



Gestion de la mémoire dynamique pour les systèmes embarqués avec mémoire hétérogène

Tristan Delizy

► To cite this version:

Tristan Delizy. Gestion de la mémoire dynamique pour les systèmes embarqués avec mémoire hétérogène. Informatique [cs]. Insa Lyon, 2019. Français. tel-02429017v1

HAL Id: tel-02429017

<https://hal.archives-ouvertes.fr/tel-02429017v1>

Submitted on 6 Jan 2020 (v1), last revised 10 Sep 2020 (v2)

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.



INSA

N°d'ordre NNT : 2019LYSEI134

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de
INSA de Lyon

École Doctorale 512
InfoMaths

Spécialité Informatique

Soutenue publiquement le 19/12/2019, par :
Tristan Delizy

**Gestion de la mémoire dynamique pour
les systèmes embarqués avec mémoire
hétérogène**

Devant le jury composé de :

Sentieys, Olivier - President
Professeur des Universités, Université de Rennes

Belleudy, Cécile - Rapporteur
Maitre de Conférences HDR, Université Nice Sophia Antipolis

Torres, Lionel - Rapporteur
Professeur des Universités, Université Montpellier

Salagnac, Guillaume - Examinateur
Maitre de Conférences, INSA de Lyon

Risset, Tanguy - Directeur de thèse
Professeur des Universités INSA de Lyon

Moy, Matthieu - Co-directeur de thèse
Maitre de Conférences HDR Université Claude Bernard Lyon 1

Département FEDORA – INSA Lyon - Ecoles Doctorales – Quinquennal 2016-2020

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	CHIMIE DE LYON http://www.edchimie-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage secretariat@edchimie-lyon.fr INSA : R. GOURDON	M. Stéphane DANIELE Institut de recherches sur la catalyse et l'environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 Avenue Albert EINSTEIN 69 626 Villeurbanne CEDEX directeur@edchimie-lyon.fr
E.E.A.	ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE http://edeea.ec-lyon.fr Sec. : M.C. HAVGOUDOUKIAN ecole-doctorale.eea@ec-lyon.fr	M. Gérard SCORLETTI École Centrale de Lyon 36 Avenue Guy DE COLLONGUE 69 134 Écully Tél : 04.72.18.60.97 Fax 04.78.43.37.17 gerard.scorletti@ec-lyon.fr
E2M2	ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION http://e2m2.universite-lyon.fr Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : H. CHARLES secretariat.e2m2@univ-lyon1.fr	M. Philippe NORMAND UMR 5557 Lab. d'Ecologie Microbienne Université Claude Bernard Lyon 1 Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69 622 Villeurbanne CEDEX philippe.normand@univ-lyon1.fr
EDISS	INTERDISCIPLINAIRE SCIENCES-SANTÉ http://www.ediss-lyon.fr Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : M. LAGARDE secretariat.ediss@univ-lyon1.fr	Mme Emmanuelle CANET-SOULAS INSERM U1060, CarMeN lab, Univ. Lyon 1 Bâtiment IMBL 11 Avenue Jean CAPELLE INSA de Lyon 69 621 Villeurbanne Tél : 04.72.68.49.09 Fax : 04.72.68.49.16 emmanuelle.canet@univ-lyon1.fr
INFOMATHS	INFORMATIQUE ET MATHÉMATIQUES http://edinfomaths.universite-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 infomaths@univ-lyon1.fr	M. Luca ZAMBONI Bât. Braconnier 43 Boulevard du 11 novembre 1918 69 622 Villeurbanne CEDEX Tél : 04.26.23.45.52 zamboni@maths.univ-lyon1.fr
Matériaux	MATÉRIAUX DE LYON http://ed34.universite-lyon.fr Sec. : Stéphanie CAUVIN Tél : 04.72.43.71.70 Bât. Direction ed.materiaux@insa-lyon.fr	M. Jean-Yves BUFFIÈRE INSA de Lyon MATEIS - Bât. Saint-Exupéry 7 Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX Tél : 04.72.43.71.70 Fax : 04.72.43.85.28 jean-yves.buffiere@insa-lyon.fr
MEGA	MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE http://edmega.universite-lyon.fr Sec. : Stéphanie CAUVIN Tél : 04.72.43.71.70 Bât. Direction mega@insa-lyon.fr	M. Jocelyn BONJOUR INSA de Lyon Laboratoire CETHIL Bâtiment Sadi-Carnot 9, rue de la Physique 69 621 Villeurbanne CEDEX jocelyn.bonjour@insa-lyon.fr
ScSo	ScSo* http://ed483.univ-lyon2.fr Sec. : Véronique GUICHARD INSA : J.Y. TOUSSAINT Tél : 04.78.69.72.76 veronique.cervantes@univ-lyon2.fr	M. Christian MONTES Université Lyon 2 86 Rue Pasteur 69 365 Lyon CEDEX 07 christian.montes@univ-lyon2.fr

*ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Résumé

La réduction de la consommation énergétique des systèmes embarqué est un enjeu majeur de la réalisation de l'Internet des Objets. Les mémoires émergentes NVRAMs présentent notamment le potentiel de consommer peu et d'être denses, mais les différentes technologies souffrent encore de désavantages spécifiques comme une latence d'écriture élevée ou une faible endurance. Pour contrebalancer ces désavantages, les concepteurs de systèmes embarqués tendent à juxtaposer différentes technologies sur une même puce. Cette thèse s'intéresse aux interactions entre l'allocation mémoire dynamique et l'hétérogénéité mémoire. Notre objectif est de fournir au programmeur d'applications embarquées un mécanisme logiciel transparent pour exploiter cette hétérogénéité mémoire. Nous proposons un simulateur au cycle près de plateformes embarquées intégrant des technologies mémoire variées qui montre que les stratégies de placement des objets alloués dynamiquement ont un impact important. Nous montrons également que des gains intéressants peuvent être dégagés même avec une faible proportion de la mémoire utilisant une technologie à faible latence mais uniquement en utilisant une stratégie intelligente pour le placement entre les différentes banques mémoires. Nous fournissons une stratégie efficace basée sur le profilage de l'application dans notre simulateur.

Abstract

Reducing energy consumption is a key challenge to the realisation of the Internet of Things. While emerging memory technologies may offer power reduction and high integration density, they come with major drawbacks such as high latency or limited endurance. As a result, system designers tend to juxtapose several memory technologies on the same chip. We aim to provide the embedded application programmer with a transparent software mechanism to leverage this memory heterogeneity. This work studies the interaction between dynamic memory allocation and memory heterogeneity. We provide cycle accurate simulation of embedded platforms with various memory technologies and we show that different dynamic allocation strategies have a major impact on performance. We demonstrate that interesting performance gains can be achieved even for a low fraction of memory using low latency technology, but only with a clever placement strategy between memory banks. We propose an efficient strategy based on application profiling in our simulator.

Remerciements

Je tiens à remercier tout ceux qui, durant cette thèse ont été à mes cotés, qu'il s'agisse de mes collègues, de ma famille ou de mes amis.

Un grand merci, bien évidemment, à mon encadrement, qui au delà de me donner l'opportunité de me lancer dans cette aventure a été présent tout du long pour m'apprendre quantités de choses. Le nombreuses discussions que nous avons partagées ont été pour moi une constante source d'idées et de motivations.

Merci aussi à Stéphane Gros, grâce à qui nous avons pu mieux comprendre les mémoires MRAM, et la manière de les utiliser. Merci à tout les collègues du laboratoire, pour l'aspect collégial et les discussions scientifiques, mais aussi pour le reste du temps que nous avons partagé. Merci à mes amis et à ma famille qui m'ont aidé à continuer d'avancer et à faire baisser la pression quand c'était nécessaire.

Et pour finir, merci aux plantes de mon jardin pour la sérénité qu'elles m'ont apporté en les regardant pousser.

Table des matières

Résumé	5
Remerciements	7
Table des matières	9
I Introduction et contexte	11
1 Introduction	13
1.1 Contexte	13
1.2 Mémoires non-volatiles émergentes pour l'embarqué	13
1.3 Hétérogénéité mémoire et mécanismes logiciels	14
1.4 Contributions	15
1.5 Organisation du document	16
1.6 Financement	18
2 Gestion mémoire dynamique	19
2.1 Allocation mémoire	20
2.2 Allocation mémoire dynamique manuelle	22
2.3 Problème de fragmentation	23
2.4 Mémoire hétérogène et allocation dynamique	24
2.5 Conclusion	27
3 Systèmes embarqués et IOT	29
3.1 Architectures embarquées	31
3.2 Programmes embarqués	32
3.3 Conclusion	34
4 Technologies mémoire pour l'embarqué	37
4.1 Mémopire de travail et mémoire de stockage	37
4.2 Mémoire idéale	39
4.3 Les technologies de NVRAM	40
4.4 Conclusion	44
5 Mécanismes logiciels pour la gestion de mémoire hétérogène	47
5.1 Hétérogénéité mémoire : systèmes cibles	47
5.2 Mécanismes logiciels : classification et caractéristiques	49
5.3 L'hétérogénéité mémoire dans le domaine de l'embarqué	50
5.4 Allocation du tas en mémoire Scratch Pad	52
5.5 Travaux connexes	55
5.6 Conclusion	57
II Contributions	59
6 Positionnement et approche	61

6.1	Problème de placement pour le tas	61
6.2	Cas d'étude	63
6.3	Méthodologie d'évaluation de la gestion mémoire dynamique	70
6.4	Couche logicielle d'allocation multi-tas	72
6.5	Architectures mémoire hétérogènes pour le tas	73
6.6	Conclusion	74
7	Étude expérimentale du comportement des applications	75
7.1	Infrastructure logicielle nécessaire à l'étude	75
7.2	Cas d'étude logiciels	79
7.3	Conception d'une métrique efficace : Fréquence d'accès par octet	89
7.4	Comportement d'allocation des applications	90
7.5	Conclusion	100
8	Impact de la stratégie de placement sur les performances	101
8.1	Stratégie baseline en ligne : Fast First	101
8.2	Résolution hors ligne	104
8.3	Stratégie hors ligne optimale	105
8.4	Stratégie hors ligne "fréquence d'accès par octet" : évaluation de la métrique	113
9	Conception et implémentation d'une stratégie efficace	117
9.1	Profilage des sites d'allocation d'une application	117
9.2	Méthodologie	118
9.3	Évaluation	121
9.4	Limitations de la stratégie proposée et Perspectives	132
9.5	Conclusion	136
10	Bilan et perspectives	139
10.1	Bilan de la thèse	139
10.2	Perspectives	140
	Références Bibliographiques	143
	III Annexes	151
A	Description des jeux de données	153
B	Analyse des objets par exécution	155
C	Résultats en performance par jeu de données	197
	Index	203

Première partie

Introduction et contexte

Chapitre 1

Introduction

1.1 Contexte

Le développement et l'expansion de l'informatique dans notre société a pris une importance décisive sur les dernières décennies. Mais dans cette tendance, les projections promettent une intégration encore plus poussée de l'informatique dans tous les aspects de la vie quotidienne. Ces projections reposent notamment sur le développement de l'Internet des Objets (IoT). Ces objets, hier isolés et considérés en eux-mêmes se retrouvent connectés dans des buts de mutualisation, d'automatisation et de collecte d'information. Ainsi, un thermomètre connecté ne se contente plus d'afficher la température, mais stocke cette donnée, peut la communiquer à un système domotique gérant le chauffage ou un service en ligne pour améliorer un modèle de prévision locale ou adapter les publicités diffusées à la météo. L'augmentation des fonctionnalités de ces objets repose sur des systèmes embarqués communicants et ouvre la porte à de nombreuses innovations. Ces innovations concernent le domaine de l'e-santé, de l'automobile, de la domotique ou de l'industrie, entre autres. L'explosion de l'Internet des Objets peut notamment être représentée par l'augmentation prévue du nombre de ces objets connectés : 50 milliards en 2020 [JLLRKR14].

Dans le domaine de l'embarqué constraint, ici, sans caches, cette expansion est centrale dans les orientations actuelles, tant pour la recherche académique que sur le plan industriel. Pour l'industrie les défis sont nombreux en termes de production des objets connectés, d'infrastructures et de technologies pour supporter leurs communications. Pour la recherche académique l'augmentation du nombre de ces appareils augmente l'intérêt de tout développement permettant de réduire leur impact énergétique et environnemental. Bien évidemment, la conception de ces appareils en eux-mêmes ainsi que de leur environnement logiciel (protocoles de communication, middleware et big data) présente de nombreux challenges. Toutefois nos travaux sont centrés sur les objets connectés en eux-mêmes : des appareils embarqués contraints en ressources de calcul et de stockage.

La consommation énergétique des systèmes embarqués est une considération centrale du domaine, notamment depuis que ceux-ci peuvent fonctionner sur batterie. Toutefois, elle connaît ces dernières années un développement significatif impulsé par la recherche et le développement de nouvelles technologies mémoire.

1.2 Mémoires non-volatiles émergentes pour l'embarqué

Ces nouvelles technologies mémoire sont regroupées sous le nom de ENVM (Emerging Non-Volatile Memory – technologies mémoires non-volatiles émergentes). Plus spécifiquement, dans ces travaux nous parlerons des NVRAMs (Non-Volatile Random Access Memory – mémoire non-volatile à accès aléatoire). En effet nous étudions l'utilisation de ces technologies comme mémoire de travail, nécessitant un accès aléatoire à l'octet. Ces technologies exploitent de nombreux effets physiques pour permettre le stockage de l'information. Elles ont pour point commun de la conserver sans nécessiter de rafraîchissement ou d'alimentation électrique hors opération. Elles sont également plus rapides et plus denses que la mémoire flash. Ainsi il est possible de les utiliser en tant que mémoire de travail.

L'utilisation de ces technologies dans le domaine de l'embarqué constraint semble prometteuse et de nombreuses architectures matérielles intègrent ces nouvelles technologies. Ces évolutions

des architectures matérielles sont amenées à changer les architectures mémoires. Ce changement entraîne également une évolution dans l'utilisation logicielle qui en est faite.

En effet l'usage classique de technologies mémoire aux performances différentes a donné lieu à des hiérarchies mémoires. Cette approche permet de limiter l'impact d'une technologie mémoire plus lente que le processeur. En effet les fréquences des processeurs ont progressé plus rapidement que celles des technologies mémoire, conduisant au problème du mur mémoire [WM95]. Ainsi la latence élevée d'une mémoire peut être masquée par l'usage d'un cache. Il s'agit une mémoire plus petite mais plus rapide contenant une partie des données qui peut donc être accédé avec une faible latence. La cohérence et la disponibilité des données sont gérées par le cache, au niveau matériel. Mais de par les caractéristiques que ces nouvelles technologies présentent, et de par leur variété, le domaine de l'embarqué voit se développer de nombreuses architectures mémoires ne correspondant pas à l'approche hiérarchique. En effet dans ces technologies, certaines consomment peu, d'autre beaucoup mais uniquement en lecture. Elles présentent également des temps d'écriture et de lectures très variables, dont certains du même ordre de grandeur que la mémoire de travail de ces systèmes. Il est également à noter que ces technologies peuvent présenter d'autres avantages ou inconvénients liés à leur endurance, à leur durée de rétention ou à la facilité d'intégration aux processus de production des systèmes sur puces.

L'architecture mémoire contenant les même éléments mémoires peut en effet être organisée différemment. La circuiterie de contrôle des caches occupe en effet beaucoup de place dans un système sur puce, et sa consommation n'est pas négligeable. Dans ce cas, le processeur accède directement à la mémoire et celle ci n'est pas uniforme car composée de différentes technologies [LJBGD+16]. Nous parlons dans ce cas d'hétérogénéité mémoire.

C'est dans ce contexte que la start-up eVaderis, partenaire industriel de cette thèse, a proposé des modules mémoires paramétrables basés sur des technologies MRAM. Cette start-up a depuis cessé ces activités suite à l'échec d'une levée de fonds pendant la phase d'industrialisation du projet. Ce partenariat nous a néanmoins fourni des exemples concrets de technologies mémoires, de leur implantations possibles et de leur performances sur lesquels nous basons nos travaux.

1.3 Hétérogénéité mémoire et mécanismes logiciels

Les travaux présentés dans cette thèse concernent l'utilisation logicielle de telles architectures mémoire. En effet dans le domaine de l'embarqué il a été démontré qu'une approche basée sur l'hétérogénéité mémoire, alliée à des mécanismes logiciels permet une plus grande efficience énergétique des systèmes que les caches matériels [BSLM02 ; KICK04].

Ces études ont été réalisées sur un exemple d'hétérogénéité mémoire répandu, les mémoires Scratch Pad, où l'architecture mémoire comporte une mémoire lente de taille conséquente et une mémoire rapide d'une taille limitée, les deux directement accessibles par le processeur. Les pistes de recherche sur ces architectures ont été explorées avant l'émergence des technologies mémoire évoquées plus tôt. Toutefois les NVRAMs permettent l'exploration de nouvelles pistes pour les mécanismes logiciels nécessaires à l'utilisation d'architectures mémoires hétérogènes.

Les mécanismes logiciels mis en place varient en fonction de quelle partie du programme est exposée à l'hétérogénéité mémoire. Ainsi ils seront différents pour positionner du code ou des variables globales, des variables locales ou des blocs mémoires alloués dynamiquement.

Nous avons focalisé nos travaux sur l'allocation dynamique dans le contexte de l'hétérogénéité mémoire. En effet, la littérature a déjà proposé des mécanismes logiciels efficaces pour l'allocation du code et des données en mémoire hétérogène, contrairement à l'allocation mémoire dynamique.

Ce choix est également motivé par le développement actuel de l'IoT. En effet, la nécessité pour les objets d'être communicants les rend vulnérables et impose de pouvoir les mettre à jour. Il est également nécessaire de disposer de mécanismes de sécurité poussés, ne pouvant pas toujours être implémentés purement en matériel. Ainsi, que ce soit pour améliorer les performances du système, pour l'adapter à l'évolution de son environnement ou pour la correction de bugs de sécurité, la présence d'un canal de mise à jour est importante. Dans le même temps les logiciels embarqués se diversifient, et doivent être capables d'évoluer, augmentant l'intérêt ou nécessitant l'utilisation de l'allocation mémoire dynamique.

Nous présentons des mécanismes logiciels de gestion du tas permettant de tirer parti de l'hétérogénéité mémoire présentée par les NVRAMs pour l'embarqué. Notre étude porte sur la

réduction du temps d'exécution de l'application. Nous considérons donc l'hétérogénéité mémoire de technologies présentant des différences de latences d'accès. Nous étudions ainsi le placement des objets alloués dynamiquement entre une mémoire rapide et une mémoire lente. Nous soutenons que cette approche permet de significativement améliorer les performances du système. En effet les programmes exhibent des régularités dans leurs accès mémoires que nous pouvons exploiter. Ainsi, ajouter une faible proportion de mémoire plus rapide améliore significativement les performances par rapport à une exécution en mémoire lente. Mais il est nécessaire de choisir les bons objets à placer dans le tas rapide.

1.3.1 Allocation mémoire dynamique et placement

L'allocation mémoire dynamique permet de répondre à des besoins mémoire arbitraires de l'application, qui sont inconnus avant l'exécution car ils dépendent du jeu de données. Ainsi l'ordre, la taille ou le nombre de requêtes n'est pas connu à l'avance, ce qui complexifie l'allocation de ces blocs mémoire.

Le problème que nous abordons est de décider dans quelle mémoire placer un objet. Toutefois nous considérons ce problème séparément de celui de l'allocation mémoire dynamique à l'intérieur du tas. Nous choisissons donc dans notre approche de réutiliser une allocateur mémoire dynamique de la littérature. En effet la résolution du problème de fragmentation est un domaine largement exploré. Pour ce faire, nous instancions un allocateur mémoire par tas considéré. Nous prenons la décision de placement entre mémoire rapide et lente dans un module logiciel dédié, le dispatcheur. Ce module transmet par la suite cette requête à l'allocateur mémoire dynamique du tas de destination.

En effet au delà de l'hétérogénéité des technologies mémoire, les objets ainsi alloués par l'application ne sont pas accédés d'une manière uniforme. Ainsi des objets beaucoup plus accédés que la moyenne placés en mémoire rapide auront un impact positif sur les performances de l'application. Au contraire, ces mêmes objets placés en mémoire lente dégradent les performances de l'application.

C'est donc le choix pris par le dispatcheur qui va apporter une solution au problème de placement.

