres a explorer).

— **Heterogeneite :** big.LITTLE (un coeur "big" pour perception/planification + un coeur "little" pour controle/safety et l'energie).

— **Caches :** tailles L1I/L1D, associativite, taille de bloc ; L2 (souvent partagee) et sa taille.

— **Frequences/DVFS :** points de fonctionnement (basse/moyenne/haute) et politique de migration des taches.

Decision finale (exemple de methode).

1. Explorer quelques tailles L1 par coeur et mesurer l'IPC (gem5), puis estimer surface/energie (CACTI + modele simple).
2. Retenir les configurations non-dominees (Pareto) et choisir un point qui respecte les contraintes (latence/energie/surface).
3. Specifier aussi **le mapping** : reserver le coeur "little" aux taches temps reel (controle/safety) pour limiter l'interference, et executer perception/planification sur le coeur "big" (et/ou accelerateurs) selon les besoins de performance.

Resultat attendu. Au final, la specification n'est pas seulement une taille de cache : c'est une *architecture + politique d'utilisation* (type de coeurs, caches, frequences, et regles d'ordonnancement) justifiee par des mesures de performance, d'energie et de surface sur un ensemble d'applications representatif du domaine.