Test Microprocessors Architecture Configuration

Javier Andres Tarazona Jimenez
Ingénieur Degree Programme
STIC
ENSTA Paris
Paris, France
javier-andres.tarazona@ensta-paris.fr

Jair Anderson Vasquez Torres
Ingénieur Degree Programme
STIC
ENSTA Paris
Paris, France
jair-anderson.vasquez@ensta-paris.fr

Maeva Noukoua
Ingénieur Degree Programme
STIC
ENSTA Paris
Paris, France
maeva-sandy.noukoua@ensta-paris.fr

Carlos
Ingénieur Degree Programme
STIC
ENSTA Paris
Paris, France
carlos@ensta-paris.fr

***Abstract*—...**
***Index Terms*—...**

I. Exercice 3

II. Exercice 4

A. Profiling (Q1)

a) Objectif.: Le *profiling* de l'*instruction mix* consiste à mesurer la répartition des instructions exécutées par grandes catégories (calcul entier, mémoire, contrôle, etc.). Cette information est essentielle en architecture : elle permet d'identifier *où se situe la pression dominante* (unités de calcul, hiérarchie mémoire, prédiction de branchements), et donc d'anticiper quels leviers microarchitecturaux sont les plus pertinents.

b) Méthode.: Nous avons simulé les exécutions avec gem5 en mode SE (ISA RISC-V), en configuration de type A7 (`se_A7.py`). Les compteurs proviennent des instructions *committed* (retired). Le nombre total d'instructions exécutées est $N = \texttt{simInsts}$. Pour chaque catégorie c, on note I_c le nombre d'instructions appartenant à c. Le pourcentage associé est :

$$\mathrm{pct}(c) = 100 \times \frac{I_c}{N}.$$

Les valeurs sont arrondies à deux décimales (somme $\approx 100\%$).

TABLE I
Répartition des instructions exécutées (profiling) — configuration A7.

Catégorie	Dijkstra (A7)		Blowfish (A7)	
	Count	%	Count	%
ALU entier	22118141	44.09%	1882443	55.42%
Chargements (Load)	11799269	23.52%	771841	22.72%
Stockages (Store)	4853941	9.68%	408487	12.03%
Branches (contrôle)	9870255	19.67%	334128	9.84%
Mult/Div entier	1517371	3.02%	6	0.00%
Flottant (FP)	12	0.00%	12	0.00%
Autres	10476	0.02%	18	0.00%
TOTAL (`simInsts`)	**50169465**	**100.00%**	**3396935**	**100.00%**

c) Résultats détaillés.:

d) Synthèse par familles (lecture plus "architecture").: Afin de mieux comparer les pressions microarchitecturales, on regroupe les catégories en trois familles : *calcul entier* (ALU + Mult/Div), *mémoire* (Load + Store) et *contrôle* (Branches).

TABLE II
Agrégation par familles : calcul, mémoire et contrôle (A7).

Famille	Dijkstra (%)	Blowfish (%)	Δ (BF – Dij) [points]
Calcul entier (ALU + Mult/Div)	47.11	55.42	+8.31
Mémoire (Load + Store)	33.20	34.75	+1.55
Contrôle (Branches)	19.67	9.84	-9.83
Reste (FP + Autres)	≈ 0.02	≈ 0.00	≈ 0

e) Interprétation (comparaison chiffrée).: Les deux applications présentent une part mémoire comparable ($\approx 33.20\%$ pour Dijkstra contre $\approx 34.75\%$ pour Blowfish), ce qui indique que la hiérarchie mémoire (L1/L2) reste un levier important dans les deux cas. En revanche, Dijkstra est nettement plus *control-heavy* : les branches représentent 19.67% des instructions contre 9.84% pour Blowfish, soit environ $\times 2$ de densité de contrôle. À l'inverse, Blowfish est davantage *compute-heavy* : la part d'ALU entier atteint 55.42% contre 44.09% pour Dijkstra (+11.33 points). Les instructions FP sont négligeables (quelques occurrences attribuables à un surcoût du runtime plutôt qu'au noyau algorithmique).

B. Q2 : Catégorie à améliorer

Sur **Dijkstra**, la part **contrôle** est élevée (**branches = 19.67%**) et s'ajoute à une pression **mémoire** déjà importante (**load+store = 23.52% + 9.68% = 33.20%**) : améliorer la **prédiction de branchements** (et réduire les bulles/flush) est donc un levier prioritaire pour ce code très *branchy*. Sur **Blowfish**, le profil est surtout **compute-heavy** en entier (**ALU = 55.42%** avec **branches = 9.84%**) : l'amélioration la plus pertinente concerne le **débit/latence des opérations entières**

(ALU), tandis que la mémoire reste secondaire bien que non négligeable (**load+store = 34.75%**). Ainsi, la catégorie à optimiser dépend de l'application : **contrôle (branches) pour Dijkstra** et **calcul entier (ALU) pour Blowfish**.