Cette décision, répétée sur l'ensemble des requêtes d'allocation dynamique de l'exécution d'une application aura un impact sur les performances. Il est important de noter que l'influence de telles décisions dépend des technologies mémoire impliquées et de l'application. En effet les performances de ces technologies définissent l'intervalle de performance que peut atteindre l'application.

Mais différentes applications n'exposent pas la même hétérogénéité dans la manière qu'elles ont d'accéder à leur tas. Ainsi une autre question que notre approche permet d'étudier est de savoir de quelle proportion de mémoire rapide une application a besoin pour améliorer significativement ses performances.

1.4 Contributions

L'objectif de ces travaux est de proposer un mécanisme logiciel permettant de tirer parti de l'hétérogénéité mémoire pour l'allocation des objets du tas. Nous réalisons cette étude pour le domaine de l'embarqué contraint, en nous basant sur les technologies mémoires émergentes. Nous proposons donc une architecture logicielle séparant la résolution du problème de placement de l'allocation mémoire dynamique. Le module que nous introduisons est appelé dispatcheur. Basé sur cette architecture nous étudions les stratégies applicables par le dispatcheur pour évaluer les gains en performances qu'elles peuvent dégager.

Un autre objectif de ces travaux est de proposer une méthode de résolution cohérente vis à vis des pratiques de l'embarqué. Nous choisissons ainsi de proposer une résolution n'impliquant pas le programmeur d'applications embarquées. En effet de même que les détails de la gestion mémoire dynamique sont abstraits par un gestionnaire mémoire, nous ne souhaitons pas laisser à la charge de chaque développeur le soin de résoudre le problème de placement.

De plus, la résolution du problème de placement dépend de la cible matérielle (technologies mémoire, espace disponible). Ainsi une résolution dans le code de l'application alourdirait significativement le travail de portage vers une cible matérielle différente.

Nous avons donc fait le choix de respecter l'API standard d'allocation mémoire `malloc()`/`free()`. Outre l'abstraction de l'hétérogénéité mémoire du tas au programmeur, ce choix permet de ne pas avoir à modifier le code applicatif. C'est un avantage dans le domaine de l'embarqué où le code source, notamment de librairies tierces, n'est pas toujours accessible.

Nous allons maintenant expliciter les contributions de nos travaux. Tout d'abord nous proposons un simulateur de plateforme embarquée permettant d'évaluer l'impact de l'hétérogénéité mémoire. Nous proposons également une méthodologie d'étude du placement mémoire des objets du tas en mémoire hétérogène. En effet, dans nos conditions, rejouer une application avec le même jeu de données d'entrée génère une exécution équivalente. Basé sur cette méthodologie, nous évaluons une stratégie naïve. Celle-ci essaye de maximiser l'utilisation de la mémoire rapide en essayant d'allouer dans le tas rapide en priorité. Cette stratégie nécessite la mise en place d'un mécanisme de redirection si le tas rapide est plein. Nous appelons mécanisme de fallback cette redirection.

Nous proposons par la suite une formulation du problème de placement sous la forme d'un programme linéaire en nombres entiers permettant sa résolution à posteriori. Cette résolution nous permet d'évaluer une sous-approximation et une sur-approximation des performances optimales. Nous pouvons ainsi borner les gains en performance atteignables pour une exécution particulière sur une architecture mémoire donnée.

Ces résultats démontrent qu'un placement optimal améliore significativement les performances. Qui plus est, il est possible d'atteindre des performances proches de celles d'un tas composé uniquement de mémoire rapide en ne disposant que de 5% ou 10% de mémoire rapide. Il est à noter que la stratégie naïve que nous avons évaluée ne permet pas du tout d'atteindre ces performances. Ainsi nous pouvons affirmer qu'une stratégie de placement intelligente est nécessaire pour améliorer significativement les performances d'une application.

Nous étudions ensuite le comportement de nos applications de manière à élaborer une stratégie efficace. Pour la mener à bien, nous avons développé et évalué une métrique permettant de calculer l'importance d'un objet pour la résolution du problème de placement : la fréquence d'accès par octet. De cette étude nous déduisons que l'information du site d'allocation est intéressante pour une stratégie de résolution durant l'exécution. Nous proposons donc une stratégie basée sur une phase de profilage et l'utilisation du site d'allocation, accessible facilement à l'exécution. Cette stratégie est de plus résiliente aux variations dans l'allocation mémoire dynamique dépendant du jeu de donnée d'entrée. Elle permet d'atteindre de meilleurs résultats que la stratégie naïve, et obtient pour une partie des applications évaluées des résultats proches de l'optimal.

1.5 Organisation du document

Le chapitre 2 donne un aperçu des mécanismes logiciels d'allocation mémoire utilisés dans le domaine de l'embarqué constraint. Nous focalisons notre propos sur les solutions applicables et appliquées pour ces systèmes. Nous présentons dans ce chapitre l'allocateur mémoire dynamique de la littérature que nous avons utilisé : le DLmalloc [LG96]. La fin du chapitre discute les interactions entre ces méthodes d'allocation et un système mémoire hétérogène. En effet la méthodologie classique d'évaluation de l'allocation mémoire dynamique n'est pas adaptée à l'hétérogénéité mémoire que nous considérons. Nous décrivons donc une méthodologie pour l'évaluation du problème de placement différente de l'évaluation classique d'un allocateur mémoire dynamique. En effet la résolution du problème de placement répartit les requêtes d'allocations entre les différents tas. Les performances de l'allocation mémoire dynamique varient donc en fonction du placement. Nous rejouons donc entièrement l'exécution d'un programme pour mesurer l'influence du placement mémoire sans négliger son impact sur l'allocation mémoire dynamique.

Le chapitre 3 présente notre domaine d'application : l'embarqué constraint. Nous nous intéressons à des plateformes ne disposant ni d'unité de gestion de la mémoire (MMU), ni de caches. En effet, nous considérons des plateformes contraintes en énergie et ces modules en sont généralement exclus. Nos travaux concernent les mécanismes logiciels de ces plateformes. Nous décrivons donc le fonctionnement logiciel de ces plateformes en fonction des architectures matérielles du domaine. L'absence d'un cache change drastiquement les accès effectivement réalisés dans la mémoire, le rôle d'un cache matériel étant d'en masquer le plus possible. L'absence de MMU interdit quant à

elle l'abstraction de la mémoire virtuelle. Il n'est donc pas possible de facilement déplacer un objet du tas après son allocation.

Le chapitre 4 présente les technologies de NVRAMs ou ENVM. Nous y discutons des différentes technologies émergentes et de leurs propriétés. Nous détaillons également dans ce chapitre les deux scénarios technologiques ainsi que les différentes architectures mémoire que nous étudions dans la suite. Suite aux discussions avec notre partenaire industriel, nous étudions un scénario juxtaposant une technologie MRAM dense à une technologie de MRAM rapide. Nous évaluons également un scénario basé sur une technologie mémoire ReRAM proposée par Intel. Dans ce scénario, la ReRAM, plus dense et très lente en écriture, est opposée à un banc mémoire en SRAM. Partant de ces scénarios technologiques nous proposons d'étudier la résolution du problème de placement sur différentes architectures mémoire. Nous explorons donc différentes répartitions du tas entre mémoire lente et de mémoire rapide.

Le chapitre 5 présente un état de l'art des mécanismes logiciels d'allocation en mémoire hétérogène. Nous présentons dans un premier temps les architectures mémoires que nous qualifions d'hétérogènes, puis nous nous intéressons aux mécanismes logiciels permettant leur allocation. Cet état de l'art concerne les mécanismes applicables au domaine de l'embarqué contraint. Nous présentons les mécanismes utilisés pour l'allocation du code, des données statiques, de la pile et pour finir du tas. Nous finissons ce chapitre par l'élargissement vers l'allocation du tas en mémoire hétérogène dans des domaines dont les capacités matérielles sont plus importantes : calcul haute performance et NUMA.

Le chapitre 6 expose le problème de placement des objets du tas. Nous restreignons donc le domaine d'application et explicitons les hypothèses de nos travaux. Ainsi nous proposons une résolution du problème de placement considérant comme performance le temps d'exécution de l'application. Nous considérons également un cas d'hétérogénéité mémoire à deux technologies mémoire, une rapide et une lente. Ces hypothèses correspondent à l'approche classique de l'hétérogénéité mémoire de la littérature [WCLL18]. Ce chapitre présente également notre méthodologie d'évaluation des solutions au problème de placement. Nous considérons que l'évaluation d'un placement mémoire ne peut pas être réalisée à posteriori. Nous choisissons donc de ré-exécuter l'application pour évaluer un placement. En effet les performances de l'allocateur mémoire dynamique dépendent des décisions de placement.

Le chapitre 7 débute par une présentation de l'infrastructure logicielle nécessaire à notre étude. Elle précise donc nos choix concernant la simulation. Nous modélisons le processeur au cycle près, ainsi que les modules mémoire, mais acceptons une couche d'abstraction des communications entre modules (SystemC TLM). Nous obtenons ainsi une simulation suffisamment précise pour évaluer le problème de placement. De plus, celle-ci reste raisonnable en temps quant à la simulation de l'exécution d'une application complexe allouant un grand nombre d'objets. Nous y détaillons aussi les aspects logiciels de l'allocation mémoire multi-tas. La suite du chapitre concerne le choix des applications étudiées, ainsi que l'analyse des objets du tas de ces applications suivant des critères de taille et de site d'allocation. En effet ces deux informations sont accessibles sans surcoût à l'exécution. Nous en déduisons que l'information de taille n'est pas utilisable en soi. Au contraire, on peut facilement distinguer que certains sites d'allocation allouent en moyenne des objets chauds (c'est à dire très accédés et donc importants pour le problème de placement). De même certains sites d'allocation n'allouent en moyenne que des objets froids. Nous proposons donc de nous baser sur cette information pour l'élaboration d'une stratégie efficace.

Le chapitre 8 présente une stratégie naïve de résolution du problème et une étude des gains atteignables. Nous proposons une formulation du problème sous forme d'un programme linéaire en nombres entiers dans cette optique et bornons l'optimal des performances atteignables. Cette étude nous permet d'affirmer que la résolution du problème de placement permet d'améliorer significativement les performances de nos applications. De plus, dans de nombreux cas une faible fraction du tas en mémoire rapide (5% ou 10%) permet d'atteindre ces gains en performances. Toutefois, nous constatons aussi que la stratégie naïve n'arrive pas à de bons résultats avec moins de 50% ou 75% de mémoire rapide. Partant de ces résultats nous construisons une métrique

d'évaluation à posteriori de la chaleur des objets. Pour valider l'intérêt de cette métrique nous l'utilisons pour résoudre hors ligne le problème de placement et évaluons la stratégie résultante. Cette évaluation présente dans la majorité des cas une amélioration des performances proche de l'optimal.

Le chapitre 9 présente une stratégie en ligne basée sur le profilage des sites d'allocation. Nous détaillons la méthodologie mise en œuvre pour l'évaluation de l'intérêt d'un site d'allocation. Nous nous basons sur la métrique proposée au chapitre 8. Nous évaluons la chaleur moyenne des objets alloués par chaque site d'allocation sur un ensemble d'exécutions de profilage. Nous agrégeons par la suite les informations des différentes exécutions de profilage et obtenons ainsi un classement des sites d'allocation de l'application. Nous explorons combien de sites sélectionner dans cette liste et évaluons la stratégie sur un autre ensemble de jeux de données. Nous étudions également l'influence du jeu de données ainsi que la résilience de la stratégie aux variations induites par l'utilisation de différents jeux de données.

Finalement, le chapitre 10 fait le bilan de ces travaux et expose les perspectives de recherches sur la résolution du problème de placement en mémoire hétérogène dans le domaine de l'embarqué constraint. Nous pensons que les conclusions de ces travaux peuvent être étendus à une hétérogénéité mémoire plus complexe : plus de mémoires différentes, plus de critères d'évaluation. Nous pensons également qu'il est possible d'améliorer les stratégies présentées dans ces travaux, notamment par l'utilisation de compteurs matériels pour disposer d'un profilage partiel durant l'exécution.

1.6 Financement

Cette thèse est financée par une allocation doctorale de recherche de la Région Auvergne-Rhône-Alpes et a reçu le support de la chaire IoT Insa-Spie.

Chapitre 2

Gestion mémoire dynamique

Pour son exécution, un programme informatique à besoin de mémoire. Cette mémoire peut être utilisée pour stocker le programme en lui même, ses entrées, ses résultats, ou pour stocker les calculs qu'ils doit effectuer. Les systèmes informatiques ont rapidement du faire face durant leur développement au problème du mur mémoire : les technologies pour le calcul ont progressé plus vite que celles mises en œuvre pour la mémoire. Ainsi la latence mémoire relative à la vitesse du processeur a augmenté pour l'accès à la mémoire.

Ce problème a entraîné la différentiation entre mémoire de travail et mémoire de stockage, basées sur des technologies différentes. La *mémoire de stockage* peut contenir beaucoup plus de données que nécessaire pour l'exécution d'un programme mais est beaucoup plus lente. La *mémoire de travail* est utilisée par le processeur pour l'exécution des programmes.

Dépendant du système informatique et de son domaine d'application, les technologies et les caractéristiques de ces mémoires varient. Néanmoins, la mémoire de travail est plus rapide et peut être accédée aléatoirement, au contraire de la mémoire de stockage, beaucoup plus lente et en accès séquentiel, ou par bloc. De plus, une plateforme dispose en général de beaucoup plus de mémoire de stockage que de travail.

Nous nous intéressons ici à la mémoire de travail, celle utilisée par le programme durant son exécution. Cette mémoire va être utilisée de différentes manières pour différents usages. Toutefois, le programme ne gère normalement pas lui même la mémoire qu'il utilise. Ce cas de figure peut néanmoins être rencontré dans le cas de l'embarqué. Au contraire du domine de l'embarqué contraint et de l'embarqué, on définit dans ce document l'informatique débarquée comme étant un système généraliste, non spécifique à une fonction prédéfinie.

Dans le cas d'un tel système, un programme ne s'exécute pas seul sur le matériel et il laisse la gestion de la mémoire à d'autres composants logiciels. La mémoire d'un programme est allouée par le système d'exploitation et l'environnement d'exécution. Le *système d'exploitation* est un programme gérant les ressources matérielles de la plateforme pour l'exécution d'autres logiciels. L'*environnement d'exécution* (ou *runtime*) est un composant logiciel permettant l'exécution d'un programme en lui fournissant des fonctionnalités définies par le langage utilisé. On distingue notamment dans ces fonctionnalités la gestion de la mémoire dynamique et la gestion d'entrées / sorties vers le système d'exploitation. Au contraire dans un système embarqué contraint le système d'exploitation n'est pas forcément présent et le programme peut s'exécuter en utilisant uniquement le support du runtime.

Lors de son exécution, un programme n'a pas le même usage de toute sa mémoire, conduisant à des mécanismes d'allocation variés. Nous allons donc aborder dans ce chapitre ces différents mécanismes d'allocation. Puis nous détaillerons le problème de fragmentation induit par la gestion mémoire dynamique ainsi que des solutions classiques qui lui ont été apportées.

Après quoi nous développerons les aspects de l'allocation mémoire dynamique interagissant avec les hypothèses de notre étude. Nous montrerons que la méthode classique d'évaluation d'un allocateur mémoire dynamique n'est pas adaptée à notre étude. En effet dans le cas d'une mémoire non uniforme les accès à l'objet alloué influent sur les performances de l'ensemble de l'application.

2.1 Allocation mémoire

Nous allons aborder dans cette section les différentes manières d'allouer de la mémoire pour un programme en fonction de l'utilisation qu'il en fait. Nous allons voir ici les différentes manières pour un système informatique d'allouer de la mémoire pour un programme. Pour illustrer ces différents moyens, nous utiliserons le langage C, dans un programme hypothétique s'exécutant sur machine nue (cas de notre étude). Il est donc intéressant de noter dès l'abord que dans ces exemples, le programme repose sur un environnement d'exécution sans OS.

2.1.1 Allocation statique

Certaines des données dont un programme a besoin sont statiques. Ces données ont la même durée de vie que le programme et occupent donc la mémoire du début à la fin de son exécution. Le code et les variables globales en sont les exemples principaux. Dans le cas d'un système débarqué, lancé par le système d'exploitation, ces données sont chargées en mémoire avant l'exécution du programme et y résident jusqu'à ce qu'il se termine. Dans le cas d'un système embarqué contraint le code pourra être exécuté en place et le runtime chargera les variables globales en mémoire avant l'exécution du programme. Dans le cas de figure de l'allocation statique, la taille et le nombre des données sont connues à la compilation. Le système d'exploitation peut donc allouer ces données et la gestion de la mémoire n'implique aucun surcoût à l'exécution mis à part la copie initiale des données si elle est nécessaire.

Dans le cas du langage C, comme pour de nombreux langages, les allocations statiques ne nécessitent en général pas d'intervention du runtime, les données étant déjà présentes dans le binaire. Si il n'est pas exécuté en place, ou par exemple dans le cas de variables globales initialisées à 0, le runtime réservera un emplacement mémoire et copiera le contenu du binaire dedans. Dans le cas de variables initialisées à 0, seule leur taille est stockée dans le binaire, et le runtime réservera l'espace et l'initialisera. L'ensemble de ces opérations a lieu avant le début de l'exécution du programme.

2.1.2 Allocation dynamique

Variables locales : la pile

Contrairement aux variables globales, le nombre des variables locales n'est pas connu avant l'exécution. En effet l'allocation des variables locales est fait à l'appel d'une fonction, et le nombre et l'ordre de ces appels peuvent dépendre des entrées du programme. Ainsi un mécanisme de gestion de la mémoire doit être mis en œuvre durant l'exécution.

La portée de ces variables, c'est à dire l'ordre dans lequel elles sont allouées et désallouées est néanmoins connus à la compilation, permettant de les gérer sous la forme d'une pile. Lorsqu'une fonction est appelée, la zone mémoire dont elle a besoin est allouée au sommet de la pile, et lorsqu'elle retourne cette zone est libérée. Cette solution permet de gérer durant l'exécution la mémoire dédiée aux variables locales en limitant l'espace mémoire qu'elles occupent. De plus, ce mécanisme logiciel n'entraîne qu'un faible coût à l'exécution et est invisible au programmeur d'application.

L'espace de la pile est réservé et initialisé par le runtime avant l'exécution du programme. Toutefois, une fois la pile initialisée par le positionnement du pointeur de pile (stack pointer), le runtime n'a plus besoin de s'en occuper dans le cas du langage C. En effet, le code généré par le compilateur contient déjà toutes les manipulations de registres permettant les empilements et les dépilements, pour peu que pointeur de pile, qui est un registre particulier du processeur soit correctement positionné.

Objets dynamiques : le tas

Un programme peut toutefois avoir besoin de variables dont l'ordre d'allocation et de désallocation est arbitraire, en plus d'une taille et d'un nombre inconnu avant l'exécution. Pour combler ce besoin, l'environnement d'exécution met à la disposition du programmeur d'application une interface d'allocation et de désallocation de la mémoire. Un programme peut donc par le biais de ce mécanisme allouer et désallouer de la mémoire de manière arbitraire. Ce mécanisme implique

une plus grande complexité à l'exécution, mais permet de gérer des besoins mémoire dont le nombre et la taille ne sont pas connus à la compilation. En effet les blocs mémoire alloués de cette manière servent généralement à stocker des structures de données complexes, construites à partir des entrées du programme. Nous appellerons ces blocs mémoire alloués dynamiquement des *objets*. Bien que les objets de la programmation orientée objet puissent être alloués de cette manière, nous désignons également une structure C allouée sur le tas comme un objet. Nous définissons également ici le *site d'allocation* d'un objet comme l'adresse de retour dans le code appelant la primitive d'allocation mémoire.

Ces objets peuvent être alloués à n'importe quel moment de l'exécution, et leur nombre et leur taille n'est pas connue à l'avance. De plus, l'ordre de leur allocation et de leur désallocation est également arbitraire. C'est ce dernier facteur qui empêche l'utilisation d'une pile, contrairement aux variables locales. Le runtime réserve donc auprès du système d'exploitation une zone mémoire qu'il va dédier à l'allocation et à la désallocation de ces objets durant l'exécution : le *tas*. Si un système d'exploitation est présent, cette zone mémoire pourra grandir (ou être réduite) durant l'exécution par des appels systèmes, dans le cas inverse la taille du tas est usuellement définie pour une cible matérielle particulière à la compilation. Maintenir une représentation de la mémoire décrivant quelle zone est occupée et quelle zone est libre est une tache complexe induisant un surcoût important à l'exécution. Cette complexité est également masquée au programmeur d'application qui doit toutefois demander l'allocation et la désallocation des objets de son application.

