Quels objets en NVRAM ? Placement en mémoires de travail hétérogènes

Tristan Delizy, Stéphane Gros, Kevin Marquet, Matthieu Moy, Tanguy Risset, Guillaume Salagnac

▶ **To cite this version:**

Tristan Delizy, Stéphane Gros, Kevin Marquet, Matthieu Moy, Tanguy Risset, et al.. Quels objets en NVRAM ? Placement en mémoires de travail hétérogènes. Compas 2018 - Conférence d'informatique en Parallélisme, Architecture et Système, Jul 2018, Toulouse, France. pp.1-8. hal-01891398

HAL Id: hal-01891398

https://hal.science/hal-01891398

Submitted on 9 Oct 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compas'2018 : Parallélisme/ Architecture / Système
ENSEEIHT - Toulouse, France, du 03 au 06 juillet 2018

Quels objets en NVRAM ? Placement en mémoires de travail hétérogènes

Tristan Delizy*, Stephane Gros‡, Kevin Marquet*, Matthieu Moy†, Tanguy Risset*, Guillaume Salagnac*

∗ Univ Lyon, INSA Lyon, Inria, CITI, F-69621 Villeurbanne, France.
‡ eVaderis, 51 Avenue Jean Kuntzmann, 38330 Montbonnot, France.
† Univ Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, LIP UMR5668, F-69342, Lyon, France.

Résumé
La généralisation de l'internet des objets passe par plusieurs défis scientifiques et technologiques, parmi lesquels la réduction et surtout la maîtrise de la consommation énergétique des systèmes embarqués. Pour cela, une piste prometteuse est offerte par l'émergence de nouvelles technologies de mémoire non volatile adressable (NVRAM), permettant la conception de systèmes-sur-puce sans cache, avec une mémoire principale composée de plusieurs bancs hétérogènes. Les performances très diverses de ces bancs en termes de latence ou d'énergie permettent d'espérer des gains substantiels, à condition de placer judicieusement chacune des données manipulées par le programme. Si ce problème de placement est relativement bien étudié dans le contexte des données statiques, le cas de l'allocation dynamique reste largement inexploré. Dans cet article, nous étudions l'influence de l'hétérogénéité mémoire, et des décisions de placement, sur la consommation énergétique du programme. Notre approche repose sur l'utilisation d'un simulateur de plate-forme pour obtenir une trace des allocations et des accès mémoire, puis sur une formulation du problème de placement sous forme de programme linéaire. Nos résultats expérimentaux montrent que les techniques d'allocation dynamique classiques ne donnent pas satisfaction sur les architectures mémoire considérées, et que de nouveaux allocateurs mémoire multi-tas sont nécessaires.

Mots-clés : NVRAM, allocation dynamique, embarqué

---

1. Contexte et état de l'art

La consommation énergétique des petits systèmes embarqués reste un des verrous majeurs pour le développement de l'Internet de Objets (IoT). Typiquement, c'est la durée de vie de la batterie qui conditionne la durée de vie du système, et donc son utilité. Les pistes de recherche pour lever ce verrou sont très variées, par exemple : récolte d'énergie dans l'environnement [13], réduction du coût énergétique des communications, ou encore calcul à basse consommation. Une des avancées clés dans cette direction est l'émergence de nouvelles technologies de mémoire non-volatile adressable (NVRAM). Mais ces technologies posent aussi de pouveaux défis en termes de programmation logicielle.
Traditionnellement, l'architecture mémoire d'un système embarqué contraint est basée sur deux types de technologies : d'une part une mémoire dite de travail et d'autre part une mémoire dite de stockage. La mémoire de travail est directement adressable par le processeur. Elle doit

donc être implémentée par une technologie rapide, en général à base de transistors (SRAM) ou plus rarement de condensateurs (eDRAM). Un inconvénient majeur de ces technologies est leur consommation d'énergie : la mémoire doit être alimentée en permanence sous peine d'en perdre le contenu. On parle ainsi de mémoire *volatile*. Pour stocker des données sur le plus long terme, il faut recourir à d'autres technologies dites *non-volatiles*, par exemple la mémoire Flash. Cependant, ces mémoires présentent souvent des latences d'accès plus importantes et ne sont typiquement pas adressables à l'octet près depuis le processeur. Ces différences se répercutent jusqu'au niveau logiciel : l'accès au stockage persistant passe par une interface explicite, par exemple un système de fichiers.

