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

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

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

I. Exercice 4 — 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).

A. 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**.

B. 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%**

REFERENCES

[1] 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