C. Q3 : Comparaison avec les charges du TP2 (SSCA2-BCS, SHA-1, poly_mult)

Pour comparer Dijkstra et Blowfish aux benchmarks du TP2, on s'appuie sur une lecture *architecture* en trois familles : (i) **calcul entier** (ALU + Mult/Div), (ii) **mémoire** (Load + Store) et (iii) **contrôle** (Branches). Cette classification renseigne directement sur les ressources dominantes (unités entières, hiérarchie mémoire, prédiction de branchement) et sur la sensibilité aux mécanismes OoO (ROB/LSQ) et au masquage de latence.

a) Dijkstra vs. SSCA2-BCS (graphes, accès irréguliers).: Dijkstra présente une composante **contrôle** élevée (**19.67%** de branches) ainsi qu'une pression **mémoire** marquée (**Load+Store = 33.20%**). Ce profil est typique d'un traitement de graphe : parcours, conditions dépendantes des données, et accès non séquentiels (structures et indices variables), ce qui dégrade la localité et rend les préchargements moins efficaces. On s'attend donc à des comportements proches de **SSCA2-BCS** (également orienté graphes) : forte sensibilité au front-end (prédiction de branchement) et à la capacité du cœur OoO à tolérer des latences mémoire (fenêtres ROB/LSQ, MLP).

b) Blowfish vs. SHA-1 (noyaux compute-bound entiers).: Blowfish est davantage **compute-heavy** en entier : **ALU+Mult/Div = 55.42%**, avec un **contrôle plus faible** (**9.84%** de branches), tout en conservant une part mémoire non négligeable (**Load+Store = 34.75%**) liée aux buffers et tables. Cette dynamique se rapproche de **SHA-1**, qui est encore plus dominé par le calcul entier : dans notre run *SHA small*, on obtient **78.34%** d'ALU entier, **Load+Store = 16.29%** et **5.37%** de branches. Ainsi, SHA-1 et Blowfish sont tous deux des noyaux à contrôle relativement réduit et à calcul entier majoritaire ; la différence principale est que **SHA-1 est plus "pur compute"**, tandis que **Blowfish** conserve davantage d'accès mémoire (p.ex. tables/S-boxes), ce qui peut augmenter la sensibilité aux caches lorsque les ensembles de données grandissent.

c) poly_mult (produit de polynômes / convolution).: À l'inverse des charges de type graphe, **poly_mult** manipule généralement des tableaux et des boucles régulières : les accès sont souvent *séquentiels* (bonne localité spatiale), et la part de branches est typiquement faible (boucles simples). On s'attend donc à un comportement plus proche d'un noyau *streaming* : la performance dépend alors (i) du **débit de calcul** (multiplications/additions) et (ii) de la **bande passante mémoire** lorsque les tableaux dépassent les caches. En conséquence, lorsqu'on rend le cœur plus agressif (meilleur IPC théorique), la latence/bande passante mémoire tend à devenir un facteur plus visible : l'optimisation de la hiérarchie mémoire (caches, miss rate effectif, éventuel préchargement) devient alors un levier majeur, surtout pour les grandes tailles.

d) Synthèse.: En résumé, **Dijkstra** se rapproche des workloads *graph/irregular* comme **SSCA2-BCS** (contrôle + mémoire élevés), alors que **Blowfish** se situe entre un noyau *compute entier* et une charge mémoire modérée, et se compare naturellement à **SHA-1** (mais avec plus d'accès mémoire). Enfin, **poly_mult** est attendu plus régulier et potentiellement limité par la bande passante mémoire sur grands tableaux, avec un contrôle faible.

TABLE III
RÉPARTITION DES INSTRUCTIONS EXÉCUTÉES POUR SHA-1 (SMALL).