Ces dernières années ont vu l'émergence et le développement de nouvelles techologies de mémoire dite NVRAM [18] combinant non-volatilité et adressabilité à l'octet, tout en présentant une latence d'accès relativement faible et une plus grande densité d'intégration. Ces propriétés estompent la distinction entre stockage et mémoire de travail et sont à l'origine d'un nombre important de travaux récents, en particulier dans le domaine des systèmes d'exploitation [4, 15].

Dans le domaine de l'embarqué contraint, l'apparition de systèmes-sur-puce à NVRAM est très prometteuse en termes de consommation. En effet, il devient envisageable de couper volontairement et fréquemment l'alimentation du système pour économiser l'énergie. Ce mode de fonctionnement dit *normalement éteint* [20] est naturellement adapté au contexte de l'IoT, dans lequel les applications ont typiquement un profil temporel périodique, alternant entre courtes périodes d'activité intense et longues périodes d'attente passive. La combinaison de NVRAM et de mécanismes logiciels ad hoc [1, 16] permet d'éteindre et rallumer le système régulièrement, de façon transparente pour le programmeur.

Toutefois, aucune technologie de NVRAM n'est encore suffisamment avancée pour remplacer entièrement la SRAM. Par exemple, la mémoire à changement de phase (PCM) [24] souffre d'une endurance limitée. La mémoire magnétorésistive (MRAM) impose une latence d'accès asymétrique entre lecture et écriture. D'autres problèmes similaires affectent chaque technologie : coût énergétique en écriture trop élevé, densité trop faible, ou simplement une latence d'accès trop supérieure à celle de la SRAM [18]. Pour pallier ces limitations, les concepteurs de systèmes embarqués sont amenés à concevoir des architectures combinant plusieurs types de mémoires.

Dans ce travail, nous nous intéressons à de petits systèmes embarqués sans cache, dotés d'une telle mémoire de travail hétérogène. En d'autres termes, nous considérons une architecture dont la Scratch Pad Memory (SPM) se compose de plusieurs bancs aux caractéristiques variées, par exemple plusieurs NVRAM et une SRAM [16].

Du point de vue logiciel, ce genre d'architecture pose un problème de placement : chaque donnée, qu'il s'agisse de code, de variables statiques, de la pile d'exécution ou du tas d'allocation dynamique, peut donc être placée dans des mémoires aux propriétés différentes. Ces décisions de placement ont un impact sur les performances de l'application (temps d'exécution), sur le vieillissement de la mémoire (endurance), ou sur la durée de vie de la batterie (consommation).

Dans ce travail, nous nous focalisons sur le placement des objets alloués dynamiquement, considérant que le placement des autres types de données est déja largement couvert dans la littérature. En effet, de nombreux travaux [2, 3, 6, 10–12, 14, 21, 22] proposent des techniques de compilation permettant de placer et déplacer partiellemment le code, les données statiques, ou encore la pile d'exécution, entre plusieurs bancs mémoire. En général, il s'agit d'insérer aux endroits propices du programme des instructions pour copier les éléments concernés d'un banc à l'autre. Même s'ils sont en général formulés pour des architectures DRAM+SPM, tous ces travaux restent essentiellement valides dans le contexte des NVRAM. Cependant, ils sont par nature limités aux éléments connus au moment de la compilation : code, variables, etc.