Mais ce mécanisme de gestion mémoire permet aussi de réutiliser les zones libérées et limite donc la taille de mémoire requise pour l'exécution d'un programme. Ainsi une variable globale restera accessible durant toute l'exécution et la mémoire d'une variable locale sera libérée au retour de la fonction où elle est déclarée. Un objet quant à lui reste en mémoire le temps de son utilisation, jusqu'à sa libération. Cette utilisation peut représenter la majeure partie de l'exécution ou le temps passé dans une fonction. Ainsi les objets d'un programme peuvent occuper l'intégralité du tas à un instant donné, puis être tous désalloués et ré-alloués plus tard, sans avoir besoin de plus de mémoire.

Dans notre exemple illustratif, l'allocation mémoire dynamique est un des rôles fondamentaux de l'environnement d'exécution. En effet la répartition de la mémoire devant être décidée pendant l'exécution, elle repose entièrement sur celui-ci. Il expose au programme un ensemble de primitives de gestion mémoire dynamique dont les plus classiques sont `malloc()` et `free()` servant respectivement à l'allocation et à la désallocation des objets. Les structures de données décrivant l'état du tas ainsi que ces primitives forment l'allocateur mémoire dynamique comme nous allons maintenant le voir.

Allocateur mémoire dynamique

L'algorithme de gestion de la mémoire dynamique est appelé un allocateur mémoire. Il maintient une représentation de l'occupation de la mémoire du tas permettant l'allocation des nouveaux objets dans des espaces non occupés et la désallocation des objets. Les primitives exposées au programmeur d'application sont assez simples, limitées à l'allocation et à la désallocation.

L'allocateur répond donc aux requêtes d'allocation et de désallocation de l'application formant le profil d'allocation pour une exécution :

DÉFINITION - *profil d'allocation* : séquence ordonnée de requêtes d'allocation et de désallocation mémoire dynamique. Chaque allocation est caractérisée par sa taille. Le profil d'allocation est spécifique à l'exécution d'une application avec un jeu de donnée d'entrée (voir chapitre 6).

La figure 2.1 présente un exemple de représentation du profil d'allocation en fonction du temps en cycles processeurs pour une de nos applications cible : Ecdsa, dont la fonctionnalité est de signer électroniquement un message avec une clé générée en utilisant les courbes elliptiques.

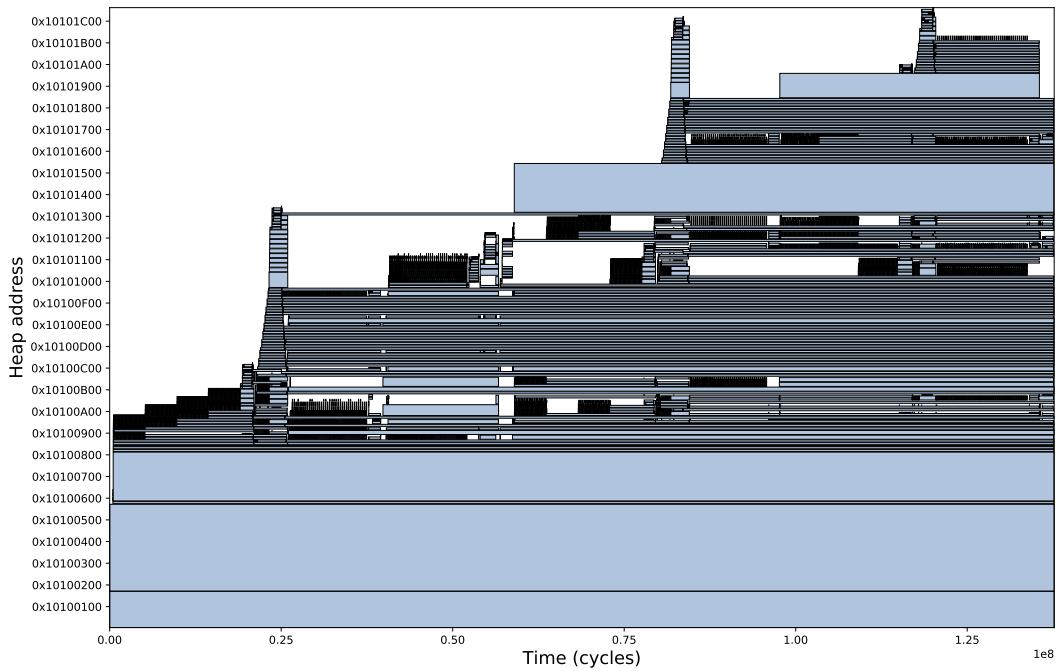


FIGURE 2.1 – Objets du tas : **Ecdsa**, exécution de référence, jeu de données 0

Chaque objet est représenté par un rectangle dont la hauteur correspond aux adresses du tas occupé, sa longueur correspondant à sa durée de vie. Les zones noires correspondent à des objets de très faible taille ou d'une durée de vie très courte.

2.2 Allocation mémoire dynamique manuelle

Le problème de l’allocation mémoire dynamique a été un problème central dans l’informatique dès 1960. Wilson et al. [WJNB95] ont produit un article de veille en 1995 qui reste d’actualité aujourd’hui pour ce qui est de la gestion mémoire manuelle. La littérature oppose généralement la *gestion mémoire manuelle*, où le programmeur applicatif doit émettre explicitement une requête d’allocation et de désallocation à la *gestion mémoire automatique* où le gestionnaire mémoire dynamique décide quel bloc mémoire ne sera plus utilisé sans intervention du programmeur.

Nous laissons de fait hors de cette étude la plus grande partie de la littérature récente qui concerne la gestion mémoire dynamique automatique, nécessitant le support de l’adressage virtuel par la plateforme matérielle.

Ces restrictions faites, les allocateurs mémoires généralistes auxquels nous nous intéressons n’ont guère changé depuis l’article de 1995. Par exemple, l’allocateur mémoire de Doug Lea, DLMalloc [LG96], publié en 1987 dans sa première version est toujours utilisé, modulo quelques adaptations, dans la bibliothèque GNU du langage C (glibc [Gli]), donc dans de nombreux systèmes Linux, notamment débarqués. On retrouve ce même allocateur mémoire dans newlib [New], la bibliothèque C de référence pour l’embarqué.

Nous considérerons donc que ces allocateurs, de par leur large adoption et leur longévité présentent de bonnes solutions dans le cas général. Toutefois comme affirment Wilson et al. [WJNB95] :

«It has been proven that for any possible allocation algorithm, there will always be the possibility that some application program will allocate and deallocate blocks in some fashion that defeats the allocator’s strategy, and forces it into severe fragmentation.»

Il n’est donc pas surprenant que des allocateurs spécialisés soient développés et utilisés dans des domaines spécifiques comme l’embarqué où le jeu vidéo [DSCM+15]. En effet, dans ces deux exemples les contraintes de performances sont importantes et le profil d’allocation des applications peut exhiber des particularités pour lesquelles les implémentations standards ne présentent pas de bons résultats.

Nous allons maintenant voir sous quelles conditions un allocateur mémoire doit fonctionner, et comment est géré le tas.

2.3 Problème de fragmentation

L'ordre, le nombre et la taille des requêtes émises par l'application est arbitraire. Il en découle que l'espace mémoire libéré par une désallocation peut être trop petit pour la satisfaction des requêtes suivantes. Il est également possible que la satisfaction d'une requête laisse un espace trop petit pour être utilisé. Ainsi la satisfaction d'une requête peut être impossible sans que le tas soit plein. De plus la satisfaction des requêtes doit satisfaire les contraintes suivantes :

- L'allocateur doit retourner un bloc faisant au moins la taille de la requête
- Les requêtes de l'application doivent être satisfaites dans l'ordre
- L'allocation doit être contigüe et sans recouvrement

Il est également à noter que dans nos conditions, il n'est pas possible de déplacer un bloc précédemment alloué. L'allocateur mémoire doit donc satisfaire ces requêtes avec pour seule information leur taille, et sa décision est irrévocabile. Le but d'un allocateur mémoire est donc la satisfaction des requêtes d'allocations sous ces contraintes précédentes tout en minimisant la fragmentation du tas, et le plus vite possible.

Il est à noter que la fragmentation peut être interne ou externe. Dans le cas de la fragmentation interne l'espace perdu est inclus dans l'objet, l'allocateur alloue un peu plus d'espace pour éviter de laisser un espace inutilisable qui devra être référencé par sa structure de donnée. La fragmentation externe est celle communément admise, correspondant à de l'espace inutilisable entre deux objets alloués dans le tas.

La résolution du problème de fragmentation dépend donc des décisions de l'allocateur, mais est conditionnée par les requêtes futures de l'application et les réponses qui y seront apportées par l'allocateur mémoire. En effet pour une requête de taille N , si l'allocateur utilise un bloc dont les voisins ne sont pas libres de taille T , comment considérer le bloc restant de taille $T - N$? Il ne peut être considéré inutilisable que si l'application n'émet que des requêtes d'une taille supérieure à $T - N$ dans le futur, tout du moins tant que les voisins du bloc cible ne sont pas libérés, ce qui peut avoir lieu à une date arbitraire.

Une notion importante pour un allocateur mémoire est celle d'empreinte mémoire.

DÉFINITION - *empreinte mémoire* : espace mémoire requis par un allocateur pour satisfaire sans échec les requêtes d'un profil d'allocation. Facilement mesurable à posteriori comme l'espace mémoire entre le début de l'objet le plus bas du tas et la fin de l'objet le plus haut dans l'espace d'adressage. Il dépend de l'allocateur, du profil d'exécution et de l'architecture mémoire, car un même allocateur peut avoir une empreinte mémoire différente pour allouer les mêmes objets si il dispose de plus ou moins d'espace mémoire.

Dans les faits l'allocateur mémoire que nous avons choisi ne présente pas ce comportement dans son implementation à destination de l'embarqué.

Comme dit précédemment, les allocateurs efficaces sont donc capables d'exploiter des régularités dans le profil d'allocation des applications. Nous allons donc illustrer dans la suite les comportements d'allocation d'applications et quelques mécanismes classiques d'allocation.

2.3.1 Comportement des applications et mécanismes d'allocation

Les applications ont des profils d'allocation différents et les allocateurs mémoires mettent en place des stratégies pour exploiter les régularités de ces profils d'allocation. On distingue trois catégories de comportement classiques d'allocation en terme de profil mémoire :

- *rampe* : accumulation monotone d'objets au cours d'une exécution.
- *pic* : accumulation rapide d'objets pour un calcul, suivi d'une libération partielle ou totale des objets alloués pour ce calcul.
- *plateau* : utilisation de structures de données construites au début de l'exécution avec peu de variation de la taille de ces structures durant l'exécution, par exemple un graphe construit au lancement du programme avec peu d'insertions et de déletions durant l'exécution.

Ces trois comportements peuvent se succéder par phases ou être superposés au cours d'une même exécution.

Une stratégie pourra par exemple être "si un bloc doit être découpé pour satisfaire une requête, essayer de minimiser le reste perdu". La notion de stratégie ne devant pas être confondue avec le mécanisme qui l'implémente d'après Wilson et al. Plusieurs mécanismes classiques ont été étudiés comme le *best fit* consistant à chercher le bloc libre pouvant satisfaire la requête au plus juste ou le *first fit* parcourant la liste des blocs vides jusqu'à trouver un bloc pouvant satisfaire la requête et découpant celui-ci.

On peut aussi noter certains mécanismes comme le "*deffered coalescing*" où la fusion d'un bloc libéré avec ses voisins (libres) n'est pas effectuée à la désallocation, mais où l'allocateur maintient une liste de blocs récemment libérés, ainsi, un programme demandant un bloc mémoire de la taille d'un de celui qu'il a libéré récemment pourra être servi plus rapidement.

Pour une description détaillée des différents mécanismes d'allocation mémoire nous recommandons au lecteur de consulter l'article de Wilson et al. [WJNB95]. Nous allons néanmoins décrire et analyser dans la suite le comportement de l'allocateur DLMalloc, étant donné que nous avons choisi de le réutiliser dans cette étude au vu de ses bons résultats dans le cas général.

2.3.2 Un allocateur mémoire : DLMalloc

Nous avons choisi de nous baser sur l'algorithme d'allocation écrit par Doug Lea [LG96], que nous désignerons dans ce manuscrit par *DLMalloc*. Écrit en 1987, il a subi de nombreuses évolutions mais la stratégie qu'il implémente n'a changé qu'à la marge.

Sa stratégie repose sur deux mécanismes permettant l'organisation de la mémoire, son allocation et la fusion des blocs libérés. La recherche d'une zone mémoire assez grande pour satisfaire une requête est faite en *best fit*, c'est à dire en allouant au plus juste de manière à minimiser la fragmentation.

L'information d'utilisation des blocs mémoires est encodée dans les blocs libres à l'aide de balises introduites aux limites des objets (mécanisme de *boundary tags*). Ce système permet à la fois une fusion facile des blocs libres côté à côté et un parcours dans les deux sens à partir de n'importe quel bloc libre. Il est à noter que les informations étant contenues dans les blocs libres, cette structure de contrôle n'induit pas de surcoût en mémoire pour décrire les blocs libres. La figure 2.2a décrit l'organisation des métadonnées dans la zone mémoire du tas.

Le mécanisme implémentant la recherche de bloc libre repose sur une *segregated free list*. Il s'agit d'un ensemble de listes contenant les blocs libres séparés par classes de taille. De plus les blocs contenus dans chaque liste sont triés par taille croissante. La structure de l'allocateur contient donc 128 listes de blocs libres doublement chaînées comme décrit dans la figure 2.2b. La répartition des classes de taille est pseudo-logarithmique et les tailles en dessous de 512 octets sont adressées spécifiquement, chaque liste contenant les blocs d'une taille unique.

L'implémentation de cette stratégie est accompagnée d'optimisations améliorant son comportement sur des points particuliers. On peut notamment évoquer un mécanisme de préservation de la "wilderness", le bloc pouvant être étendu par un appel au système d'exploitation, pour limiter le nombre d'appels système, réputés cher en temps processeur. Ces optimisations n'étant pas forcément appliquées dans le cas d'un système sans système d'exploitation.

Partant de cet exemple nous allons maintenant aborder l'impact de devoir utiliser un algorithme d'allocation mémoire dynamique sur un système mémoire hétérogène.

2.4 Mémoire hétérogène et allocation dynamique

Une des hypothèses traditionnelles de l'allocation mémoire dynamique est que la mémoire du tas est uniforme. Nous allons voir ici quelles sont les conséquences pour l'allocation mémoire dynamique de contredire cette hypothèse.

2.4.1 Influence sur les performances

Les performances d'un allocateur mémoire sont généralement mesurées selon deux aspects. La fragmentation représente l'espace perdu à l'intérieur du tas, et le temps en cycles processeur passé dans les routines d'allocation et de désallocation. Ces deux aspects mesurent l'efficacité avec laquelle un allocateur mémoire est capable de satisfaire une requête.

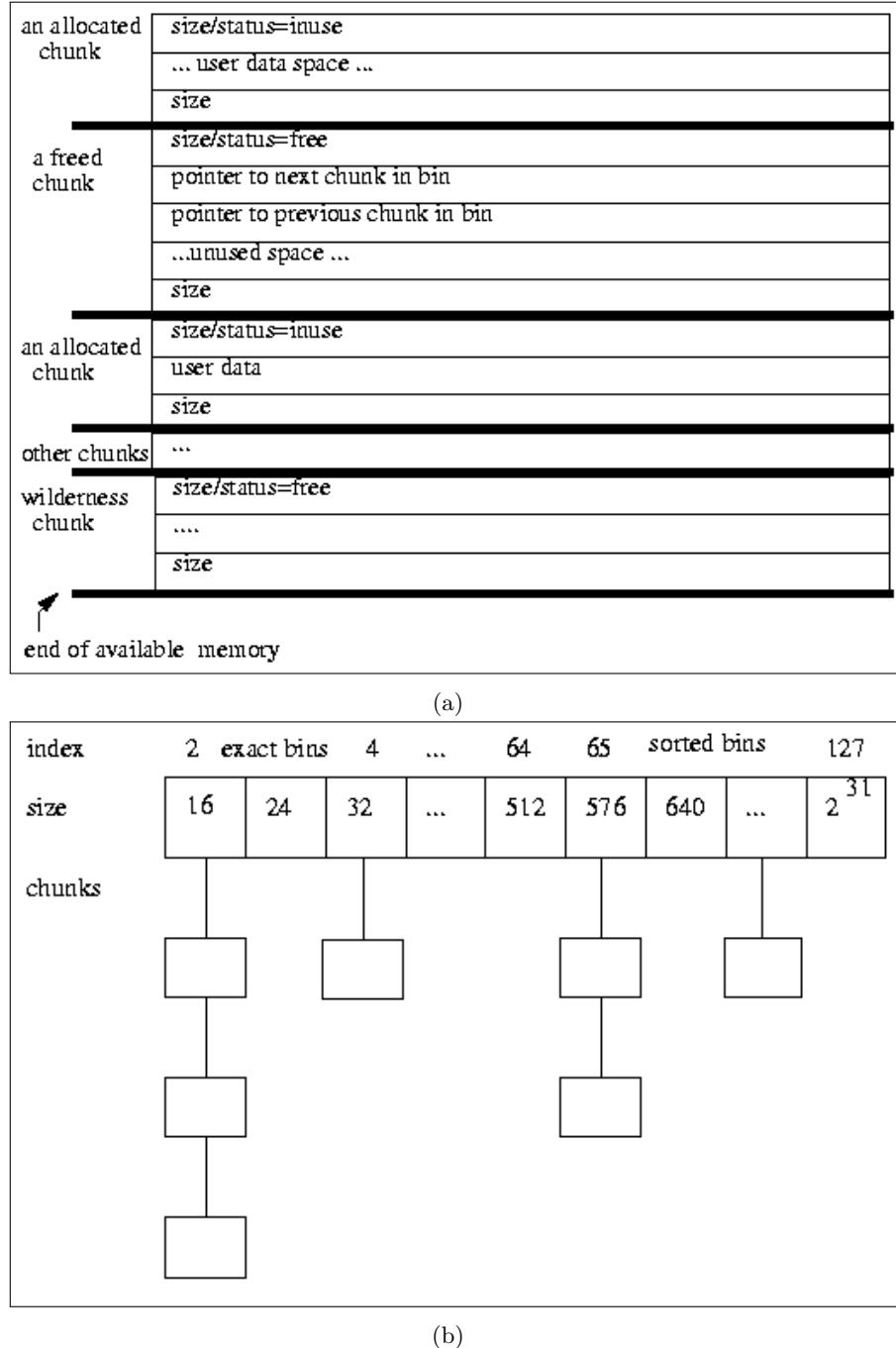


FIGURE 2.2 – Structures de données et organisation mémoire de l'allocateur DLmalloc Doug LEA et Wolfram GLOGER. A memory allocator. 1996
 (a) métadonnées contenue dans la mémoire, (b) structure pour le parcours des blocs libres

Mais si la mémoire du tas est hétérogène, l'adresse choisie par l'allocateur a un autre impact sur les performances de l'application. Nous considérons un exemple où le tas est composé d'une zone mémoire rapide et une autre lente, exemple dans lequel les accès de l'application à ses objets ne sont pas uniformes. Alors un objet plus accédé placé dans une mémoire ayant une latence plus faible pourra accélérer l'exécution de l'application ou à l'inverse le ralentir si il est placé dans une mémoire ayant une latence plus élevée.

Un des buts de ce manuscrit est de mettre en évidence cet impact, ainsi que de quantifier son importance vis à vis des performances du gestionnaire mémoire dynamique en lui-même. Nous mettrons donc en avant dans les contributions la nécessité de prendre en compte les mécanismes et les optimisations de l'allocateur mémoire dynamique.

2.4.2 Méthodologie d'évaluation des allocateurs mémoire

La méthodologie classique d'évaluation d'un allocateur mémoire présentée par Wilson et al. [WJNB95] est de rejouer les traces d'appels à l'allocateur émises par des programmes réels et de mesurer le temps et la fragmentation de l'allocateur en test pour la satisfaction de ses requêtes.

Mais comme nous l'avons vu, si le tas n'est pas constitué de mémoire homogène, alors les choix de l'allocateur mémoire ont un impact sur les performances globales du système, en dehors du temps mis par l'allocateur à répondre à la requête de l'application. Or la méthode classique d'évaluation des performances d'un allocateur mémoire ne permet pas de capturer cet impact. Nous avons donc fait le choix de rejouer les applications cibles plutôt que de seulement rejouer leur profil d'allocation. Ainsi nous seront à même de mesurer l'impact sur les performances provoqué par l'hétérogénéité mémoire du tas.