Catégorie	Count	%
ALU entier	10032072	78.34%
Chargements (Load)	1496416	11.69%
Stockages (Store)	589104	4.60%
Branches (contrôle)	687432	5.37%
Mult/Div entier	89	0.00%
Flottant (FP)	12	0.00%
Autres	55	0.00%
TOTAL	**12805180**	**100.00%**

D. Q11 : Efficacité énergétique

a) Rappel méthodologique.: L'efficacité énergétique est définie comme le rapport entre la performance obtenue (IPC) et la puissance consommée à fréquence maximale :

$$\text{Efficacité énergétique} = \frac{IPC}{P\ (\text{mW})}.$$

Les puissances maximales ont été déterminées à la question Q10 à partir des consommations en mW/MHz et des fréquences maximales.

b) Puissance maximale des processeurs.: Pour le Cortex A7 :

$$P_{A7} = 0.10\ \text{mW/MHz} \times 1000\ \text{MHz} = 100\ \text{mW}.$$

Pour le Cortex A15 :

$$P_{A15} = 0.20\ \text{mW/MHz} \times 2500\ \text{MHz} = 500\ \text{mW}.$$

Ainsi :

$$P_{A7} = 100\ \text{mW}, \quad P_{A15} = 500\ \text{mW}.$$

Cortex A7

TABLE IV
EFFICACITÉ ÉNERGÉTIQUE DU CORTEX A7 EN FONCTION DE LA TAILLE DU CACHE L1.

L1 (KB)	IPC	Efficacité énergétique ($IPC/100$)
1		
2		
4		
8		
16		

TABLE V
EFFICACITÉ ÉNERGÉTIQUE DU CORTEX A15 EN FONCTION DE LA TAILLE DU CACHE L1.

L1 (KB)	IPC	Efficacité énergétique ($IPC/500$)
2		
4		
8		
16		
32		

Cortex A15

c) Analyse qualitative attendue.: L'efficacité énergétique augmente lorsque l'IPC augmente, c'est-à-dire lorsque la réduction des cache misses améliore l'utilisation des unités de calcul.

Cependant, malgré un IPC généralement plus élevé pour le Cortex A15, sa consommation énergétique est cinq fois supérieure à celle du Cortex A7 (500 mW contre 100 mW). Ainsi, l'amélioration d'IPC doit être suffisamment significative pour compenser ce surcoût énergétique.

On s'attend donc à observer :

- Une amélioration de l'efficacité énergétique lorsque la taille du L1 augmente jusqu'à un point de saturation.
- Une stabilisation des gains au-delà d'une certaine taille de cache (rendements décroissants).
- Une efficacité énergétique globalement supérieure pour le Cortex A7, en raison de sa consommation nettement plus faible.

En conclusion, le Cortex A7 devrait offrir un meilleur compromis performance/énergie pour des charges modérées, tandis que le Cortex A15 privilégie la performance brute au détriment de la consommation.

E. Q12: Architecture big.LITTLE

a) Objectif.: L'architecture big.LITTLE associe un cœur *économe* (A7) et un cœur *performant* (A15). Pour chaque application, on souhaite proposer une configuration de caches L1 (I-$L1$ et D-$L1$ de même taille) qui offre le meilleur compromis *performance/énergie* à fréquence maximale, en s'appuyant sur les résultats des questions Q10–Q11 [1].

b) Principe de décision (règle utilisée).: Pour chaque processeur et chaque application, on retient :

- la configuration de $L1$ située au *coude* des courbes (rendement décroissant), c.-à-d. la plus petite taille de $L1$ à partir de laquelle l'IPC n'augmente plus significativement ;
- et/ou la configuration maximisant l'efficacité énergétique $E = IPC/P$.

Cette stratégie reflète un choix "concepteur" : éviter d'augmenter $L1$ lorsque le gain de performance devient marginal.

F. Recommandation pour Dijkstra

a) Cortex A7.: D'après les résultats de Q11, l'efficacité énergétique de Dijkstra sur A7 est maximale (ou proche du maximum) pour $L1 = [\underline{\ldots}]\,\mathrm{KB}$. Au-delà, l'IPC progresse faiblement tandis que l'intérêt énergétique devient marginal. Nous proposons donc :

$$L1_{A7}^{(\mathrm{Dijkstra})} = [\underline{\ldots}]\ \mathrm{KB}.$$

b) Cortex A15.: Sur A15, l'IPC est supérieur mais la consommation est plus élevée, ce qui réduit l'efficacité énergétique. Les résultats montrent un coude / plateau à $L1 = [\underline{\ldots}]\,\mathrm{KB}$. Nous proposons :

$$L1_{A15}^{(\mathrm{Dijkstra})} = [\underline{\ldots}]\ \mathrm{KB}.$$

c) Synthèse Dijkstra.: La configuration big.LITTLE recommandée pour Dijkstra est :

$$(A7, A15) = ([\underline{\ldots}]\ \mathrm{KB},\ [\underline{\ldots}]\ \mathrm{KB}).$$