Bien moins nombreux [5, 19] sont les travaux qui s'intéressent à l'allocation dynamique. L'ap-

proche de Dominguez *et.al.* [5] est similaire aux techniques de compilation mentionnées plus haut, mais inclut également les objets du tas. Les résultats expérimentaux montrent que l'utilisation d'une SPM permet d'améliorer considérablement les performances du programme. Cependant, cette technique repose sur un profilage préalable de l'exécution, et surtout ne s'applique qu'à des applications avec un très petit nombre d'allocations. Afin de pouvoir traiter des scénarios plus complexes, Mück et Frölich [19] demandent au programmeur d'ajouter des annotations dans son code source afin de suggérer à l'allocateur un choix de banc pour chaque objet. Leurs résultats montrent un gain significatif de performance à l'exécution. Mais cette approche repose sur l'expertise du programmeur qui doit connaître finement l'architecture mémoire cible, ainsi que le profil d'allocation de son programme.

2. Notre approche

Dans le cadre de cette étude nous considérons une architecture mémoire basée sur une SPM directement connectée au processeur, possédant des bancs de latences variables. L'objectif de cet article est dans un premier lieu d'évaluer le gain en temps d'exécution possiblement atteignable par les décisions de placement intelligentes. Pour les zones mémoires dédiées au tas nous évaluons une mémoire rapide type SRAM (1 cycle de latence) et une mémoire lente, pouvant être composé de NVRAM (10 cycles de latence). Nous choisissons d'avoir deux exécutions de référence : une ou l'intégralité du tas est situé en mémoire lente et une autre en mémoire rapide. Pour des architectures incluant deux bancs mémoire (SRAM et NVRAM), les stratégies évalués peuvent être positionnées par rapport à ces exécutions de référence. Les contributions de cet article sont les suivantes :

— Étudier l'impact du système d'allocation multi-tas sur les performances de l'application.
— Proposer plusieurs stratégies de placement et évaluer quel gain pourrait générer un placement efficace.
— Évaluer quelles métriques sont déterminantes dans le choix de placement.

Stratégies de placement

Étant donné l'exécution d'une application, une stratégie de placement désigne l'algorithme qui décider de quel banc mémoire va accueillir l'objet. On distingue notamment les stratégies *en ligne* et les stratégies *hors ligne*. Les stratégies *hors ligne* appliquent juste des solutions générées spécifiquement pour une exécution d'une application particulière, leur but est d'évaluer les performances atteignables dans la résolution du problème. Au contraire les stratégies *en ligne* sont des algorithmes proposant une résolution du problème durant l'exécution, pouvant être embarquées dans le système final.

3. Allocation dynamique dans plusieurs tas

Pour être capable d'allouer des blocs mémoires libres satisfaisant à la fois aux requêtes de l'application et aux décisions de placement des stratégies considérées, certaines modifications de l'allocateur mémoire sont nécessaires. Dans un premier temps nous utilisons plusieurs instances d'un allocateur, de manière à maintenir les informations relatives aux différents bancs mémoires dans des structures de données différentes. Nous séparons donc la décision de placement du choix de l'adresse où allouer l'objet à l'intérieur du banc choisi.

Une première stratégie consiste à allouer systématiquement dans le tas rapide et en cas d'échec allouer dans le tas lent. Cette approche dégrade toutefois significativement les performances de l'allocateur mémoire. En effet le tas rapide est maintenu plein par cette stratégie, générant beaucoup d'échecs d'allocation qui doivent être redirigés vers le tas lent. Or, un algorithme d'allocation mémoire n'est pas prévu pour devoir échouer régulièrement. Au contraire, quand un tel algorithme n'a plus de mémoire disponible il en demande plus au système au travers

d'un appel système coûteux en temps d'exécution ou, dans le cas d'une exécution sans système d'exploitation provoque une défaillance du système. Utiliser un allocateur standard mène donc à de mauvaises performances.

Maintenir à jour dans l'allocateur mémoire une variable contenant la taille du plus grand bloc libre dans ce tas permet de résoudre ce problème, au prix des adaptations suivantes :

- désallocation : si le bloc libéré, après consolidation, est plus grand que le plus grand bloc libre on récupère sa taille comme taille de plus grand bloc libre.
- allocation : implique un second balayage partiel de la structure de donnée du tas après la majorité des allocations.