Nous ne discutons pas ici les notions de localités en terme d'accès mémoire vers les objets du tas. En effet les objets alloués au même moment ont plus de chance que la moyenne d'être désalloué au même moment, et ce comportement est exploité dans la résolution du problème de fragmentation par l'allocateur mémoire. Toutefois, dans des systèmes disposant d'un cache matériel des objets accédés strictement à la suite alloués contigûement en mémoire peuvent améliorer les performances en permettant au cache d'exploiter la localité des accès. Comme nous le verrons dans le chapitre suivant, notre étude se place dans un cas excluant l'usage du cache matériel, nous ne prenons donc pas en compte ce mécanisme.

2.4.3 Notion de temps pour l'allocateur mémoire

D'après la définition d'un allocateur mémoire que nous avons adopté, un allocateur mémoire n'a pas de représentation interne du temps. Nous entendons par là que l'algorithme du gestionnaire mémoire n'a pas besoin de mesurer le temps qui passe pour remplir sa fonction. Wilson et al. choisissent donc de représenter l'allocation mémoire en fonction du temps en mégaoctets alloués. Bien que surprenante de prime abord, cette approche est intéressante pour analyser le comportement d'un allocateur mémoire. En effet, les requêtes reçues par l'allocateur mémoire sont ordonnées et d'une certaine taille, mais que deux requêtes soient reçues à 10 ou 1000 cycles processeur d'intervalle n'a aucun impact sur l'allocation.

Du point de vue de l'allocateur, qui n'est exécuté que lors de l'émission d'une requête de l'application, seul l'état du tas importe. L'occupation des différents blocs mémoire, c'est à dire la fragmentation, influe sur ses performances.

Dans nos conditions, cette considération n'est plus exacte, quant au placement à une adresse donnée, il est également important de prendre en compte le type de mémoire et les accès à l'objet. Cependant la notion de temps n'est pas plus visible pour l'allocateur.

Ainsi nous ne considérons pas la notion de temps en cycle processeur pour ce qui est relatif à l'allocateur mémoire dynamique dans cette étude. Comme nous le verrons dans les chapitres suivant, dans nos conditions, et du point de vue de l'allocateur, le nombre d'objet alloué est une mesure du temps plus significative pour l'utilisation du tas. C'est la même approche méthodologique qui fait graduer l'axe des temps en Mégoctets à Wilson [WJNB95] : du point de vue de l'allocateur mémoire il ne se passe rien entre deux requêtes.

2.5 Conclusion

Nous avons brièvement introduit dans ce chapitre l'allocation mémoire dynamique. Cette introduction nous permettra d'analyser le comportement de l'allocation mémoire dynamique dans nos conditions. Nous avons de plus présenté un algorithme de gestion mémoire dynamique largement adopté sur lequel nous avons basé notre travail. Nous avons finalement discuté les implications de la remise en question d'une hypothèse traditionnelle de l'allocation mémoire dynamique dans notre étude. Nous allons maintenant introduire le domaine ciblé par notre étude, à savoir l'embarqué contraint, et son impact sur les solutions à notre problème.

Chapitre 3

Systèmes embarqués et IOT

Notre travail est centré sur les systèmes embarqués. Les contraintes de ce domaine influent sur les architectures matérielles comme sur les architectures logicielles de ces systèmes. Nous allons donc dans ce chapitre aborder les caractéristiques de ces systèmes.

Nous détaillerons également à l'intérieur de ces systèmes les quels sont concernés par notre approche. En effet le domaine de l'embarqué peut présenter des conditions d'exécutions très variées et des contraintes très différentes. Ainsi le domaine de l'embarqué désigne des plateformes matérielles et logicielles très variées. Celles ci allant du microprocesseur dans une carte à puce ou un tag RFID [BPSY+08] à des appareil capable de calculs importants, disposant de plusieurs niveaux de caches [DSCM+15].

Nous aborderons donc différents types de systèmes embarqués pour restreindre le périmètre d'application de notre approche. Tout d'abord nous évoquerons les systèmes dits critiques. Dans ces systèmes, l'utilisation de l'allocation mémoire dynamique est généralement exclue. En effet les algorithmes d'allocation mémoire dynamique ne présentent pas de garanties temporelles à l'exécution.

Une autre contrainte centrale du domaine est celle de l'alimentation, et de l'efficience énergétique. En effet certains modules matériels intégrés au système permettent d'augmenter ses performances. Toutefois, dans le domaine de l'embarqué, la rentabilité énergétique de ces modules n'est pas garantie. Nous développerons ce raisonnement notamment vis à vis du cache matériel et de l'unité de gestion de la mémoire. En effet ces deux modules matériels représentent une consommation énergétique et une place sur la puce qui est trop importante dans de nombreux cas. Ainsi, de nombreuses architectures matérielles n'en incluent pas.

Nous choisissons donc de nous intéresser à des systèmes en étant dépourvu. Par la suite nous discuterons les conséquences de cette hypothèse. En effet l'absence d'une MMU rend le déplacement d'un objet impossible après son allocation. Et la présence d'un cache change les accès effectivement réalisés dans la mémoire.

Nous aborderons finalement l'architecture logicielle de ces systèmes. Notre approche s'intègre également aux contraintes du domaine sur cet aspect. En effet nous considérons une architecture logicielle sur machine nue/. Ainsi nous ne reposons sur aucun service d'un système d'exploitation, pas toujours présent dans le domaine. De plus nous faisons le choix de ne pas modifier l'application. Nous limitons donc les modifications apportées au logiciels à la couche logicielle du runtime, sans en modifier l'interface.

Mais dans un premier temps nous allons ici proposer une définition des systèmes intéressés par notre approche, c'est à dire les systèmes embarqués contraints.

DÉFINITION - système embarqué contraint : Système informatique autonome remplissant une ou plusieurs fonctions spécifiques dans des conditions contraintes en ressources : espace mémoire, énergie, temps d'exécution. Ainsi, contrairement à un système plus généraliste, ou par exemple à un système embarqué haute performance, le logiciel développé pour cette plateforme lui est spécifique et est lui aussi contraint.

Par exemple un smartphone présente un OS et l'utilisateur installe des applications dont l'utilisation des ressources ne sont pas optimisées pour sa plateforme. Au contraire, un objet connecté ou une station météo déployée dans la nature présentent des logiciels très optimisés

pour l'exécution sur leur matériel. Nous considérons également que notre approche intéresse de nombreux objets connectés.

DÉFINITION - *IoT*, Internet of Things – Systèmes embarqués communicants à faible coûts. Le domaine de l'IoT englobe toutefois une vision large, des réseaux de capteurs aux infrastructures réseau permettant l'acheminement des données, au traitement de ces données par des stratégies big data.

Les objets de l'internet des objets sont donc dotés de microprocesseurs capables de communiquer vers le monde extérieur. Ils doivent de plus limiter au maximum leur consommation énergétique. Du point de vue de cette étude ces objets sont donc proches de ceux du domaine de l'embarqué constraint. Nos travaux considèrent cependant uniquement les objets communicants en eux mêmes. D'un point de vue économique il est prévu que le nombre de ces systèmes, déjà important, augmente fortement dans les années futures [JLLRKR14].

3.0.1 Critique et temps réel

Les systèmes embarqués les plus contraints sont ceux dont la mission ne peut se permettre d'échouer. Ces systèmes, dits critiques, doivent pouvoir garantir leur exécution, mais doivent souvent le faire en respectant des contraintes de délai.

Nous avons présenté au chapitre 2 le problème de fragmentation. Ce problème, résolu à l'exécution par le gestionnaire mémoire dynamique ne peut pas être facilement borné temporellement. En effet, l'ordre des requêtes, leur taille et leur nombre sont arbitraire. Ainsi il n'est pas possible de prévoir un temps d'exécution pour le système respectant ses contraintes temporelles. Le calcul de pire temps d'exécution ou *WCET*, pour Worse Case Execution Time dépendant dans ce cas du jeu de donnée. Dans ces systèmes l'utilisation de l'allocation mémoire dynamique est donc généralement exclue.

3.0.2 Aspect économique et méthodologies de développement

Le domaine de l'embarqué, visant la réalisation de plateformes spécialisées pour une ou quelques fonctions prédéfinies est structuré différemment de ce que l'on peut trouver pour le reste des systèmes informatiques. En effet si le logiciel cible, ou en tout cas la fonction à remplir, est connue en amont de la conception de la plateforme il est possible d'optimiser le fonctionnement de celle-ci.

La conception d'une telle plateforme se base notamment sur la notion de *conception conjointe matériel-logiciel*. Cette approche de co-développement permet de baisser le temps de conception et de production du système (time to market).

Il est possible d'intégrer des modules préconçus et paramétrables dans le design matériel de la puce. Ainsi, par exemple la taille de la mémoire peut être paramétrée pour l'application cible. En contrepartie ces modules nécessitent une phase d'intégration. C'est le positionnement adopté par la startup eVaderis, partenaire initial de ce projet[LJBDG+16].

En revanche, et contrairement à l'informatique débarquée, le code est souvent spécifique à une cible matérielle, au moins partiellement, et est donc cher à développer. Les systèmes embarqués sont donc plus optimisés, particulièrement en termes de consommation énergétique, pour une fonction, mais en contrepartie ils nécessitent un temps de développement plus long et plus coûteux.

Il est également à noter que cette distinction peut être plus ou moins forte suivant les cas. Certains systèmes embarqués permettent d'installer ou de modifier leur logiciel. De plus, la conception d'un système embarqué prévoit généralement une possibilité de mise à jour logicielle, notamment pour ce qui est du domaine de L'IoT. En effet, un défaut de mise à jour dans un environnement connecté pose de graves problèmes de sécurité comme l'ont montré les réseaux botnet dans les dernières années [KKS17].

Ainsi les systèmes embarqués contraints ont des besoins qui nécessitent l'utilisation d'algorithmes de plus en plus complexes. Or l'implémentation d'une fonction en matériel peut limiter la capacité de mise à jour de cette fonction. Dans ces cas là, l'utilisation d'une solution logicielle semble justifiée.

3.1 Architectures embarquées

Comme nous venons de le voir, les systèmes que nous étudions sont soumis à de nombreuses contraintes. Nous allons maintenant considérer l'influence de ces contraintes sur l'architecture matérielle des systèmes.

Dans un contexte où les coûts de développement sont importants, les architectures matérielles sont optimisées pour remplir les fonctions cibles le plus efficacement possible. Mais le coût de la plateforme doit être également maîtrisé car le secteur est concurrentiel et ces systèmes sont généralement produits en grand nombre. Cette tendance étant accentuée par l'émergence de l'internet des objets.

Les architectures embarquées limitent donc au maximum la surface de silicium utilisée par la puce et la consommation énergétique. Ainsi le système, alimenté sur batterie, peut être allégé (en réduisant la capacité de la batterie) ou voir sa durée de vie augmenter.

Nous allons donc voir dans la suite les spécificités des architectures que nous considérons pour cette étude.

3.1.1 Spécificités matérielles des systèmes embarqués contraints

Les architectures débarquées, ou celles de l'embarqué haute performance présentent de nombreux modules visant l'accélération de l'exécution. Au contraire les architectures de l'embarqué constraint doivent s'adapter à un manque de ressources. Ainsi, un module matériel accélérant l'exécution n'est intéressant dans ce contexte que si il permet d'augmenter l'efficience énergétique du système à un prix raisonnable. De plus ces modules occupent de la surface de silicium, et augmentent le prix unitaire de la plateforme finale. De nombreuses optimisations courantes dans l'informatique débarquée sont ainsi absentes des architectures embarquées.

Unité de gestion de la mémoire

Le module matériel permettant l'abstraction de la mémoire physique et donc l'adressage virtuel est appelé une *MMU*. L'*adressage virtuel* permet d'exposer un espace mémoire différent pour les différents programme exécutés par un système d'exploitation. Il permet également de facilement déplacer le contenu de la mémoire physique, la MMU se chargeant de la traduction entre adresse physique et adresse virtuelle. Extrêmement répandu en dehors de l'embarqué constraint, elle se révèle très coûteuse pour un système dont le logiciel, comme nous le verrons ci-après, peut se passer de l'adressage virtuel dans un soucis d'économie d'abstraction. De plus une MMU est composée de la mémoire rapide et consommatrice d'énergie, et la place occupée sur la puce est non-négligeable. Nous considérons ainsi dans nos travaux ne pas disposer de MMU sur notre matériel.

Cache matériel et mémoire Scratch Pad

Un cache matériel est un système d'abstraction automatique de la hiérarchie mémoire. Contenant une petite partie des données du programme il charge automatiquement depuis la mémoire principale (plus lente) les données accédées par le programme. Ainsi, les programmes accédant souvent aux données précédemment accédées, leur temps d'accès est grandement réduit si elles sont déjà dans le cache. Or, pour le domaine de l'embarqué constraint, la recherche académique [BSLBM02 ; Zen06] a démontré que pour la même surface en silicium il était plus intéressant d'utiliser une mémoire Scratch Pad. Une mémoire *Scratch Pad* (ou SPM, pour Scratch Pad Memory) est un banc de mémoire rapide directement accessible par le processeur. Elle joue le même rôle qu'un cache matériel, c'est à dire être une zone de mémoire rapide où stocker les données accédées régulièrement. Toutefois, contrairement à un cache matériel la gestion de l'espace mémoire est laissé au logiciel. Elle peut être intégrée à une hiérarchie mémoire (à la place du cache) ou être accessible en parallèle de la mémoire principale (hétérogénéité mémoire).

Cette approche est intéressante pour l'embarqué constraint car une zone mémoire gérée par le logiciel présente l'avantage de ne pas implémenter toute la circuiterie de contrôle autour de la mémoire que requiert le cache matériel. Ainsi les systèmes que nous étudions se passent de caches matériels. Cette décision fait donc reposer sur le logiciel la gestion de la mémoire Scratch Pad.

Exécution dans le désordre

Une optimisation efficace dans les systèmes haute performance est l'usage de modules permettant la réorganisation des instructions directement durant l'exécution. L'exécution dans le désordre permet d'éviter des attentes processeur, notamment dues à l'insertion de bulles dans le pipeline d'instruction. Ces mécanismes permettent d'accélérer significativement l'exécution d'un programme mais consomment beaucoup de surface sur la puce et également beaucoup d'énergie, ils sont ainsi généralement exclus des processeurs destinés à l'embarqué contraint.

Prédiction de branchement

La prédiction de branchement consiste à prédire quel chemin d'exécution suivra le programme. Si cette prédiction échoue, le pipeline est rempli des mauvaises instructions, et il faut l'invalider, résultant en une exécution plus lente. Encore une fois contrairement aux architectures hautes performances, les architectures embarquées sont plus efficientes énergétiquement, mais beaucoup moins performantes en terme de temps d'exécution. Dans le cas du prédicteur de branchement, celui ci, si il est présent, se trompera plus souvent mais occupera moins d'espace sur la puce et consommera moins d'énergie.

Exécution de code en place

Les architectures matérielles sont également contraintes en mémoire. L'approche classique est d'avoir une mémoire de stockage où résident les programmes qui ne sont pas exécutés et une mémoire de travail où ils sont chargés pour être exécutés. Au contraire dans le domaine de l'embarqué contraint le code est généralement exécuté directement depuis la mémoire où il est stocké. Cette approche permet d'économiser de la mémoire de travail pour la plateforme qui n'en a plus besoin pour stocker le code du programme. Elle est également possible car la fréquence d'opération des processeurs embarqués contraints est plus faible, et ainsi, l'impact sur les performances d'une telle exécution en place est moindre.

Ainsi les architectures matérielles des systèmes embarqués sont moins complexes mais plus efficientes énergétiquement. Comme nous l'avons défini au début de ce chapitre le domaine de l'embarqué contraint vise des systèmes replissant une ou plusieurs fonctions prédéfinies. L'architecture correspondante sera donc optimisée pour remplir cette fonction au plus juste en terme de consommation énergétique tout en respectant des délais temporels, le tout sur une puce la moins cher possible. Par exemple si une fonction doit s'exécuter plus rapidement, il peut être plus intéressant d'adapter sa fréquence d'exécution que d'introduire un module matériel consommant plus et occupant plus de place sur la puce.

3.2 Programmes embarqués

De même que les architectures matérielles évoquées ci-dessus, les logiciels que nous étudions sont spécifiques. Destinés aux plateformes embarquées contraintes, ils ne suivent pas une logique de design identique aux systèmes débarqués.

Dans un premier temps nous évaluons les contraintes directes pour le logiciel découlant de l'architecture matérielle. En effet, de nombreuses abstractions standard des systèmes débarqués sont trop coûteuses pour ces systèmes. Puis nous discuterons de l'impact de l'absence de ces abstractions et décrirons le contexte logiciel de notre projet.

Nous terminerons par discuter l'usage de la gestion mémoire dynamique dans ces conditions. En effet comme nous l'avons abordé au chapitre 2, il s'agit d'un mécanisme logiciel dont le coup en ressources (temps processeur et mémoire) peut s'avérer important dans l'exécution d'un programme. Il est toutefois nécessaire dans de nombreuses situations de l'embarqué contraint.

3.2.1 Contraintes logicielles

Le logiciel embarqué doit s'exécuter dans un environnement dont les ressources sont limités au strict minimum. En effet, pour garder le coût unitaire de la puce le plus bas possible la quantité de mémoire présente sur la puce est limitée. De plus la puissance de calcul est également limitée pour consommer le moins d'énergie possible. Il est néanmoins nécessaire de remplir une ou des

fonctions, généralement complexes et nécessitant une communication vers l'extérieur, au moins pour les plateformes IoT.

Dans le cas du logiciel débarqué, les applications se reposent normalement sur un ensemble de couches logicielles et d'abstractions du matériel permettant une plus grande facilité de développement. Ces abstractions permettent aussi d'exécuter des programmes sur beaucoup plus de cibles matérielles différentes si ces abstractions sont présentes dans les couches logicielles.

La première de ces abstractions est l'utilisation d'un système d'exploitation. Cette abstraction est largement répandue, que ce soit dans le domaine du débarqué, du calcul haute performance ou même pour l'embarqué haute performance. Toutefois pour les systèmes embarqués contraints, cette abstraction peut être dispensable. En effet, ces systèmes sont développés pour remplir une seule fonction et ne prévoient pas que l'utilisateur installe une application "inconnue" au gré de ses envies. De plus ces systèmes n'ont généralement qu'une interface limitée. Ainsi il est possible d'économiser le surcoût de cette couche logicielle.

La mémoire virtuelle est également une abstraction classique des systèmes moins contraints. Dans ce cas, sans le support matériel d'une MMU il n'est pas possible de se reposer sur cette abstraction. De plus la gestion de la mémoire virtuelle est souvent intégrée à l'OS.

De même pour l'exécution d'applications installées par l'utilisateur la notion de processus permet de répartir les ressources matérielles entre plusieurs programmes ou plusieurs parties d'un programme. Fortement liée à l'OS qui en assure la gestion, cette abstraction se révèle également moins intéressante pour répondre à des fonctions prédéfinies.

Toutes ces abstractions ne sont pas nécessaires à l'exécution d'un système embarqué contraint. Ainsi leur coût en ressources est généralement évité, en contrepartie d'une plus grande complexité du code mis en œuvre. L'approche retenue est donc généralement celle du développement dit *bare metal* ou *développement sur machine nue*. Le matériel est ainsi beaucoup plus exposé au logiciel qui doit être capable de le gérer directement pour remplir sa fonction.

Toutefois cette volonté d'éviter les abstraction coûteuses en ressources n'est pas forcément en contradiction avec la notion de ré-utilisabilité du code. Ainsi, même dans un logiciel sur machine nue, le code est généralement séparé en deux couches. On distinguera la couche applicative, réalisant la fonction ciblée par le système qui se repose sur une couche logicielle traitant directement avec le matériel. Ainsi une application devant s'exécuter sur un autre matériel n'aura qu'à adapter son interface vers cette couche logicielle bas niveau. Cette couche contient les drivers gérant chaque module matériel et permettant sa mise à disposition pour le logiciel, elle est souvent dénommée *Board Support Package* (BSP, logiciel de support de la plateforme) ou *Hardware Abstraction Layer* (HAL, couche d'abstraction matérielle). Ces deux couches logicielles ne sont d'ailleurs généralement pas développées par les mêmes développeurs. Nous nous intéressons dans cette étude à la couche applicative, c'est en effet elle qui à le plus recours à l'allocation mémoire dynamique.