G. Recommandation pour Blowfish

a) Cortex A7.: Pour Blowfish, les résultats de Q11 indiquent que l'efficacité énergétique sur A7 est optimale (ou quasi optimale) pour $L1 = [\underline{\ldots}]\,\mathrm{KB}$. Nous proposons :

$$L1_{A7}^{(\mathrm{Blowfish})} = [\underline{\ldots}]\ \mathrm{KB}.$$

b) Cortex A15.: Pour A15, l'augmentation de $L1$ améliore l'IPC jusqu'à $L1 = [\underline{\ldots}]\,\mathrm{KB}$, puis les gains deviennent faibles. Au regard de l'efficacité énergétique, on retient :

$$L1_{A15}^{(\mathrm{Blowfish})} = [\underline{\ldots}]\ \mathrm{KB}.$$

c) Synthèse Blowfish.: La configuration big.LITTLE recommandée pour Blowfish est :

$$(A7, A15) = ([\underline{\ldots}]\ \mathrm{KB},\ [\underline{\ldots}]\ \mathrm{KB}).$$

d) Discussion générale.: On s'attend à ce que le cœur A7 soit privilégié lorsque la contrainte énergétique domine (efficacité élevée), tandis que le cœur A15 est mobilisé lorsque la performance brute est requise. Les tailles retenues correspondent au meilleur compromis observé entre gain d'IPC et coût énergétique, en évitant les configurations où l'augmentation de $L1$ n'apporte plus de bénéfice notable.

H. Q13 : équivalence des configurations et compromis

a) Comparaison des configurations optimales.: Les tailles de cache L1 retenues pour Dijkstra et Blowfish ne sont pas nécessairement identiques. Cette différence s'explique par la nature distincte des applications :

- Dijkstra présente une pression importante sur le contrôle et la mémoire,
- Blowfish est davantage dominée par le calcul entier.

Ainsi, la taille optimale de L1 peut différer selon le profil microarchitectural de la charge.

b) équivalence.: Si les tailles optimales obtenues pour les deux applications sont différentes (par exemple $L1 = [\underline{\ldots}]\,\mathrm{KB}$ pour Dijkstra et $L1 = [\underline{\ldots}]\,\mathrm{KB}$ pour Blowfish), alors les configurations ne sont pas strictement équivalentes.

En revanche, si les performances et l'efficacité énergétique restent proches dans une même plage de tailles, on peut considérer les configurations comme quasi équivalentes.

c) Proposition de compromis.: Dans un système réel, le cache L1 est fixe et ne peut être redimensionné dynamiquement. Il est donc pertinent de choisir une taille offrant :

- une performance proche de l'optimum pour les deux applications,
- une efficacité énergétique satisfaisante,
- un coût en surface raisonnable.

Nous proposons ainsi un compromis à $L1 = \underline{[...]}\,\mathrm{KB}$, correspondant au meilleur équilibre global observé.

d) Conclusion sur les applications étudiées.: Les résultats montrent que le dimensionnement optimal du cache dépend fortement du profil applicatif. Les applications orientées contrôle/mémoire bénéficient davantage d'un L1 suffisamment dimensionné pour réduire les miss, tandis que les charges compute-heavy sont plus sensibles au débit des unités de calcul qu'à l'augmentation excessive du cache.

I. Q14 : Approche méthodologique pour la spécification d'une architecture

La conception d'une architecture destinée à exécuter plusieurs applications dans un domaine spécifique doit suivre une approche systématique :

1) **Profiling représentatif :** Identifier un ensemble d'applications représentatives du domaine et mesurer leur instruction mix ainsi que leurs métriques de performance.
2) **Identification des pressions dominantes :** Déterminer si les charges sont principalement compute-bound, memory-bound ou control-bound.
3) **Exploration paramétrique :** Faire varier les paramètres microarchitecturaux clés (taille des caches, largeur pipeline, prédiction de branchement, etc.) afin d'observer leur impact sur les métriques de performance, d'énergie et de surface.
4) **Analyse multi-critères :** évaluer les compromis performance/énergie/surface à l'aide d'indicateurs normalisés (IPC, efficacité énergétique, efficacité surfacique).
5) **Choix robuste :** Sélectionner une configuration offrant des performances stables sur l'ensemble des applications, même si elle n'est pas optimale pour chacune individuellement.

Cette approche permet de concevoir une architecture équilibrée, robuste et adaptée au domaine cible, plutôt qu'optimisée pour un cas particulier.

REFERENCES

[1] TP4, "Microprocessors Architecture Configuration," document de travaux pratiques du cours.

[2] A. Huamán, "Feature Description," *OpenCV Documentation* (OpenCV 4.14.0-pre), accessed Feb. 6, 2026. [Online]. Available: https://docs.opencv.org/4.x/d5/dde/tutorial_feature_description.html