La modification de la routine d'allocation étant trop coûteuse sur les performances globales du système, la variable contenant la taille du plus grand bloc libre est gérée de manière paresseuse. Cette implémentation évite en majeure partie l'impact négatif de devoir échouer une partie des allocations dans le tas rapide et n'implique pas de surcoût significatifs en terme de temps passé dans l'allocateur mémoire pour l'application d'une stratégie en ligne gloutonne.

Nous utilisons cette stratégie en incluant cette adaptation de l'allocateur dans nos benchmarks comme référence de stratégie simple à mettre en place. Notre but par la suite est de proposer de meilleures stratégies et de mesurer le gain par rapport à cette stratégie de référence.

4. Résolution hors-ligne du problème de placement

Dans cette section nous présentons la formulation du problème de placement sous la forme d'un problème d'optimisation linéaire en nombres entiers (ILP, *Integer Linear Programming*). Nous présentons également une métrique permettant de générer une solution gloutonne hors ligne plus efficace que la solution gloutonne en ligne. Bien que ces approches ne soient pas utilisables en ligne et que le placement calculé reste spécifique à une exécution elles nous permettent d'estimer l'impact qu'une stratégie en ligne pourrait espérer avoir.

4.1. Simplification du problème

Dans une première approche du problème nous ne considérons que les choix de placement entre deux tas, un rapide et un lent. De plus, pour des raisons de solvabilité du problème nous négligeons la fragmentation du tas dans la construction des solutions hors ligne, impliquant que certaines allocations seront redirigées vers le tas lent. Les solutions construites hors ligne seront donc sous optimales à l'exécution. Mais le système d'allocation étant étant adapté aux conditions d'exécution multi-tas elles restent efficaces pour estimer des gains de performances atteignables.

4.2. Programme linéaire en nombres entiers

On considère une variable par bloc à allouer $x_i \in \{0, 1\}$ spécifiant le tas dans lequel le bloc i est alloué ($x_i = 0 \Leftrightarrow$ le bloc est alloué dans le premier tas). Les contraintes du problème sont qu'à chaque instant la somme des tailles des blocs en cours d'utilisation dans chaque tas doit être inférieure à la taille du tas. On abstrait ainsi les problèmes de fragmentation, comme expliqué précédemment. La fonction objectif à minimiser correspond à la somme des latences des différents accès en tenant compte des bancs dans lesquels ils sont placés (x_i).

4.3. Métrique de densité d'accès

Nous souhaitons également dégager des métriques plus légères à calculer permettant de classer les objets. Le nombre d'accès à un objet est une bonne indication de son importance, alors que sa taille représente l'encombrement dans le tas dans l'espace d'adressage et que sa durée de vie reflète l'encombrement temporel. Nous proposons donc une métrique de densité d'accès :

$$density(b) = \frac{nb_R(b) + nb_W(b)}{size(b) * (lifespan(b))}$$

Nous utilisons cette métrique pour construire d'autres solutions hors ligne faisant la même simplification du problème que la construction des solutions ILP. Ces solutions sont construites de manière gloutonne sur la métrique de densité d'accès ci-dessus.

5. Resultats expérimentaux

5.1. Applications embarquées et architectures mémoire

Nos expérimentations portent sur plusieurs programmes représentatifs d'applications embarquées. Le programme **jpeg**, tiré de la suite de benchmarks Mibench [9], est un algorithme de compression d'image. Le programme **json**, basé sur une bibliothèque *C* destinée à l'embarqué [8], fait le parsing puis la sérialisation d'un fichier JSON. Le programme **dijkstra**, tiré de la suite de benchmarks Mibench [9], fait des calculs de plus court chemin dans un graphe représenté par une matrice d'adjacence. Enfin, le programme **h263**, tiré de la suite de benchmarks Mediabench II [7], est un algorithme de compression vidéo. Comme illustré à la figure 1(a), ces différents programmes présentent une diversité considérable en terme de nombre d'objets alloués dynamiquement et de taille de ces données. La troisième colonne donne l'empreinte mémoire, c'est à dire la taille maximum atteinte au cours de l'exécution par l'ensemble du tas.