3.2.2 Contraintes logicielles du domaine de l'embarqué en général

De la structuration des couches logicielles comme décrites ci dessous découle un certain nombre de spécificités qui s'appliquent au domaine de l'embarqué et qui guident certains des choix de notre travail.

Par exemple, le code étant généralement plus optimisé et plus complexe, le code source des librairies tierces parties est souvent indisponible. De plus les bases de code sont plus difficiles à modifier. Ces aspects nous poussent à ne pas proposer une interface différente de l'interface standard de l'interface d'allocation mémoire dynamique standard, c'est à dire les fonctions `malloc()` et `free()`.

De plus le code embarqué est plus souvent propriétaire car lié à un produit spécifique et souvent hérité de la version précédente du projet. Ces pratiques ne facilitent pas l'adoption de nouvelles approches si celles-ci ne s'adaptent pas à ces conditions. Elles guident notamment notre choix de ne pas modifier l'interface exposée au programme pour l'allocation mémoire dynamique.

3.2.3 Utilisation de la gestion mémoire dynamique

Nous avons présenté les aspects du domaine de l'embarqué contraint qui nous semblent importants pour cette étude. Ainsi nous pouvons maintenant aborder le sujet de l'utilisation de la gestion mémoire dynamique dans ce domaine.

Les architectures matérielles ne supportent pas l'abstraction de la mémoire virtuelle et le logiciel est dans notre cas sur machine nue. Ainsi nous pouvons nous interroger sur la gestion mémoire dynamique dans le même esprit de remplir la fonction du système au plus juste. Dans cette optique, une abstraction permettant de gérer la mémoire ayant un surcoût important comme présenté au chapitre 2 pourrait logiquement être laissée de coté. Cependant l'utilisation d'un tas permet de drastiquement augmenter la réutilisation de la mémoire. Ainsi dans les benchmarks que nous évaluons, la mémoire du tas peut être ré-alloué jusqu'à 59 fois.

En effet certains programmes peuvent être exécutés dans une zone mémoire de faible taille grâce à l'utilisation de la gestion mémoire dynamique. Nous soutenons que c'est dans ces cas d'utilisation que la gestion mémoire dynamique est indispensable pour le domaine de l'embarqué constraint. En effet, la ressource mémoire étant limitée, le fait de pouvoir réutiliser la mémoire devenue inutile permet d'utiliser des algorithmes dont les besoins en mémoire seraient trop importants sans gestion mémoire dynamique.

Nous présentons ci dessous quelques exemples d'utilisation intensives de l'allocation mémoire dynamiques applicables au domaine de l'embarqué constraint. Le routage, notamment pour des drones, ou la découverte d'environnement (qu'il s'agisse de l'environnement physique ou de services) requiert l'usage de l'allocation mémoire dynamique. De nombreux algorithmes de traitement du signal reposent de même sur ce mécanisme. Ces traitements peuvent correspondre à de l'amélioration ou de la fusion de données de capteurs ou de l'encodage multimédia. On peut également penser à la compression ou la décompression de fichiers. Dans l'optique du développement de l'IoT également, les algorithmes de cryptographie doivent être considérés, et ceux ci peuvent avoir fortement recours à l'allocation mémoire dynamique. L'IoT rejoint sur ce point l'évolution des systèmes embarqués contraints vers plus de complexité [DSCM+15].

Une solution qui peut correspondre à la méthodologie de l'embarqué constraint serait d'utiliser un module matériel dédié pour alléger la fonction à remplir en logiciel. Toutefois, la résolution du problème de fragmentation par un module matériel ne semble pas prometteuse. En effet, les requêtes sont de taille et de nombre arbitraire et les ordres d'allocation et de désallocation sont arbitraire.

Nous considérons donc dans cette étude l'utilisation de la gestion mémoire dynamique dans l'embarqué constraint comme ayant un coût important mais étant nécessaire dans de nombreux cas. De plus, nous limitons le périmètre de cette étude à la couche applicative. Nous pensons que la couche driver peut bénéficier de l'allocation mémoire dynamique en mémoire hétérogène. Néanmoins, cette dernière devant être adaptée pour chaque plateforme matérielle et utilisant beaucoup moins intensément l'allocation mémoire dynamique nous ne pensons pas qu'il soit nécessaire se reposer sur une API non modifiée pour l'utiliser à ce niveau. De plus le développeur de driver étant par définition au contact du matériel il est à même de l'utiliser directement.

En fonction de la cible logicielle et matérielle la gestion de la mémoire dynamique dans le domaine peut varier. Pour l'utilisation de systèmes très faible consommation, généralement récoltant leur énergie dans l'environnement on trouve des implémentations simpliste de la gestion mémoire dynamique. Ce domaine, le *tiny embedded* ne permet cependant pas d'utiliser des algorithmes tels que ceux que nous avons évoqué ci-dessus. L'implémentation la plus utilisée pour l'embarqué constraint est celle présentée au chapitre précédent : le DLMalloc. C'est ce gestionnaire mémoire dynamique qui est utilisé dans newlib [New], la librairie standard C de référence pour l'embarqué.

3.3 Conclusion

Nous avons présenté dans ce chapitre le domaine de l'embarqué et restreint le périmètre de notre étude aux systèmes embarqués contraints non critiques.

Nous pensons qu'il est intéressant de tirer parti de l'allocation mémoire dynamique dans de tels systèmes. En effet, ce mécanisme n'est utilisé que lorsqu'il est nécessaire, mais peut toutefois être utilisé intensément comme nous le verrons au chapitre 7. En effet, un programme embarqué peut allouer plusieurs milliers d'objets. Dans le cas de nos application le temps passé dans l'allocateur mémoire dynamique peut aller jusqu'à 15% du temps total d'exécution de l'application.

En résumé, les systèmes que nous considérons dans cette étude présentent les caractéristiques suivantes. Il ne s'agit pas d'un système critique, excluant l'usage de l'allocation mémoire dynamique.

Ils ne reposent ni sur l'usage d'une MMU, ni sur celui d'un cache matériel. En effet, ces modules sont généralement exclus des systèmes embarqués contraints que ce soit pour la place occupée sur la puce où pour leur consommation énergétique. Ces aspects matériels influent sur l'aspect logiciel de ces systèmes. Ainsi nous considérons des systèmes ne proposant pas d'abstraction comme la mémoire virtuelle et fonctionnant le plus souvent sans système d'exploitation. Nous considérons toutefois deux couches logicielles différentes, une directement au contact du matériel, incluant le runtime et une autre applicative.

Nous allons maintenant présenter les technologies mémoires émergentes dont les caractéristiques et l'intégration dans des puces embarqué motivent cette thèse.

Chapitre 4

Technologies mémoire pour l'embarqué

Nous avons présenté au premier chapitre les besoins en mémoire d'un programme informatique, notamment en terme d'allocation mémoire dynamique. Par la suite nous nous sommes intéressés au domaine de l'embarqué, et plus spécifiquement au domaine de l'embarqué contraint. C'est dans ces conditions que se place cette étude. Nous allons maintenant présenter les différentes technologies de mémoire émergentes non volatile qui sont l'une des motivation de ce travail.

En effet le domaine de l'embarqué est l'un des domaines d'application de ces nouvelles technologies mémoire. De plus, leur intégration aux systèmes sur puces actuels amène de nouveaux challenges et fait évoluer les architectures matérielles.

Notre focus concerne l'aspect logiciel de l'allocation mémoire dynamique sur des systèmes mémoires incluant ces technologies au même niveau matériel. Nous allons donc détailler ici les caractéristiques de ces différentes technologies et mettre en avant les caractéristiques avec lesquelles nous auront à travailler au niveau logiciel.

4.1 Mémopire de travail et mémoire de stockage

Tout d'abord nous souhaitons évoquer les différentes technologies mémoires utilisées traditionnellement dans le domaine de l'embarqué pour la conception de systèmes embarqués.

Nous avons expliqué au chapitre 2 qu'un programme, qu'il soit embarqué ou débarqué, a besoin de mémoire de travail pour s'exécuter, et usuellement de mémoire de stockage pour des données et du code. Nous avons également vu que dans le domaine de l'embarqué contraint, le code de l'application cible peut être exécutée directement en place dans la mémoire de stockage, limitant ainsi la quantité de mémoire de travail nécessaire à l'exécution du programme.

Mémoire de travail

La mémoire de travail usuellement utilisée pour les plateformes embarquées est la SRAM, pour Static Random Access Memory. En effet celle ci est rapide, facilement intégrable au cœur de la puce et dispose d'une très grande endurance. Au contraire, les systèmes débarqués nécessitent souvent une grande quantité de mémoire de travail et utilisent donc une autre technologie, la DRAM, pour Dynamic Random Access Memory. Celle-ci est en règle générale externe à la puce et beaucoup plus dense. Toutefois, cette augmentation de la taille de mémoire de travail disponible implique généralement l'usage d'un cache matériel pour masquer la latence de la DRAM et de la communication entre la mémoire et le processeur.

DÉFINITION - SRAM : Static Random Access Memory (mémoire à accès aléatoire statique). Basée sur un circuit de bascules pour retenir l'information. Elle ne nécessite pas de rafraîchissement pour conserver l'information et ne la retient pas en cas de coupure de courant. Elle consomme peu, mais doit donc être alimentée en permanence et présente ainsi une part de consommation statique. Le circuit qui la compose est relativement complexe, bien que basé exclusivement sur des technologies compatibles avec le processus de réalisation des circuit imprimés classique (CMOS). Ainsi la mémoire SRAM, bien que très rapide présente une implantation peu dense et donc coûteuse en surface de silicium.

DÉFINITION - *DRAM* : Dynamic Random Access Memory (mémoire à accès aléatoire dynamique). Cette technologie est extrêmement dense comparé au circuit requis pour un point mémoire de mémoire SRAM. Elle n'est pas facilement intégrable à l'intérieur des puces, et doit être rafraîchie en régulièrement pour conserver l'information, même sous tension. En effet sa conception stocke l'information à l'intérieur d'un condensateur, contrôlé par un transistor.

Dans le cas de l'embarqué contraint, et notamment grâce à l'exécution du code en place, une quantité très faible de mémoire suffit aux usages considérés jusqu'à présent. Toutefois, les implications de l'IoT, notamment l'augmentation des capacités (et des fonctionnalités) de ces plateformes, ainsi que le besoin de mises à jour et de sécurité font croître les besoins en mémoire de travail.

Mémoire de stockage

La mémoire de stockage la plus répandue à l'heure actuelle pour le domaine de l'embarqué est la mémoire flash. Les technologies EEPROM sont également utilisées : leur usage en opération normale étant restreint à la lecture elles permettent la reprogrammation. Ces deux technologies sont présentées ci dessous. Toutefois il est intéressant de noter que leur usage est globalement contraint par leur caractéristiques et que leur intégration dans le système répond à un besoin de place en terme d'espace de stockage et de non-volatilité. En effet d'une part la mémoire de travail est volatile, d'autre part les technologies employées présentent une implantation peu dense. Le système doit donc être capable de faire persister le code de l'application, ses données statiques, des informations de configuration ou autres entre les périodes d'activité du système. De plus l'intégralité du code d'une application ne tient pas forcément en mémoire de travail. Sous ces contraintes, les systèmes ne chargent généralement pas leur code et l'exécutent en place depuis la mémoire de stockage. Il est à noter que cette application n'est pas envisageable pour des variables dont la valeur sera amenée à changer : en effet les technologies de mémoire de stockage traditionnelles présentent une forte asymétrie lecture/écriture, avec une écriture lente et une endurance en écriture trop faible.

DÉFINITION - *EEPROM* : Electricaly Erasable Read Only Memory (mémoire en lecture seule électriquement effaçable). Crée pour son aspect non volatile et reprogrammable, cette mémoire peut être effacée bit par bit et reprogrammée mais ces opérations sont très lentes par rapport à leur vitesse de lecture.

DÉFINITION - *Flash* : la mémoire flash est une technologie plus récente et issue des technologies EEPROMs. Elle gagne en rapidité d'opération, notamment en écriture par rapport à ces technologies. En effet elle permet de réaliser des écritures par blocs. Toutefois, cette mémoire repose sur un effet de piège d'électron dans une grille flottante d'un transistor. Ce principe de fonctionnement requiert une haute tension d'opération, résultant en une opération d'écriture très coûteuse énergétiquement.

Ces mémoires présentent de plus le problème de ne pas permettre un nombre de cycle d'écriture par cellule mémoire suffisante pour un usage en tant que mémoire de travail.

4.1.1 Limitations

Les limitations de ces technologies mémoires sont nombreuses, mais nous allons les observer principalement du point de vue de l'embarqué contraint.

Tout d'abord en terme de mémoire de travail, la SRAM, bien que suffisamment rapide représente un coût financier très important, en effet son implantation occupe une bonne partie de la puce. Au contraire, la mémoire de stockage la plus répandue, la mémoire flash, présente une bonne densité d'intégration. Toutefois son intégration en elle-même pose des problèmes d'incompatibilité des processus avec ceux permettant la réalisation de la puce du reste du système.

Ces deux facteurs augmentent drastiquement le coût du système résultant. En effet, le coût unitaire des appareils produits dépend de la surface de silicium utilisée et de la complexité du processus mis en place pour sa gravure (nombre de masques et composés chimiques à mettre en place).

Au delà de cet aspect financier, c'est également la consommation énergétique de ces technologies qui impacte le système conçu. En effet dans le cas d'un système embarqué contraint, le fait de

devoir alimenter en permanence la mémoire de travail composée de SRAM est en opposition avec le fonctionnement du système par pics d'activité intense. Ainsi la consommation statique de la mémoire de travail est inutile dans les périodes d'inactivité du système où l'on préférerait qu'elle garde l'information sans alimentation. Mais pour ce même système la mémoire de stockage flash présente des besoins énergétiques très importants notamment dus à la nécessité d'une tension élevée pour l'écriture des données. De plus la granularité d'écriture de la flash, qui lui permet d'atteindre des vitesses de lecture et d'écritures plus rapides que l'accès séquentiel à un disque dur par exemple, augmente cette consommation énergétique. En effet il est n'est pas certain que l'opération considérée concerne tout le bloc qui sera effacé et réécrit. Ces sont ces limitations qui ont poussé la recherche de nouvelles technologies mémoires pour l'embarqué. Pour d'autres domaines comme le HPC (High Performance Computing – calcul haute performance), ces motivations peuvent être interprétées différemment, mais restent proches. En effet, la consommation énergétique d'un data-center et la densité d'implantation des technologies considérées présentent le même genre de limitations.

En terme d'objectif, et pour un système donné, la recherche vise de produire à des coûts raisonnables une technologie mémoire idéale, telle que définie ci-dessous.

4.2 Mémoire idéale

Pour un processeur, la *mémoire idéale* (ou mémoire universelle) présente les caractéristiques suivantes. Une mémoire idéale répond suffisamment vite pour ne pas introduire de délais au niveau du processeur. Dans notre cas, une mémoire répondant aux un cycle remplit ces conditions. La technologie mémoire doit de plus être assez dense pour contenir l'intégralité du programme, ses données et ses variables sans occuper une part importante de la surface de la puce. Étant universelle, cette mémoire sera utilisée à la fois comme mémoire de travail et de stockage, elle doit donc présenter une rétention suffisante pour ce dernier usage. Elle doit néanmoins également présenter une endurance suffisante pour l'usage en tant que mémoire de travail. De plus, d'un point de vue d'intégration à la production du système sur puce, cette mémoire doit être facilement intégrable au processus CMOS et présenter un coût de production raisonnable.

Or traditionnellement les technologies mémoires ont toujours présenté des lacunes sur certains de ces points, qui ont amené la distinction entre mémoire de travail et mémoire de stockage que l'on retrouve dans les systèmes actuels.

La mémoire de travail se rapproche donc de la mémoire idéale en terme de latence du point de vue du processeur là où la mémoire de stockage se rapproche de la mémoire idéale sur l'aspect densité.

4.2.1 Implications pour le domaine de l'embarqué de technologies de mémoire non volatile rapides et peu chères

Pour un système embarqué, quelles seraient les implications d'inclure de la mémoire idéale ? Le domaine de l'embarqué ne propose pas forcément les mêmes réponses que l'informatique débarquée.

En effet, au delà de l'amélioration des performances qui bénéficierait tout autant à un système débarqué qu'à un système embarqué, l'impact sur les contraintes du système serait significatif. Pour l'embarqué haute performance, plus besoin d'utiliser un cache matériel par exemple, mais globalement une mémoire idéale permettrait de simplifier la conception de la puce et d'augmenter son efficience énergétique.

Mais au niveau logiciel les implications peuvent aller plus loin. Nous avons évoqué au chapitre précédent les systèmes à alimentation intermittente, mais dans ce cas, ces systèmes pourraient fonctionner d'une manière intermittente beaucoup moins contraignante. De même l'exécution en place devient beaucoup plus efficace, et chaque accès mémoire consomme moins d'énergie.

Du point de vue des usages finaux, les systèmes embarqués contraints, et plus particulièrement l'IoT, doivent encore vaincre des verrous technologiques limitant leur usages. Disposer d'une telle technologie mémoire permettrait d'augmenter les cas d'usages possibles de ces systèmes. En effet, certaines contraintes environnementales ou économiques ne permettent pas l'utilisation de

ces systèmes dans toutes les situations. Pour certains cas, le calcul demanderait trop d'espace mémoire pour le budget énergétique de la plateforme, ou alors la durée de vie de la batterie ne serait pas assez longue entraînant des coûts de maintenance trop élevés. Au delà de la possibilité d'utiliser ces systèmes sous des contraintes plus fortes, on distingue aussi la possibilité de baisser les coûts de maintenance, et donc de dégager des économies d'échelle importantes.

4.3 Les technologies de NVRAM

Nous allons maintenant présenter les différentes technologies mémoire émergentes que l'on regroupe sous le terme de ENVM (Emergent Non Volatile Memory – mémoires non-volatiles émergentes) ou *NVRAM* (Non Volatile Random Access Memory – mémoire à accès aléatoire non-volatile). C'est l'apparition de ces différentes technologies qui dirige l'apparition d'hétérogénéité mémoire dans les systèmes embarqués au niveau de la mémoire de travail. En effet, les différentes technologies présentent des désavantages, et différentes technologies, ou différentes implantations de la même technologie, peuvent présenter un compromis entre ces désavantages.

Ces technologies mémoires partagent toutefois certaines caractéristiques. Elles présentent toutes une rétention d'information hors tension, qui peut être plus ou moins longue. De plus elles présentent toutes la capacité à être organisées pour l'accès non séquentiel. C'est le choix qui est fait pour une majorité des développements actuels visant l'embarqué.

Cependant, les différentes technologies en elles-mêmes sont très différentes. Les effets physiques envisagés sont très variés, passant du changement de phase d'un matériau à l'effet tunnel de la mécanique quantique [YSS12].

Ces différentes technologies ne présentent pas forcément le même comportement, ni les mêmes possibilité de passage à l'échelle. Toutefois il est à noter que ce sujet de recherche est particulièrement actif et est amené à constamment envisager de nouvelles possibilités.

Nous allons maintenant dresser un inventaire rapide de ces différentes technologies mémoire ainsi que des caractéristiques intéressantes pour cette étude.

4.3.1 Catalogue

Nous dressons donc ici un catalogue rapide des différents types de mémoires dont la technologie est actuellement développement. Il est à noter que ces différentes technologies ne sont pas toutes au même stade de maturité. En effet pour certaines, la littérature ne propose qu'un article prouvant l'effet physique à la base du nœud mémoire, pour d'autres des industriels sont déjà en production, voir dans le cas de la FeRAM, des microcontrôleurs en intègrent déjà [Ins14]. Ainsi, toutes ces technologies ne cohabiteront probablement pas, en tout cas pas sur les mêmes usages [MSCT14].

Aspects importants

Avant de lister les différentes technologies émergentes, nous allons lister ici quels aspects importants pourraient décider de l'adoption de ces technologies, notamment pour le domaine de l'embarqué contraint.

Latence : l'influence des latences mémoire sur le temps d'exécution des systèmes contraints peut être importante. De plus le temps d'exécution de tels systèmes peut influer fortement sur la consommation énergétique. C'est le cas par exemple pour une exécution dite *Normally-Off*, c'est à dire que les longues périodes d'inactivité des systèmes embarqué sont remplacée par une mise hors tension de la plateforme ou d'une grande partie de celle-ci. Nous développerons cet aspect plus loin dans ce chapitre.