Notre gestionnaire mémoire est constitué d'un *placeur* appliquant les stratégies présentées en sections 3 et 4, et de plusieurs *allocateurs*. Pour l'allocation proprement dite, nous utilisons dans chaque tas l'algorithme `dlmalloc` [17] augmenté des adaptations évoquées à la section 3.

Notre plateforme de simulation est développée en SystemC/TLM, et basée sur le simulateur de jeu d'instruction MIPS32 du projet Soclib [23]. Ce processeur simule, au cycle près, un pipeline à trois étages et exécute les instructions *dans l'ordre*. Le reste de la plateforme est également simulé au cycle près. Nous exécutons chaque application sur plusieurs architectures mémoire, comme illustré à la figure 1(b). Les deux architectures A0 et A3 n'ont qu'un seul banc mémoire, donc ne présentent pas de problème de placement. Les performances obtenues sur ces configurations de référence nous servent à «borner» les performances attendues sur les autres, où le tas est partagé entre une mémoire rapide et une mémoire lente. Afin que le problème de placement se pose effectivement malgré la disparité des tailles allouées, nous adaptons à chaque fois la quantité de mémoire disponible dans le simulateur en fonction de l'empreinte mémoire.

App.	nb objets	empreinte mem.
jpeg	21	228 ko
json	1638	38 ko
dijkstra	14980	10 ko
h263	53821	1232 ko

(a)

Arch.	SRAM	NVRAM
A0	100%	0%
A1	25%	75%
A2	5%	95%
A3	0%	100%

(b)

FIGURE 1 – (a) Applications et (b) architectures mémoire étudiées dans les expérimentations.

5.2. Disscussion

Pour chaque application cible nous avons évalué le temps d'exécution en considérant les combinaisons d'architectures et de stratégies suivantes :

- exécutions de référence / **baseline** sur A0 et A3 : ne considérant qu'un seul tas ces architectures ne sont pas soumises au problème de placement et définissent un intervalle de valeurs à priori atteignables sur les autres architectures.
- stratégie **baseline** sur A1 et A2 : stratégie gloutonne en ligne de référence (section 3)
- stratégie **ilp** sur A1 et A2 : stratégie hors ligne basée sur la résolution ILP du problème
- stratégie **density** sur A1 et A2 : stratégie hors ligne basée sur la métrique de densité

La figure 2 ci-dessous présente les résultats obtenus par simulation pour les différentes stratégies sur différentes architectures mémoire pour toute les applications cibles. Pour chaque application

le temps passé dans l'allocateur dans les architectures 1 et 2 (multi-tas) est supérieur à celui correspondant à l'allocation des mêmes objets dans un seul tas (architectures 0 et 3). Cette augmentation reste toutefois limitée du à l'adaptation de l'allocateur mémoire au contexte multi-tas et nous permet d'observer des gains significatifs, notamment dans l'utilisation des stratégies hors ligne. Malgré tout la stratégie **baseline**, sur l'architecture 2, présente de mauvais résultats dans les cas où l'application a besoin d'allouer beaucoup d'objets sans avoir énormément de calcul à réaliser dessus. On constate aussi que la stratégie hors ligne **density** réalise de très bons résultats, proches de ceux de la stratégie **ilp**. Ces deux stratégies étant calculées sur le problème simplifié ignorant la fragmentation, une partie des allocations prévues dans le tas rapide ne peut pas être alloué dans celui-ci, ce qui permet d'expliquer que dans certains cas la stratégie **density** réalise un temps d'exécution plus faible.

La figure 3 présente ces résultats par application. L'application *jpeg* montre des résultats très semblables selon les différentes stratégies. Cette application n'allouant qu'une vingtaine d'objets d'une manière quasi statique (allocation en début d'exécution, désallocation en fin), il n'est pas surprenant qu'elle ne présente que peu d'interaction avec les stratégies de placement proposées. L'analyse syntaxique de fichiers json impliquant la création d'un arbre d'objets en mémoire, cette application présente un cas d'usage intensif de l'allocateur mémoire, où moins d'accès mémoire par objets sont réalisés mettant en avant le coût de maintenir et d'utiliser un tas plein pendant la majorité de l'exécution. Dans le cas d'une application réalisant de nombreux calculs sur des objets variés comme h263, les résultats montrent un écart important entre la stratégie gloutonne en ligne **baseline** et les stratégies hors ligne. Dans le cas de dijkstra, l'application présente des pics d'allocation réguliers mais n'alloue que des objets petits et d'une taille identique (24 octets). Dans ces conditions, l'adaptation du système de gestion de la mémoire à un contexte multi-tas est moins efficace et le temps passé dans l'allocateur est un coût trop important pour dégager de bons résultats.

Les expériences que nous avons réalisées mettent en lumière les interactions entre l'utilisation d'une architecture de mémoire de travail hétérogène pour le tas et l'allocation mémoire en elle même. Il semble dès lors pertinent de penser qu'une stratégie en ligne efficace se doit aussi de s'intéresser à l'état du tas pour prendre les décisions de placement. On note aussi que les cas trop simplistes ne permettent pas la mise en évidence des problèmes liés à l'interaction avec l'allocateur.

App.	Stratégie	Temps exec. (10^6cycles)				% passé dans l'allocateur			
		A0	A1	A2	A3	A0	A1	A2	A3
jpeg	baseline	92.2	97.0	111	116	0,013%	0,014%	0,013%	0,012%
	ilp	n/a	97.0	111	n/a	n/a	0,013%	0,013%	n/a
	density	n/a	97.0	112	n/a	n/a	0,013%	0,012%	n/a
json	baseline	6.26	7.21	7.55	7.43	13%	15%	16%	13%
	ilp	n/a	6.99	7.24	n/a	n/a	15%	15%	n/a
	density	n/a	7.12	7.26	n/a	n/a	14%	14%	n/a
dijkstra	baseline	201	222	224	223	3,7%	5,0%	5,2%	4,0%
	ilp	n/a	212	221	n/a	n/a	4,5%	4,6%	n/a
	density	n/a	212	221	n/a	n/a	4,5%	4,6%	n/a
h263	baseline	24.0k	26.2k	29.3k	29.4k	0,12%	0,15%	0,15%	0,12%
	ilp	n/a	24.9k	26.6k	n/a	n/a	0,15%	0,16%	n/a
	density	n/a	24.4k	27.1k	n/a	n/a	0,14%	0,15%	n/a

FIGURE 2 – Temps d'exécution pour les quatre architectures mémoires et pourcentage du temps passé dans le gestionnaire mémoire

6. Conclusion et perspectives

Cet article propose une étude empirique de l'influence de l'utilisation d'une architecture de mémoire de travail hétérogène pour le tas d'applications embarquées. Nous avons proposé une formulation du problème de placement entre deux tas et mis en lumière les interactions entre ce problème et le problème de l'allocation mémoire dynamique en général. La résolution du problème hors ligne nous a permis de démontrer le gain possible d'une telle approche de même que l'insuffisance d'une solution naïve.

Ce travail est préliminaire à l'étude et la proposition de solutions en ligne au problème de placement. Il est envisageable à ce stade d'adopter une méthode de profilage d'application permettant de proposer une stratégie en ligne basée sur d'autres exécutions de la même application sur des jeux de données différents. La prise en compte de l'état des tas maintenus dans les différentes mémoires semble aussi une piste prometteuse dans la conception de solutions en ligne. Enfin, à plus long terme la réalisation de solutions en ligne adaptatives, basées sur des informations récoltées durant l'exécution laisse espérer une résolution en ligne efficace du problème de placement.