La vitesse d'exécution de l'application peut également être une métrique intéressante en soi, et les latences de la mémoire ont un impact important sur celle-ci. On prêtera de plus un intérêt spécifique à l'importance de la possible asymétrie entre les latences pour la lecture et l'écriture d'une donnée. En effet les applications n'accèdent pas leurs objets d'une manière homogène entre lecture et écriture et nous verrons dans nos expériences que cet aspect a un impact sur les mécanismes logiciels tirant parti de l'hétérogénéité mémoire.

Consommation énergétique : du point de vue d'une étude des aspects logiciels telle que la notre, la consommation énergétique des différentes technologies envisagées à différents stade de maturité n'est pas une métrique facile à prendre en compte. De plus il est intéressant de noter que toutes ces technologies consomment moins qu'une écriture en mémoire flash. La consommation énergétique d'une technologie mémoire n'en reste pas moins une donnée primordiale pour la conception d'une plateforme matérielle embarquée.

Endurance : Les problèmes d'endurance pour les mémoires non volatiles comme la mémoire flash sont traditionnellement adressés par une solution de répartition de l'usure gérée par le matériel. Toutefois l'ordre de grandeur du nombre d'écriture par cellule mémoire peut être une limitation pour certains usages, notamment celui qui nous intéresse c'est à dire l'usage en mémoire de travail.

Rétention des données : la capacité de ces nouvelles technologies mémoire à conserver le contenu de la cellule mémoire sans alimentation est une piste de recherche active pour réaliser des systèmes consommant moins d'énergie. Toutefois, cet aspect n'est pas considéré dans cette étude.

D'autres aspect peuvent entrer en ligne de compte pour la sélection d'une technologie, qu'ils soient d'ordre économiques ou techniques. Par exemple certaines technologies mémoire peuvent être plus ou moins facilement intégrés aux processus des fondeurs industriels. Cependant, leur impact sur l'aspect logiciel que nous traitons ici n'est pas direct.

Resistive Random Access Memory (ReRAM)

Nous commençons ce catalogue par la technologie ayant été théorisée dans les années 70 par Léon Chua sous le nom de memristor [Chu71]. Cette théorisation de l'existence d'un élément électronique passif dont l'état dépendrait du courant l'ayant traversé sans nécessiter d'alimentation dérivant des équations de Maxwell.

Ce n'est néanmoins que beaucoup plus tard, que les laboratoires HP Labs ont mis en œuvre un empilement de matériaux présentant les caractéristiques de ce composant théoriques [Wil08]. Toutefois, la technologie résultant de ces découvertes est généralement désignée sous le terme de ReRAM, pour Resistive Random Access Memory (mémoire résistive à accès aléatoire). En effet comme nous l'avons évoqué précédemment les différentes technologies que nous considérons sont basées sur un ensemble très variés d'effets physiques [YSS12] et ainsi certains comportements peuvent être très différents, soumis à des incertitudes et une variabilité différente. Ainsi, les auteurs du premier article sur le memristor considèrent que les différentes technologies étudiées actuellement sont des memristors [Chu11] mais il n'y a pas de consensus sur le sujet.

Dans la classification que nous proposons, nous séparons ainsi les mémoires ReRAMs, basées sur le déplacement de vacances d'oxygène dans un substrat, formant une zone plus conductrices, des CBRAMs présentées après, basées sur un phénomène similaire mais où le déplacement concerne des ions métalliques formant un filament conducteur. Nous suivons ainsi la classification des effets physiques proposée par [YSS12].

Les technologies proposées basées sur ce principe physique présentent donc une couche d'oxyde entre deux électrodes métalliques et le passage du courant au travers va repositionner les vacances d'oxygène. Ainsi, la résistance présentée par la couche d'oxyde diminue et il est possible de mesurer un état haute résistance et un état basse résistance, servant à encoder l'information de la cellule mémoire.

Toutefois ces technologies nécessite une phase de formation, appelée "forming". En effet pour que la cellule soit fonctionnelle il est nécessaire qu'elle soit traversée par un courant bien plus important que son courant d'opération, pour préformer le chemin électrique sur lequel les vacances d'oxygène seront mobile pour faire varier la résistance de la cellule en opération.

Cette opération est un frein à l'intégration de ces technologies au processus de fonte d'un microcontrôleur ou d'un processeur car cette tension qui doit être appliquée pour former la cellule mémoire peut être destructive pour d'autres parties de la puce. De plus les phénomènes mis en jeux présentent souvent une variabilité problématique pour la conception de la mémoire [GYW12].

Elles sont toutefois intéressantes car elles peuvent fonctionner à des tensions d'opération très basses. Il est également à noter qu'on peut également les désigner sous le terme OxRAM.

Conductive Bridge Random Access Memory (CBRAM)

Certains types de ReRAM reposent sur un phénomène physique différent, bien que très proche. La résistance de la cellule va toujours varier en fonction de la présence ou non d'un filament conducteur entre deux électrodes. Mais au contraire des ReRAMs classiques, ce sont des ions métalliques qui vont créer le filament conducteur amenant une variation de la résistance de la cellule mémoire.

Nous les séparons donc car ces deux types de ReRAM ne suivent pas les mêmes pistes de recherche, et la modélisation des effets mis en cause ne sont pas les mêmes, par exemple sur l'aspect de la variabilité [BDFK+14].

Magnetoresistive Random Access Memory (MRAM)

Les technologies basées sur l'effet tunnel, permettant à des électrons de traverser une couche très fine d'isolant entre deux aimants présente elle aussi une résistance variable, cette fois en fonction de l'orientation de l'aimantation de part et d'autre de l'isolant.

De nombreuses technologies ont donc été développée sur ce principe. L'idée générale étant d'avoir un aimant dont la direction de polarisation reste fixe d'un coté de la jonction, tout en faisant varier l'orientation de l'aimantation de l'autre, de manière à créer une résistance différentes au courant traversant la jonction par effet tunnel.

Les premières solutions envisagées souffraient de problème de sélection de la cellule mémoire à inverser car elles nécessitaient des cellules mémoires trop grosses requinquant du même fait trop d'énergie pour leur transition. Ces transitions ont été dans un premier temps réalisées par l'application d'un champ magnétique (Field Assisted Switching). Le problème de sélection a été résolu en chauffant localement la cellule mémoire à inverser (Thermal Assisted Switching). Ainsi chauffée, la cellule nécessite un champ électromagnétique plus faible pour son basculement de valeur, évitant ainsi d'interférer avec les cellules voisines froides.

La technologie récente qui permet aux MRAM d'être concurrentes, notamment pour les usages nécessitant une haute densité d'intégration sont les mémoires STT-MRAM, reposant sur l'effet de Spin Torque Transfert (couple de transfert de spin). Grâce à cet effet, polariser le spin des électrons traversant la cellule suffit à inverser la direction de l'aimant dont la polarisation est variable, facilitant la sélection de la cellule à inverser [AODSK+13].

Les développements récents de cette technologie considèrent le fait d'orienter différemment les aimants par rapport à la jonction pour gagner en performances [AWDSCDN10]. Les mémoires STT-MRAM peuvent donc atteindre des vitesses importantes, mais peuvent encore présenter des problèmes d'endurance limitée et d'intégration du à la variabilité des différentes cellules les unes par rapport aux autres.

Une nouvelle organisation du nœud mémoire semble promettre de passer outre les problèmes d'endurance en évitant de faire traverser le courant permettant le basculement de la cellule, reposant cette fois sur le Spin Orbit Torque (couple d'orbite de spin)[SKR16 ; PJVP+16]. Cette technologie n'est cependant pas à un stade de maturité aussi avancé que la STT-MRAM.

Phase Change Memory (PCM)

La PCM pour mémoire à changement de phase repose sur la capacité d'un matériau particulier (verre de chalcogénure) dont le refroidissement plus ou moins rapide donne lieu à une cristallisation ou présente une forme amorphe, ces deux états influant sur la résistance de la cellule mémoire.

Cette technologie mémoire présente elle aussi des problèmes de variabilité, mais cette fois dans les courants d'opération entre différentes cellules, provoquant une baisse de la durée de vie des cellules étant opérées à un courant trop élevé. Lee et al. [LIMB09] montrent dans leur article de veille sur le sujet que les désavantages de cette technologie sont compensable pour un usage de remplacement de la DRAM. Toutefois, une endurance limitée présente moins d'opportunités pour l'usage en tant que mémoire de travail.

Topological Random Access Memory (TRAM)

Plus récemment et également basée sur le changement d'organisation de l'état solide d'un matériau, Takaura et al. [TOTK+14 ; TOTMKA14] présentent la TRAM. En se basant sur des

materiaux différentes ils arrivent à changer la résistance de la liaison sans avoir besoin d'aller jusqu'au point de fusion du matériau. Toutefois cette technologie n'est pas encore mature.

Nanotube Random Access Memory (NRAM)

Nous évoquons ici également la NRAM basée sur l'usage de nanotubes de carbone. Présentée notamment par Rosendale et al. [RKMH+10] la technologie est prometteuse mais les informations sur le développement des puces mémoires ne sont pas disponibles au public. Cette technologie présente toutefois de bonnes performances en terme de courant d'écriture et de lecture, ainsi qu'une réduction possible de la taille du noeud technologique intéressante pour l'intégration à des systèmes de plus en plus miniaturisés.

Ferroelectric Random Access Memory (FeRAM)

Nous terminons cet inventaire par la FeRAM. En production depuis plusieurs années cette NVRAM a néanmoins présenté très rapidement des limitations en terme de densité et de coûts, du aux matériaux mis en œuvre. Toutefois, Fan et al. [FCW16] ont présenté récemment une alternative en terme de matériaux permettant de surmonter ces limites théoriques notamment en terme de densité imposée par les matériaux précédemment utilisés. Cette technologie repose contrairement à toutes les autres présentées jusqu'à ce point sur le piégeage d'une charge électrique plutôt que d'impliquer une variation de résistance. Cette propriété peut être un avantage dans le développement d'une puce mémoire, les circuits de contrôles autour du noeud mémoire étant proches de ceux habituellement utilisé pour la mémoire flash.

4.3.2 Opportunités

Nous avons brièvement introduit différentes technologies de mémoires non volatiles émergentes. Notre approche ne nous permet cependant pas d'ordonner ces technologies par intérêt pour le domaine de l'embarqué constraint. En effet elles présentent toutes des avantages et des inconvénients, mais ne sont pas toutes aussi avancées en terme de développement. De plus les différents usages qui semblent possibles pour une technologie mémoire dans un domaine peuvent ne pas être valable dans un autre.

Ainsi dans le domaine du calcul haute performances (HPC, High Performances Computing) La ReRAM semble une technologie prometteuse pour remplacer la DRAM de la mémoire principale et la PCM serait plutôt utilisable efficacement pour du stockage [GNWS+19].

Pour le domaine de l'embarqué, et plus spécifiquement de l'embarqué constraint, la MRAM, et les différentes variantes d'implantation du noeud mémoire qu'elle présente semblent intéressantes. En plus de présenter plusieurs variantes qui sont autant de pistes de recherches actives, la MRAM présente des variations possible dans son implantation. Celles-ci permettant de s'adapter aux différentes conditions spécifiques que peuvent représenter l'intégration dans des systèmes embarqués.

De plus l'aspect non volatile de ces technologies mémoire permettent de concevoir des systèmes Ultra Low Power étant en grande majorité éteint pour économiser l'énergie. Cette piste de recherche ouverte par l'usage de checkpoints du programme en mémoire flash [RSF11] a dans un premier temps concerné les systèmes à alimentation intermittente (TPS, Transiently Powered Systems). De nombreux travaux ont depuis exploré le lien entre ces technologies mémoire et l'usage pour les TPS [LJBDG+16 ; AMS14 ; BWMAHBB15 ; BDMRS17].

Mais la possibilité d'intégrer cette technologie directement à l'intérieur de la puce permet aussi d'envisager l'usage d'un processeur non-volatile. Cette piste de recherche vise à réduire la consommation énergétique des plateformes à très faible consommation et dans ce cas l'usage de la MRAM semble prometteur [STSG16].

Les très nombreuses pistes de recherche sur les technologies mémoire à proprement parler, les travaux explorant les usages potentiel de ces technologies mémoires ainsi que les architectures matérielles proposées forment un ensemble extrêmement riche de possibilités pour l'exécution des logiciels embarqués.

Ainsi, du point de vue logiciel, il est difficile de réaliser des expériences sur l'ensemble de ces différentes technologies mémoires. Les axes de recherches logiciels raisonnent donc en règle

général sur un exemple technologique. De plus, les différentes technologies mémoires évoquées dans ce chapitre ne sont en très grande majorité actuellement pas disponibles sur le marché. Ainsi l'étude de leur interaction avec le logiciel repose sur l'utilisation de simulations.

Nous allons donc simuler l'exécution de plateformes dont le sous système mémoire sera composé au moins en partie de ces technologies mémoire. La simulation nous offre ainsi la possibilité de tester deux scénario technologiques éloignés facilement. De plus, le choix de la simulation facilite l'exploration architecturale pour les tailles des mémoires implantées dans les plateformes cibles. Ainsi la simulation des technologies mémoire nous permet d'étudier le comportement du logiciel, non seulement face aux différentes technologies mais également vis à vis de différentes proportions de chaque technologie considérée.

4.3.3 Technologies sélectionnées pour cette étude

Nous allons maintenant présenter quelles technologies nous avons considéré pour les architectures mémoires utilisées dans nos travaux.

Ce projet ayant été construit avec le partenariat de la start-up eVaderis concevant des mémoires MRAM, nous avons pu dialoguer efficacement sur ce point et bénéficier d'informations importantes et de chiffres crédible sur ces technologies. Il est à noter que cette start-up a depuis cessé son activité faute de réussir une levée de fond permettant la mise en production.

Nous avons donc mis en place un scénario technologique reposant sur l'usage de la même technologie mémoire STT-MRAM avec deux variantes créant l'hétérogénéité mémoire. Ainsi, les modules mémoires développés par l'entreprise eVaderis présentent des performances différentes en fonction de leur densité d'implantation. Nous avons donc dans un premier temps considéré une hétérogénéité mémoire similaire à celle présentée par Layer et al. [LJBDG+16]

Pour caractériser ce scénario technologique, nous pouvons le décrire de la manière suivante. Les technologies mémoires considérées ont des performances proches, mais présentent toutefois une hétérogénéité en performances. En effet les latences de lectures varient de 1 à 3 et les latences d'écritures de 2 à 30. Nous désignons dans la suite ce scénario technologique comme le scénario "*MRAM*".

Pour étudier l'impact et la résolution du problème de placement des objets en mémoire hétérogène, nous avons décidé de mettre en œuvre un autre scénario technologique crédible. Pour ce faire nous avons utilisé une hétérogénéité mémoire qui peut être envisagée entre de la SRAM et de la mémoire ReRAM présentée par Intel [JASL+19] Ainsi, l'hétérogénéité mémoire se présente entre une mémoire rapide (1 cycle en lecture et en écriture pour la SRAM) et une technologie qui de prime abord serait plutôt utilisée comme la mémoire flash, sans pour autant tout ses inconvénients, notamment énergétique dont la lecture se fait en 4 cycles sur notre plateforme mais dont l'écriture nécessite environ 4000 cycles. Cet ordre de grandeur de latence mémoire induit souvent l'usage d'un cache matériel, mais nous avons vu que l'utilisation d'un cache n'est pas toujours rentable dans les conditions de l'embarqué contraint, et nous montrerons que suivant l'application il peut ne pas être nécessaire d'en inclure un. Nous désignons dans la suite ce scénario technologique comme le scénario "*ReRAM*".

Nous avons choisi ces technologies mémoires par rapport à d'autres car elles semblaient plus proche du marché que d'autres et présentent deux scénarios technologiques variés. Nous détaillerons plus en détail les caractéristiques des mémoires utilisées au chapitre 6

4.4 Conclusion

Nous avons brièvement présenté dans ce chapitre les différentes technologies mémoires non volatiles émergentes. Ces différentes technologies présentent un intérêt certain pour l'amélioration des systèmes embarqués contraints, notamment dans le domaine de l'IOT. Nous avons également vu qu'elles sont développées dans le cadre du calcul haute performances.

Toutefois, du point de vue des logiciels que nous considérons, l'exécution sur ces différentes mémoires ne se voit qu'en terme de performances, contrairement aux cas où le modèle de programmation se trouve impacté [BDMRS18].

Nous considérons donc des systèmes mémoires hétérogènes pour l'embarqué et allons montrer dans la suite de ces travaux comment améliorer les performances de tels systèmes, en se reposant sur les différences de performances proposées par ces différentes technologies mémoire.

Mais tout d'abord nous dresserons dans le chapitre suivant l'état de l'art des mécanismes logiciels pour la gestion de l'hétérogénéité, notamment ceux applicables à notre domaine d'intérêt

Chapitre 5

Mécanismes logiciels pour la gestion de mémoire hétérogène

Dans les chapitres précédents nous avons discuté des éléments contextuels de notre sujet. Notre objectif est de mettre en place des mécanismes logiciels de gestion de l'hétérogénéité mémoire pour les objets du tas. Toutefois ces mécanismes sont influencés par le matériel de plusieurs manières, et notre étude porte sur l'embarqué constraint. Ainsi d'une part l'architecture et la micro-architecture de la cible contraignent les mécanismes envisageables. D'autre part les technologies mémoire mises en œuvre définissent l'hétérogénéité exposée au logiciel, et donc l'impact sur les performances. Et finalement, pour atteindre ces performances il faut employer des mécanismes logiciels. Nous allons donc maintenant aborder les mécanismes logiciels que la recherche académique à proposé pour tirer profit de l'hétérogénéité mémoire.

Dans un premier temps nous présenterons l'hétérogénéité mémoire des plateformes matérielles qui peuvent donner lieux à ces mécanismes. Nous discuterons également des similitudes et des différences que nous pouvons rencontrer à ce niveau matériel et de leurs implications pour les mécanismes logiciels. Dans un deuxième temps nous présenterons une classification de ces mécanismes logiciels. Nous pourrons ainsi nous positionner par rapport aux autres travaux cités dans ce chapitre.

Nous présenterons dans la suite l'état de l'art de la littérature sur la gestion des mémoires Scratch Pad. La recherche académique a en effet proposé de nombreux mécanismes logiciels de gestion de l'hétérogénéité mémoire dans le domaine de l'embarqué sur ce type d'architecture mémoire. Notre analyse de la littérature montrera que la majorité de ces mécanismes ne sont pas adaptés à la gestion des objets du tas. Certaines approches proposent toutefois une gestion du tas en mémoire hétérogène. Nous préciserons donc notre positionnement vis à vis de ces approches et discuterons de leurs résultats.

Dans un dernier temps nous discuterons des mécanismes logiciels de la gestion mémoire dynamique pour des systèmes à mémoire hétérogène ne concernant pas l'embarqué constraint. Ces mécanismes logiciels présentent de nombreuses similarités avec notre approche. Néanmoins, les limitations matérielles de notre domaine, par exemple l'absence de MMU, ne permettent pas d'appliquer ces approches. Malgré ces limitations ces approches visent la résolution du même problème dans un contexte différent. Nous présenterons donc ces mécanismes logiciels en dégageant les similarités méthodologiques et les différences d'approches imposées par les contraintes du domaine.

5.1 Hétérogénéité mémoire : systèmes cibles

Nous allons donc dans un premier temps lister les différents cas d'hétérogénéité pouvant être rencontrés. Nous opposons donc ici architecture mémoire hétérogène et architecture mémoire hiérarchique comme illustré par la figure 5.1

Hétérogénéité directe Dans le cas nous concernant, l'hétérogénéité est présente entre les mémoires intégrées directement dans la puce. Ainsi se retrouvent au même niveau des mémoires avec des caractéristiques différentes. Elles sont donc directement adressable par le cœur de calcul

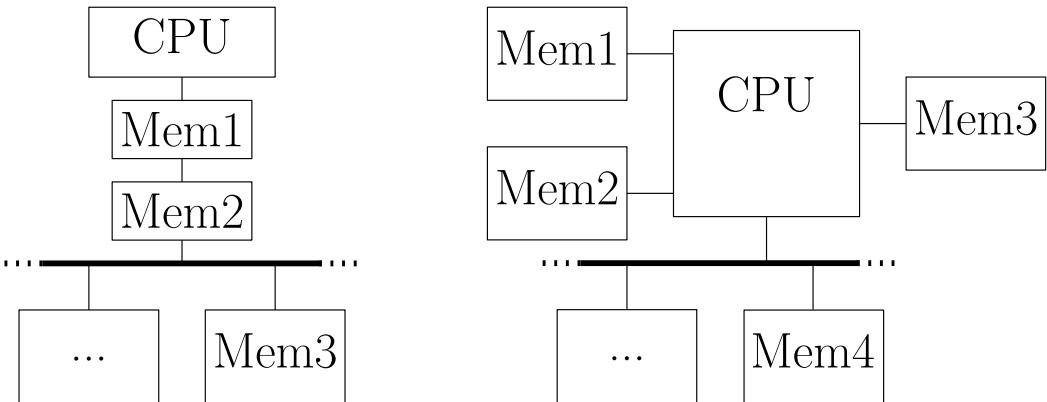


FIGURE 5.1 – Hiérarchie et hétérogénéité mémoire

sur le bus interne de la cible, ou sur une extension de celui ci. Ce cas de figure n'inclue pas de cache matériel, l'accès aux différentes mémoires se fait donc à la granularité de l'octet. Ainsi une mémoire Scratch Pad accessible de la même manière qu'une mémoire principale hors puce présente ce type d'hétérogénéité. C'est donc dans cette catégorie que l'on peut classer la majeure partie de la littérature sur la gestion des mémoires Scratch Pad.