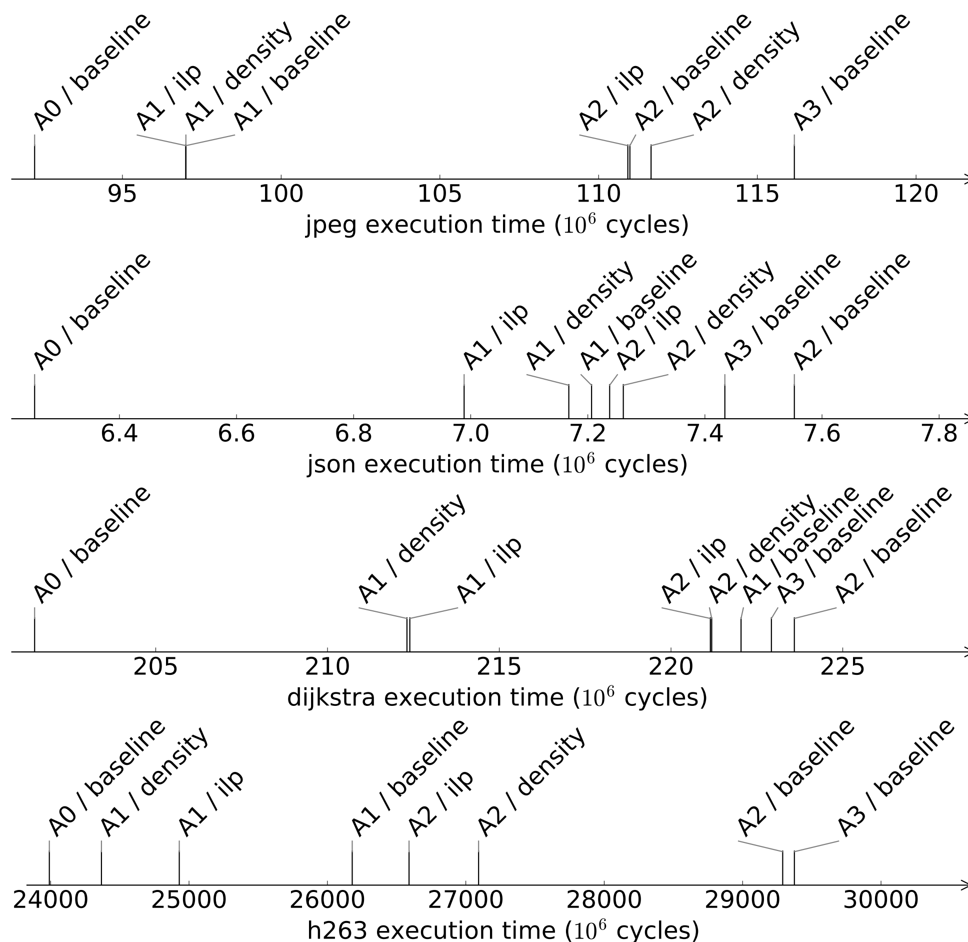

FIGURE 3 – temps d'exécution pour les différentes architectures / stratégies par application

Bibliographie

1. Ait Aoudia (F.), Marquet (K.) et Salagnac (G.). – Incremental checkpointing of program state to NVRAM for transiently-powered systems. – In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014 9th International Symposium on*, pp. 1–4. IEEE, 2014.
2. Angiolini (F.), Benini (L.) et Caprara (A.). – Polynomial-time algorithm for on-chip scratchpad memory partitioning. – In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pp. 318–326. ACM, 2003.

3. Avissar (O.), Barua (R.) et Stewart (D.). – An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 1, n1, 2002.
4. Bailey (K.), Ceze (L.), Gribble (S. D.) et Levy (H. M.). – Operating system implications of fast, cheap, non-volatile memory. – In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, pp. 2–2, Berkeley, CA, USA, 2011. USENIX Association.
5. Dominguez (A.), Udayakumaran (S.) et Barua (R.). – Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, vol. 1, n4, 2005, pp. 521–540.
6. Egger (B.), Kim (S.), Jang (C.), Lee (J.), Min (S. L.) et Shin (H.). – Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU. *IEEE Transactions on Computers*, vol. 59, n8, Aug 2010, p. 1047–1062.
7. Fritts (J. E.), Steiling (F. W.) et Tucek (J. A.). – Mediabench II video : expediting the next generation of video systems research. *Microprocessors and Microsystems*, vol. 33, 2009.
8. Gabis (K.). – Parson : Lightweight JSON library written in C. – https ://github.com/kgabis/parson.
9. Guthaus (M. R.), Ringenberg (J. S.), Ernst (D.), Austin (T. M.), Mudge (T.) et Brown (R. B.). – Mibench : A free, commercially representative embedded benchmark suite. – In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001.
10. Hu (J.), Xue (C. J.), Zhuge (Q.), Tseng (W.-C.) et Sha (E. H.-M.). – Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. – In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6. IEEE, 2011.
11. Hu (J.), Xue (C. J.), Zhuge (Q.), Tseng (W.-C.) et Sha (E. H.-M.). – Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, n6, Jun 2013, p. 1094–1102.
12. Hu (J.), Zhuge (Q.), Xue (C. J.), Tseng (W.-C.) et Sha (E. H.-M.). – Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors. *ACM Transactions on Embedded Computing Systems*, vol. 13, n4, Mar 2014, p. 1–25.
13. Jayakumar (H.), Lee (K.), Lee (W. S.), Raha (A.), Kim (Y.) et Raghunathan (V.). – Powering the Internet of Things. – In *Low Power Electronics and Design (ISLPED), 2014 IEEE/ACM International Symposium on*, pp. 375–380, Aug 2014.
14. Kandemir (M.), Ramanujam (J.), Irwin (M. J.), Vijaykrishnan (N.), Kadayif (I.) et Parikh (A.). – Dynamic management of scratch-pad memory space. – In *Design Automation Conference, 2001. Proceedings*, pp. 690–695. IEEE, 2001.
15. Kultursay (E.), Kandemir (M.), Sivasubramaniam (A.) et Mutlu (O.). – Evaluating STT-RAM as an energy-efficient main memory alternative. – In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 256–267. IEEE, 2013.
16. Layer (C.), Javerliac (V.), Bernard-Granger (F.), Decloedt (L.), Becker (L.), Jabeur (K.), Claireux (S.), Dieny (B.), Prenat (G.), Pendina (G. D.) et et al. – Reducing System Power Consumption Using Check-Pointing on Nonvolatile Embedded Magnetic Random Access Memories. *ACM Journal on Emerging Technologies in Computing Systems*, vol. 12, n4, May 2016, p. 1–24.
17. Lea (D.). – A memory allocator. – `http://g.oswego.edu/dl/html/malloc.html`, 2012.
18. Meena (J. S.), Sze (S. M.), Chand (U.) et Tseng (T.-Y.). – Overview of emerging nonvolatile memory technologies. *Nanoscale research letters*, vol. 9, n1, 2014, p. 526.
19. Mück (T. R.) et Fröhlich (A. A.). – Run-time scratch-pad memory management for embedded systems. – In *37th Annual Conference on IEEE Industrial Electronics Society*, pp. 2833–2838. IEEE, 2011.
20. Nakamura (H.), Nakada (T.) et Miwa (S.). – Normally-off computing project : Challenges and opportunities. – In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014.
21. Rodríguez (G.), Touriño (J.) et Kandemir (M. T.). – Volatile STT-RAM Scratchpad Design and Data Allocation for Low Energy. *ACM Transactions on Architecture and Code Optimization*, vol. 11, n4, 2014.
22. Shrivastava (A.), Kannan (A.) et Lee (J.). – A software-only solution to use scratch pads for stack data. *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 28, n11, 2009.
23. SOCLIB Consortium. – Projet SOCLIB : Plate-forme de modélisation et de simulation de systèmes integrés sur puce. Technical report, CNRS, 2003.
24. Yang (J. J.), Strukov (D. B.) et Stewart (D. R.). – Memristive devices for computing. *Nature Nanotechnology*, vol. 8, n1, Dec 2012, p. 13–24.