Hétérogénéité indirecte On peut également trouver des cibles matérielles exhibant une hétérogénéité mémoire dans le domaine du calcul haute performance et des architectures NUMA dont il sera question plus loin dans ce chapitre. Contrairement au premier cas d'hétérogénéité ci-dessus, l'hétérogénéité est présente au sein d'une hiérarchie mémoire. Les requêtes du ou des coeurs de calculs passent donc d'abord au travers d'un cache matériel. Cette différence peut avoir un impact significatif : en effet la granularité de l'accès passe à la taille d'une ligne de cache. De plus, les accès aux données du programme changent drastiquement en fonction de la localité. Là où chaque accès arrive jusqu'à la mémoire dans le cas d'une hétérogénéité sur puce, seulement les accès aux adresses non chargées en cache génèrent un accès mémoire dans ce cas.

Hétérogénéité mixte Nous devons également lister des plateformes où une zone mémoire est accédée directement (mémoire Scratch Pad) et un autre type de mémoire est accédé derrière un cache matériel. Certaines de ces architectures sont représentées dans l'embarqué non contraint, où par exemple, la SPM est utilisée en parallèle d'une mémoire principale externe en DRAM.

Dans nos travaux, nous nous concentrerons sur le cas le plus simple de l'hétérogénéité mémoire directe. Ainsi le processeur peut accéder directement aux deux mémoires sans passer par un cache matériel. C'est en effet le cas du domaine de l'embarqué contraint. Il est à noter que les résultats de cette étude sont donc conditionnés par ce choix. En effet les différents domaines concernés n'ont pas les mêmes cibles applicatives. Mais de plus, les différences d'architecture matérielles, et donc d'hétérogénéité exposées, imposent des différences dans les approches pour en tirer parti.

5.1.1 Hétérogénéité mémoire et mécanismes logiciels dans les systèmes embarqués contraints

Comme nous venons de le voir, la plateforme matérielle cible influe sur l'hétérogénéité mémoire présentée au logiciel. Néanmoins, l'aspect le plus marquant pour les mécanismes logiciels d'un tas hétérogène est le support de l'adressage virtuel. Cette abstraction permet pour la gestion mémoire dynamique de déplacer les objets. Ainsi il devient possible de mettre en œuvre des mécanismes plus complexes, plus adaptatifs et plus efficaces basés sur l'utilisation, par exemple, de ramasse-miettes. Nous illustrerons cet aspect plus loin dans ce chapitre.

Mais nous allons expliciter ici quelles spécificités du domaine de l'embarqué contraint interagissent avec le placement des objets du tas.

L'adressage virtuel est réalisé avec le support d'une MMU qui, comme nous l'avons vu au chapitre 3, est un composant matériel coûteux en énergie. Dans le domaine de l'embarqué contraint

il est souvent considéré trop coûteux. La cible matérielle du projet, proche de celle proposée par Layer et al [LJBDG+16], et proposée par notre partenaire industriel n'en inclue pas. De telles cibles matérielles excluent donc de facto l'utilisation de nombreux mécanismes de la gestion de l'hétérogénéité mémoire.

Mais le choix du domaine de l'embarqué constraint à d'autres impacts. Du point de vue logiciel il est généralement acceptable de recourir au profilage pour les applications embarquées contraintes. Au contraire, pour des applications visant une cible matérielle comme un ordinateur de bureau il n'est pas possible de se reposer sur un profilage systématique.

Une autre pratique courante pour le développeur d'applications embarquées est d'utiliser des librairies tierces dont le code source n'est pas disponible. Dans ces conditions nous prétendons qu'il n'est pas envisageable de baser les mécanismes de gestion de la mémoire sur la modification ou même l'annotation du code source. En effet une partie des allocations mémoires peuvent provenir de ces librairies tierces, et si ces objets sont fortement accédés, ne pas pouvoir les prendre en compte dégraderait les performances d'une manière significative.

Nous faisons ainsi le choix de garder l'interface usuelle d'allocation mémoire dynamique, `malloc / free` dans cette étude.

5.2 Mécanismes logiciels : classification et caractéristiques

Nous allons maintenant proposer une classification des différents mécanismes logiciels de la littérature sur la gestion de l'hétérogénéité mémoire. Nous reproduisons ici la classification d'Alam et Horspool [AH16] publiée en 2016. Intitulée «A survey : Software-Managed On-Chip Memories», elle traite de nombreux cas de l'hétérogénéité mémoire et présente les mécanismes logiciels mis en œuvre.

Cet article fait une synthèse des différentes problématiques liées à l'exploitation de l'hétérogénéité mémoire pour les différents types de données à placer. Il est cependant important de noter que dans de nombreux cas les solutions présentées ne sont pas applicables aux cibles matérielles que nous envisageons (pas de caches, pas de MMU).

La classification développée permet toutefois de distinguer précisément quelle donnée va être placée dans la mémoire de quelle manière et sous quelles conditions.

Cette classification sépare dans un premier temps les mécanismes de gestion de la mémoire selon trois axes définis ci-dessous :

- DÉFINITION - Nature de l'allocation : définit si l'allocation de la mémoire est statique ou dynamique durant l'exécution. Dans le cas statique les données dont il est déterminé qu'elle doivent résider en SPM y resteront durant toute l'exécution. Le cas dynamique implique au contraire des chargement / déchargement des données durant l'exécution. Il est à noter que pour la gestion du tas en SPM il n'est pas possible de se retrouver dans le cas statique dans la mesure où l'ordre d'allocation et de désallocation des objets est arbitraire.
- DÉFINITION - Type de l'allocation : permet de définir quelles données du programme sont positionnés par le mécanisme considéré. Nous distinguons ici le code, les variables statiques, la pile et le tas.
- DÉFINITION - Méthode d'allocation : si le placement peut être statique ou dynamique (nature de l'allocation) il n'est pas pour autant forcément décidé durant l'exécution. La méthode d'allocation désigne la manière dont ce placement est décidé, et donc le moment où cette décision est prise. Ainsi nous opposons les mécanismes dont la décision de placement est prise au moment de la compilation, ou avant l'exécution de l'application, aux décisions prises durant l'exécution de l'application et pouvant donc s'adapter à celle-ci.

Nous pouvons donc d'ors et déjà situer notre approche dans cette classification. Notre objectif est de mettre à disposition pour le développeur d'applications embarquées des outils permettant la gestion mémoire dynamique en mémoire hétérogène. Les objets du tas n'étant pas connu au moment de la compilation, et étant alloués et désalloués de manière arbitraire par l'application, nous pouvons donc classer nos travaux de la manière suivante :

- Nature : dynamique
- Type : tas

— Méthode : dynamique

Les auteurs de cet article de veille insistent sur plusieurs points dans l'évaluation des mécanismes considérés. Il est à noter que ces conclusions visent à la généralisation de l'utilisation de mémoire sur puce présentant une hétérogénéité par rapport à la mémoire principale. Cette généralisation vise des systèmes dit "general purpose" (cas général). Ainsi certains axes importants de ce positionnement sont donc en contradiction avec les hypothèses de l'embarqué contraint.

Toutefois, ils considèrent que l'interface exposée au programmeur doit être simple, si possible transparente, conclusions que nous avons également tirée au chapitre 3. Ils insistent de même sur l'importance d'une méthode d'allocation dynamique pour les objets du tas, de manière à pouvoir mieux tirer parti de l'hétérogénéité mémoire. Ils affirment aussi que la littérature dédiée à la gestion des objets du tas n'est pas suffisante. Nous partageons cette conclusion et la développerons à partir de la section 5.4. En effet de nouveaux travaux apportent des solutions dans les cas moins contraints, mais les solutions accessibles dans notre contexte ne sont pas suffisantes. Ainsi nous considérons une plateforme qui ne possède pas de MMU, et ne peut donc pas recourir à l'abstraction de la mémoire virtuelle.

Nous souscrivons également aux deux derniers points avancés par les auteurs, à savoir que le profilage devrait dans l'idéal être réalisé de manière dynamique et que les mécanismes logiciels devraient être flexibles. Ainsi, ils seraient plus facilement adaptables à différentes plateformes, différentes applications. Néanmoins, ces considérations semblent soit inapplicables pour le domaine de l'embarqué contraint, soit prématurées pour nos travaux.

Nous allons maintenant détailler les différentes solutions logicielles proposées dans le domaine de l'embarqué et de l'embarqué contraint pour la gestion de l'hétérogénéité mémoire.

5.3 L'hétérogénéité mémoire dans le domaine de l'embarqué

Le domaine de l'embarqué utilise des architectures variées, qui présentent depuis plus d'une décennie des systèmes mémoire hétérogènes, par opposition à une hiérarchie mémoire.

La recherche académique à donc produit de nombreuses propositions pour l'utilisation de ces architectures mémoire.

Toutefois comme nous l'avons vu au chapitre 3 l'embarqué contraint est un domaine où l'utilisation de l'allocation mémoire dynamique est restreinte car considérée coûteuse. Comme nous le démontrerons plus loin dans ce chapitre la littérature n'a donc pas fourni de solutions satisfaisante pour ce type de données (les objets du tas).

Nous commencerons ici par aborder les mécanismes logiciels mis en place pour l'optimisation de l'accès aux données statiques, puis au code et enfin des données stockées sur la pile. Les mécanismes que nous présentons ne reposent pas sur l'utilisation de modules matériels dédiés exceptés pour quelques un dont cet aspect est discuté. Ainsi, la quasi totalité des plateformes cibles de ces mécanismes ne possèdent pas de MMU ni de cache.

5.3.1 Les données statiques

La littérature tirant partie de la présence d'une hétérogénéité dans la mémoire de travail d'un système embarqué a commencé par s'intéresser au données statiques d'un programme. En effet pour des calculs intensifs l'accès répété à des tableaux ou des structures de données représente un coût significatif en terme de temps d'exécution. De plus ces éléments sont connus à la compilation et l'intensité de leur accès peut être estimée par une analyse statique. Ainsi de nombreuses solution d'analyse de programmes à la compilation ont permis d'utiliser efficacement les architectures Scratch Pad disponibles.

Angiolini et al [ABC03] proposent un mapping statique des données les plus intéressantes à placer dans la mémoire Scratch Pad, et le calculent en temps polynomial. Ce premier mécanisme a donc proposé une allocation de nature et de méthode statique.

Kandemir et al [KRIVKP01] estiment l'intensité des accès à l'intérieur des boucles d'un programme et proposent un placement des données adjoint de l'insertion de code de transfert des-dites données. La méthode reste donc statique mais l'allocation décidée et appliquée est dynamique, impliquant le remplacement de tout ou partie des données durant l'exécution. Cette approche leur permet de réduire de 26% les accès réalisés en mémoire principale.

Nous devons également citer ici Verma et al [VWM04] qui propose une formulation mathématique du problème permettant une résolution de nature dynamique, utilisant une méthode statique insérant du code de chargement et de déchargement à la compilation. Il est à noter que le mécanisme proposé prend en charge non seulement les variables statiques de l'application mais également son code source.

Nous terminons ici par les travaux de Cho et al qui se sont intéressés à l'utilisation de profilage dans la construction des mécanismes de placement des données [CIDYP07]. Leur méthode correspond à l'insertion de code de chargement et de déchargement des données en SPM. Ils formulent le problème sous l'intitulé du "Data Layout Decision Problem", évaluant des placements de donnée spatio-temporelle. Leur travail s'intéresse notamment aux accès non réguliers, ne pouvant être facilement identifiés à l'aide d'une analyse statique, le profilage permettant au contraire de les détecter et de les prendre en compte. Dans une extension de ces travaux, ils s'intéressent plus spécifiquement aux données 'applications multimédia [CIPDAP09]. En se basant sur différents profils pré-calculés sur des jeux de données d'entrées significatifs ils améliorent les performances temporelles et énergétiques des applications. Pour ce faire ils utilisent un module matériel permettant de récupérer durant l'exécution les statistiques d'accès à certaines zones mémoires. Ces informations leur permettent alors de choisir entre les différents profils pré-calculés lequel est le plus intéressant à appliquer. C'est à notre connaissance le seul mécanisme visant les données statiques d'une application dont la méthode d'allocation en SPM soit dynamique. En effet dans le cas des applications multimédia, les données d'entrées peuvent varier d'une manière impliquant que les accès s'y référant ne soient pas efficacement prévisibles.

5.3.2 Le code

Le code, comme nous l'avons vu au chapitre 2, est l'une des données d'un programme les plus stables durant l'exécution. Il permet de réaliser de nombreuses optimisations sous réserve de mettre en place les bons mécanismes logiciels. En effet il est statique et un seul bloc de base est exécuté à la fois dans un système monocœur. Ainsi il est facile de localiser une partie du code en SPM mais sa taille est généralement trop importante pour pouvoir envisager un placement statique. La littérature s'est donc focalisée sur des mécanismes dont le placement, bien que calculé à la compilation, évolue durant l'exécution.

Egger et al. ont proposé de tels travaux. Leur première approche, proposée en 2006 [EKJNLM06] compare les performances d'une architecture contenant une mémoire Scratch Pad à celles présentées par un cache matériel. Ils ne se contentent pas de faire une allocation statique de la mémoire, mais instrumentent le code pour récupérer un profil. Ce profil contient les informations d'utilisation de chaque bloc de base, et en utilisant un solveur ILP les auteurs insèrent du code de chargement / déchargement autour des blocs de base les plus accédés. Ainsi le code d'une boucle par exemple sera localisé en SPM et pourra être exécuté beaucoup plus rapidement sans pour autant nécessiter beaucoup de mémoire.

Ces travaux ont par la suite été développés sur deux axes de travail : les systèmes avec MMU [ELS08] et sans MMU [EKJLMS10]. L'article consacré aux systèmes dotés d'une MMU utilise la mémoire virtuelle pour améliorer ses performances. Au contraire l'article de 2010 améliore le procédé du premier en utilisant la programmation en nombre entiers en se basant sur le profil pour classer les blocs de base du programme en trois catégories :

- code externe : ne sera jamais chargé en SPM
- code épingle : résidera toujours dans la mémoire Scratch Pad.
- code paginé : code pouvant être chargé et déchargé de la SPM à la volée.

Bien que ce mécanisme ne soit pas identique à la pagination mémoire et repose sur du code inséré autour des blocs de base du programme il permet de réaliser ce chargement/déchargement quand nécessaire et ainsi améliorer les performances de l'application.

Toutefois là où le mécanisme repose sur une plateforme utilisant une MMU il dégage 31% de réduction du temps d'exécution, alors que sans le support de ce module il ne dégage qu'une réduction du temps d'exécution de 12%

Ce raffinement des mécanismes mis en place nous semble tendre vers les conclusions que les données d'un programme (tout au moins son code) présente des intensités d'accès différente et qu'une phase de profilage peut donner suffisamment d'information pour décider à priori qu'une partie des données doit se trouver dans la mémoire rapide ou lente. Ce type d'optimisation semble

prometteuse, permettant de rapidement traiter la partie des données ayant le plus d'impact sur les performances.

5.3.3 La pile

Les accès aux données allouées sur la pile présentent une autre spécificité comparée au code de l'application. Les variables de la pile sont allouées et désallouées en fonction de leur portée. Ainsi elles n'occupent la mémoire uniquement pendant qu'elles sont utiles et ne fragmentent pas la mémoire. Les positionner dans une mémoire plus rapide peut permettre des gains importants tout en gardant disponible cette mémoire pour le reste de l'exécution.

La pile est toutefois une structure dynamique et même si cette structure est connue à la compilation, la taille maximale de la pile ne l'est pas forcément, notamment en cas d'appels récursifs.

L'allocation sur la pile se produit à l'entrée dans une fonction où l'espace nécessaire pour les variables de celle-ci est alloué au sommet de la pile. L'espace ainsi alloué est appelé *stack frame* et est désalloué au retour de la fonction.

Avissar et al [ABS02] ont présenté un mécanisme d'allocation de nature statique décidant à la compilation de quelles stack frames allouer en SPM en se basant sur une analyse du code de l'application. Ainsi, si une fonction accède intensément à ses variables locales elle sera allouée en SPM plutôt que dans la mémoire principale.

Dominguez et al [DNB07] ont par la suite proposé un mécanisme d'allocation automatique de la pile, distribuée entre une mémoire Scratch Pad et une mémoire principale. Cette approche requiert une phase de profilage mais est capable, en plus de l'allocation distribuée des stack frames de les charger et décharger de la SPM durant l'exécution. La méthode d'allocation proposée reste toutefois statique. Le profilage permet d'estimer quels niveaux de récursion sont les plus intéressants à placer dans la SPM pour un appel récursif. Les auteurs affirment avoir rencontrés des programmes contenant des fonctions dont les premiers appels récursifs étaient les plus intéressants à placer en SPM ou au contraire les moins intéressants. Ils notent également certains cas où les niveaux de récursion les plus intéressants à allouer en SPM forment un intervalle intermédiaire.

Shrivastava et al [SKL09] ont finalement proposé un mécanisme de nature et de méthode dynamique, allouant les nouvelles stack frame en SPM et évinçant la plus vieille de la pile vers la mémoire principale en cas de manque de place. Leur approche est néanmoins de ce fait sensible à une corruption mémoire. En effet si un pointeur est passé en argument d'un appel de fonction, il est possible qu'il pointe sur une variable locale de l'une des fonctions appelantes. Pour résoudre ce problème un mécanisme d'analyse statique du code et une vérification des arguments d'appels des fonctions sont utilisés. Toutefois ces éléments limitent les gains en terme de temps d'exécution, allongeant significativement le temps d'allocation d'une stack frame.

5.4 Allocation du tas en mémoire Scratch Pad

Nous allons maintenant porter notre attention sur les publications académiques visant le placement des objets du tas en mémoire hétérogène pour l'embarqué contraint.

La majorité des techniques utilisées par la communauté ayant présenté les solutions pour le code et les variables non dynamique sont issue du domaine de la compilation. Dominguez et al. [DUB05] ont donc proposé une extension de ces techniques pour les objets du tas que nous allons discuter. Une autre proposition concerne un allocateur mémoire spécialisé et l'article le plus récent fait une première description du problème tel que nous l'abordons dans cette thèse.

5.4.1 Positionnement

Mück et Fröhlich [MF11] étudient l'impact d'allouer les objets du tas d'une application embarquée dans une SPM ou dans la mémoire principale en se basant sur des annotations du programmeur. L'hétérogénéité prise en compte est donc mixe, mais il est important de noter que si leur plateforme utilise un cache d'instruction, elle ne dispose pas de cache de donnée, exposant ainsi l'hétérogénéité mémoire directement au processeur. Ils démontrent que leur approche peut réduire jusqu'à 30% le temps d'exécution, 13% en moyenne, en plaçant une partie des objets de l'application en SPM. Nous avons discuté du domaine de l'embarqué contraint au chapitre 3 et

nous pensons qu'une approche de résolution du problème nécessitant la modification de code source, notamment celui des librairies tierces n'est pas suffisante. De plus dans le cas présent ces annotations dépendraient de la cible matérielle, nécessitant une adaptation à chaque portage, ce qui semble contre-productif. Ce mécanisme alloue une taille fixe pour la SPM, ainsi suivant les benchmarks entre 3,6% des objets et 100% des objets peuvent être contenus dans la SPM. Nous estimons que cette méthode d'évaluation ne permet qu'une première approche de l'impact de placer les objets du tas en mémoire hétérogène.

5.4.2 Allocateur mémoire spécialisé

McIlroy, Dickman et Sventek [MDS08] proposent un allocateur mémoire dédié à l'utilisation d'une SPM : SMA. Cet allocateur est hautement optimisé pour gérer un tas de faible taille. Le mécanisme logiciel proposé ne propose pas de résolution du problème de placement. Au contraire, ils font la supposition qu'une couche logicielle résout ce problème et ne transmet à l'allocateur qu'un sous ensemble des objets à allouer dans la mémoire Scratch Pad. Leur approche propose donc un allocateur mémoire dynamique adapté à la faible taille de la SPM, utilisant le support matériel d'opérations particulières conjointement avec une double organisation de la mémoire en fonction de la taille des blocs concernés. Ils se comparent à l'allocateur mémoire que nous avons utilisé dans cette étude, DLMalloc, sur des mémoires de 4 et 16 ko. Notre contexte d'étude concernant les NVRAMs pour l'embarqué et ces technologies ayant une densité d'intégration plus élevée, nous ne limitons pas les tailles de SPM que nous explorons aux tailles où cet allocateur est plus efficace que DLMalloc.

5.4.3 Technique à la compilation

Dominguez et al. [DUB05] ont proposé en 2005 une technique de placement des objets du tas entre une SPM et la mémoire principale guidée à la compilation. Ils se placent dans un cas d'hétérogénéité mémoire où la cible matérielle dispose d'une mémoire Scratch Pad et d'une mémoire principale, sans cache matériel. L'hétérogénéité présentée concerne donc une mémoire rapide et une mémoire lente. La taille de la mémoire Scratch Pad est paramétrée à 5% de l'espace requis par le programme pour ses variables globales, sa pile et son tas.

Leur approche repose sur l'utilisation de nombreux tas. Ainsi leur système logiciel disposera d'autant de tas dans la mémoire Scratch Pad que l'application possède de sites d'allocation. De plus ils utilisent un tas en mémoire principale pour les objets ne tenant pas dans le tas correspondant à leur site d'allocation. Cette approche est la seule à proposer une solution calculée durant la compilation au problème d'allocation des objets du tas en mémoire hétérogène. Chaque site d'allocation du programme se voit donc attribuer un tas dont la taille maximale est définie en fonction d'une analyse statique du code et de la taille de la SPM cible. À l'allocation d'un nouvel objet, ce tas spécifique au site d'allocation est parcouru et l'objet est alloué dedans si assez d'espace contigu est disponible. Sinon, l'objet est alloué sur un tas commun à tout les sites d'allocation situé en mémoire lente. L'approche du problème à la compilation permet aux auteurs d'insérer du code chargeant et déchargeant en SPM les différents tas utilisés dans les différentes régions du programme.

Dans ces conditions ils présentent des gains en performance en se comparant par rapport à un tas situé entièrement en mémoire lente. Ils présentent une amélioration du temps d'exécution de 35% pour les technologies mémoires suivantes :

- SPM : SRAM répondant en un cycle
- Mémoire principale : DRAM répondant en 20 cycles

Il nous semble toutefois que la méthode proposée n'est pas efficace dans des cas d'utilisations intenses de l'allocation mémoire dynamique. Leurs résultats sont obtenus sur 5 benchmarks dont seulement 4 allouent plus de 5 objets, le dernier en allouant environ 150. La somme des tailles allouées par ces applications ne dépasse jamais 20 ko (4,4 ko en moyenne) et les objets sont alloués par au plus 5 sites d'allocations. Nous expliquerons en détail au 7.1 l'impact sur l'algorithme d'allocation et ses performances d'avoir à faire à plusieurs tas gérés indépendamment dont certains de faible taille. Puis nous démontrerons expérimentalement au 8.1 l'impact que devrait avoir un usage significatif de l'allocation mémoire dynamique sur les performances de l'allocateur DLMalloc non modifié. Dominguez et al. affirment utiliser cet algorithme non modifié.

Nous pensons donc que cette méthode, applicable efficacement à des profils d'allocation simples présenterait de mauvaises performances dans le cas contraire. De plus les applications utilisant intensément l'allocation mémoire dynamique peuvent présenter un nombre de sites d'allocation beaucoup plus important (jusqu'à une soixantaine dans nos applications). Ainsi, le surcoût en espace requis pour les structures de donnée serait considérablement augmenté, ainsi que la pression sur les nombreux tas qui seraient remplis rapidement et verraient leurs performances diminuer drastiquement.

5.4.4 Discussion sur l'évaluation de l'allocation mémoire dynamique en mémoire hétérogène

La gestion mémoire dynamique hétérogène pour l'embarqué constraint est un sujet de recherche présentant plusieurs difficultés méthodologiques. L'une de ces difficultés est la faible disponibilité de benchmarks ayant un usage significatif de l'allocation mémoire dynamique dans le domaine de l'embarqué. Comme développé au chapitre 2 la gestion mémoire dynamique est un mécanisme coûteux et dans le domaine de l'embarqué constraint le programmeur d'application essaye de s'en passer. Ainsi les cas où elle est utilisée sont des cas réellement dynamiques. Nous pensons qu'il n'est pas non plus raisonnable d'utiliser des benchmarks standard de la littérature de la gestion des SPM en allouant sur le tas des variables globales ou des variables qui auraient du être allouées sur la pile. Au delà d'être incohérente avec les pratiques du domaine ces modifications génèrent des profils d'allocation non dynamiques, ne permettant pas une exploitation des caractéristiques des objets du tas.

De même sur les trois articles cités dans cette section, seul celui de McIlroy réalise plusieurs exécutions avec des traces d'allocations différentes. Or la taille et le nombre des objets ne sont pas connus avant l'exécution. Le fait d'exécuter un seul jeu de donnée d'entrée pour des applications utilisant l'allocation mémoire dynamique implique de courir le risque de tirer parti de régularités spécifiques à cette exécution qui ne pourront donc pas être exploitées dans le cas général.

5.4.5 Mécanismes logiciels visant l'utilisation des NVRAMs

Nous avons jusqu'à présent focalisé notre état de l'art sur les mécanismes logiciels pouvant être appliqués au domaine de l'embarqué constraint. Nous allons ici nous focaliser sur les mécanismes logiciels reposant sur une mémoire Scratch Pad mais faisant intervenir des technologies mémoires NVRAMs. En effet, ces technologies mémoire sont l'une des motivations de ces travaux.

La littérature sur le sujet présente des mécanismes visant le placement du code ou des variables statiques d'un programme. Les mécanismes logiciels mis en place reposent sur la littérature déjà présentée plus haut, mais plusieurs aspects importants diffèrent. Li et al [LZH+12] s'intéresse à la formulation du problème d'allocation des variables du programme sous forme d'ILP et sous forme de problème de coloration de graphe. Une formulation *ILP*, pour Integer Linear Programming, programmation linéaire en nombre entier en français, est une manière de décrire un problème sous forme de contraintes linéaires de variables entières avec une fonction objectif à minimiser. Cette approche de résolution hors ligne du problème n'est pas novatrice en soi, mais l'architecture considérée inclue une mémoire Scratch Pad hybride contenant un banc de SRAM et un banc de PCM en plus de la mémoire principale. Ainsi, par exemple sur l'aspect énergétique le coût d'écriture d'une donnée dans le banc de PCM peut être plus élevé que le coût d'écriture en mémoire principale.

Ainsi l'adoption de ces différentes technologies mémoires permet de considérer l'hétérogénéité mémoire plus largement qu'une juxtaposition de deux mémoires. Nous avons malgré tout limité le périmètre de cette étude à l'exploration de cette configuration (une mémoire rapide contre une mémoire lente) pour des raisons pratiques. C'est toutefois dans un sens plus large que nous entendons le terme d'hétérogénéité mémoire.

Par exemple Rodriguez et al [RTK14] présente une SPM adjointe d'un mécanisme logiciel pour en tirer partie en diminuant les coûts énergétiques des écritures mémoires. La mémoire est divisée en différents bancs, toutes de technologies identiques (MRAM), mais chaque banc écrit ses données avec une énergie d'écriture différente. Comme nous l'avons vu au chapitre précédent cette technologie présente un compromis entre énergie d'écriture et temps de rétention. Ainsi, en fonction de leur durée de vie les variables sont placées dans des bancs différents, permettant

d'optimiser l'énergie utilisée pour une variable n'ayant qu'une faible durée de vie. La durée de vie des variables est considérée en calculant le WCET entre leurs utilisations.

Hu et al ont proposé dans des travaux récents sur l'utilisation d'une SPM incluant de la PCM et de la SRAM. Leur approche implique une résolution du problème à la compilation mais présente une nature dynamique durant l'exécution grâce à l'insertion de code déplaçant les données [HXZTS11 ; HXZTS13 ; HZXTS14]. Leur algorithme d'allocation des données en SPM essaye notamment de maximiser la durée de vie de la mémoire non volatile présente en allouant les objets ayant le plus d'accès en écriture dans la SRAM, tout en essayant de minimiser le temps d'exécution et l'énergie consommée. Ils s'intéressent dans leur dernier article cité ici à ce mêmes conditions dans le cadre du multicœur.

Nous allons maintenant nous intéresser aux travaux connexes aux nôtres, présentant des mécanismes logiciels pour la gestion de mémoire hétérogène pour d'autres systèmes que ceux de l'embarqué constraint.

5.5 Travaux connexes

5.5.1 Embarqué non constraint

Le domaine de l'embarqué constraint est assez restrictif car du aux contraintes matérielles il limite l'utilisation à des mécanismes logiciels simples. Néanmoins la littérature sur le sujet propose de nombreuses avancées en dehors de notre domaine. Nous allons donc parler ici des travaux décrivant des mécanismes qui approchent le problème de l'utilisation d'un tas en mémoire hétérogène hors de l'embarqué constraint.

Tout d'abord, nous devons ajouter à l'état de l'art présenté dans ce chapitre le travail de Francesco et al [FMABCM04] visant le domaine de l'embarqué non constraint. Les auteurs proposent une gestion à l'exécution d'une SPM pour allouer des données statiques et les objets du tas. Ils sélectionnent les structures de données les plus accédées et les sites d'allocation générant les objets les plus accédés et les allouent de manière dynamique. Leur solution permet également d'allouer dynamiquement les données statiques. cette approche requiert de nombreux transferts de données, les auteurs proposent donc une intégration matérielle, ajoutant un *DMA* dédié permettant de charger et décharger des données en SPM plus efficacement. Un module DMA, pour "Direct Memory Access" – accès mémoire direct, est un module permettant de réaliser des transferts mémoire de manière efficace, sans intervention du processeur après la configuration du transfert. Leur cible matérielle est une plateforme dotée d'une MMU, d'un cache matériel et d'une SPM, ainsi qu'une mémoire principale hors de la puce. Bien que les contraintes matérielles de notre étude ne permettent pas d'appliquer ces techniques les réponses qu'ils apportent au problème reposent sur l'étude de l'intensité préalable des accès aux objets guidant une décision prise à l'exécution. Nous pensons que cette approche, utilisée également par Cho et al [CIDYP07] que nous avons abordé pour l'allocation des données statiques en SPM est prometteuse dans le domaine de l'embarqué constraint.

Comme le souligne la veille évoquée au début de chapitre, le profilage à l'exécution permet de répondre de la manière la plus dynamique au problème. Toutefois elle est souvent inenvisageable pour un système constraint. Cho et al [CPIDAP09] lèvent cette contrainte en utilisant un module matériel dédié effectuant un profilage léger à l'exécution de manière à pouvoir choisir parmi plusieurs profils calculés préalablement.

5.5.2 Haute performance et big-data

Hors du contexte de l'embarqué, des solutions de profilage dynamique adjointes de déplacement des objets donnent de bons résultats. Dans le contexte du calcul haute performances (*HPC*) notamment, les ressources de calcul doivent être maîtrisées, mais contrairement au domaine de l'embarqué, les architectures matérielles incluent de nombreux modules d'optimisation permettant une exécution beaucoup plus rapide des programmes. Narayan et al [NZANC18] proposent une solution capable de profiler durant l'exécution les objets alloués de manière à les classifier et à les répartir entre trois mémoires différentes. Une mémoire accueille les objets bénéficiant d'une faible latence d'accès, une autre vise ceux dont la bande passante d'accès doit être élevée et une troisième mémoire basse consommation accueille les objets moins accédés.

Akram et al [ASME18] envisagent l'hétérogénéité de la mémoire principale des datacenters. Pour remplacer une partie de la DRAM par de la PCM, beaucoup plus dense et sans consommation statique comme toute les mémoires non volatiles, ils proposent un ramasse miette limitant les écritures dans la PCM. En effet celle ci a une endurance limitée et est plus lente à l'écriture. Au contraire de la gestion de la mémoire présentée dans ce travail, ils utilisent une gestion mémoire automatique, basée sur un ramasse miette, capable de déterminer sans appel du programmeur quels blocs mémoire ne seront plus accédé par le programme. Plus spécifiquement ils utilisent un ramasse miette générationnel. Ce type de ramasse miette repose sur l'hypothèse de mortalité infantile des objets ou, autrement dit, un objet ayant une durée de vie élevée à moins de chance d'être désalloué rapidement qu'un objet alloué récemment. Pour exploiter cette hypothèse un ramasse miette générationnel alloue les objets dans une nurserie, dans laquelle il vérifie régulièrement quels objets sont encore en vie. Puis il déplace ces objets dans une zone où les objets sont collectés moins souvent. Les auteurs utilisent donc un ramasse-miettes générationnel dont la nurserie est en DRAM car ils observent que plus de 70% des écritures y sont réalisées. Puis les survivants sont placés dans une zone d'observation en DRAM où leurs écritures sont profilées dynamiquement. finalement seuls les objets peu écrits sont déplacés vers la PCM. Cette approche leur permet de multiplier la durée de vie des zones mémoires en PCM par 11 en réduisant le nombre d'écritures y étant réalisées de 91%. Dans la suite de leur travaux ils utilisent toutefois un profilage préalable plutôt que de profiler une partie des objets durant l'exécution. Cette approche permet d'améliorer leurs résultats, réduisant le nombres d'écritures dans la PCM d'un tiers supplémentaire [ASME19].

Kannan et al [KGGS18] proposent une solution intégrée à l'OS permettant d'effectuer cette gestion en dessous de machines virtuelles. Finalement nous pouvons citer les travaux de ces domaines proposant de nouvelles approches non pas pour l'allocation des objets du tas mais pour les données statiques. Zhao et al [ZQCG18] proposent une solution multi critère basé sur un algorithme de programmation dynamique et Qiu et al proposent un algorithme génétique pour le calcul de l'allocation des données.

Ces différents travaux, adressant le problème dans des conditions moins contraintes, permettent de mettre en avant l'importance de prendre en compte différents aspects de performance pour ces mémoires.

On peut aussi noter que même dans des contextes hautes performances le profilage en ligne reste une technique coûteuse à mettre en œuvre pour une utilisation intensive du tas.

5.5.3 NUMA

NUMA, pour Nonuniform Memory Access (accès mémoire non uniformes) et une abstraction utilisée pour le calcul sur machine multicoeurs. Dans ces architectures, chaque unité de calcul a un contrôleur mémoire associé. la non uniformité des accès émerge de la distance entre chaque cœur et les différentes mémoires contenant les données. Malgré tout dans le cas général la mémoire est considérée uniforme [WIL17]. L'article de veille sur les mécanismes NUMA de Diener et al [DCANK17] identifie la prise en compte de l'hétérogénéité des mémoires comme une piste de recherche prometteuse.

L'hétérogénéité que peut présenter ces systèmes consiste donc en des technologies mémoires différentes, réparties à des distances différentes des unités de calcul. Ainsi, reposant sur l'abstraction de la mémoire virtuelle c'est l'allocation des pages physiques des différentes mémoires qui peut tirer parti de leurs différences. Nous nous intéressons donc ici aux mécanismes NUMA destinés à tirer partie de cette hétérogénéité.

Williams et al [WIL17] affirme que la distance NUMA utilisée classiquement ne permet pas de tirer parti d'autres chose que des différences de latence entre les mémoires. Dans la suite de leur travaux [WILL18], les auteurs proposent une interface permettant de grouper les objets en plusieurs tas séparés en fonction de leur comportement dans l'application. Ce groupement est toutefois laissé au programmeur d'application. Ainsi, le problème d'allouer ou de déplacer une page sans savoir si les objets qu'elle contient bénéficieront tous de ce déplacement peut être contourné. En effet dans ces conditions les objets similaires se trouvent groupés sur les mêmes pages de l'adressage virtuel. Par exemple, les objets bénéficiant grandement d'une mémoire à haute bande passante puis pouvant être déplacé après se retrouveront alloués dans un seul tas, ne contenant pas d'objets au comportement opposé. Leur approche permet de déplacer les différentes

pages d'un tas particulier, le nombre de tas nécessaire pour l'application étant également délégué au programmeur.

Pour guider le choix du programmeur, Wen et al [WCLL18] proposent un outil de profilage adapté aux architectures NUMA présentant de l'hétérogénéité, ProfDP. Leur outil permet d'analyser l'influence sur le temps d'exécution du programme d'allouer les objets de chaque site d'allocation dans une mémoire à haute bande passante ou dans un mémoire à faible latence comparée à la mémoire principale (de chaque noeud).

Ces approches montrent de bons résultats mais ne considèrent pas de limitation stricte des tailles des différentes mémoires. Par exemple les trois sites d'allocations alloués en mémoire rapide pour l'une de leurs applications correspondent à 37% de l'empreinte mémoire du tas de l'application. Ainsi, contrairement aux approches du domaine de l'embarqué contraint la taille de la mémoire rapide peut représenter une part plus importante des besoins mémoire de l'application. D'une part, l'utilisation de peu de mémoire rapide permet d'en laisser plus disponibles pour d'autres applications s'exécutant sur le même noeud. Mais d'autre part des mécanismes comme la pagination permettent une gestion plus efficace de la mémoire rapide.

Nous référençons également ici les articles de Shrivastava et al [BS10] et Lin et al [LLCS19]. Contrairement aux travaux ci dessus, ils se concentrent sur une mémoire locale de taille limitée. Toutefois, les architectures considérées ne permettent pas d'accéder directement à la mémoire principale. Dans ce cas là, la SPM est intégrée dans une hiérarchie mémoire, à la place d'un cache matériel, et l'hétérogénéité mémoire n'est pas exposée au programmeur.

5.6 Conclusion

Nous avons proposé dans ce chapitre un ensemble de mécanismes logiciels pouvant être utilisés pour tirer profit d'un système mémoire hétérogène. Tout en détaillant notre étude de la littérature pour les domaines utilisant des mécanismes applicables à nos conditions nous avons mis en évidence que pour le domaine de l'embarqué contraint il n'existe pas de solution satisfaisante pour la gestion du tas en mémoire hétérogène. Toutefois des mécanismes logiciels tirant parti de l'hétérogénéité mémoire pour placer judicieusement les objets du tas existent dans de nombreux autres domaines comme nous l'avons illustré. Ces approches reposent sur l'utilisation du profilage pour évaluer les sites d'allocations, choix que nous avons également fait.

La suite de ce manuscrit présente donc le problème de placement des objets du tas pour l'embarqué contraint et l'étude de sa résolution.

Deuxième partie

Contributions

Chapitre 6

Positionnement et approche

La première partie de ce manuscrit décrit ce qu'est le problème de placement en mémoire hétérogène. Nous avons également montré que les objets du tas ne peuvent être placés en utilisant les mêmes méthodes.

Nous allons dans cette deuxième partie formuler mathématiquement ce problème et proposer une méthodologie de résolution. Par la suite nous étudierons les caractéristiques des applications cibles. Puis nous évaluerons l'impact que peut avoir la résolution de ce problème sur les performances de l'application. À la suite de quoi nous pourrons proposer une méthode de résolution efficace.

Pour commencer, ce chapitre formalise le problème de placement mémoire pour le tas, ainsi que les hypothèses de notre cas d'étude. Ainsi nous dégagerons une méthodologie d'évaluation adaptée pour mesurer l'impact du placement mémoire sur les performances de l'application. Nous illustrerons ensuite le problème formulé à l'aide d'un exemple simpliste qui nous permettra d'expliquer les concepts utilisés. La formulation du problème et la réduction au cas d'étude nous permettront de décrire la méthodologie d'évaluation des solutions au problème de placement à la fin de ce chapitre.

6.1 Problème de placement pour le tas

6.1.1 Placement et hétérogénéité mémoire

Nous avons vu au chapitre 5 que le placement des données d'un programme en mémoire influençait les performances de celui-ci. De la même manière que pour le code ou les variables locales, les objets du tas sont accédés de manière hétérogène par le programme. Nous parlerons donc d'objets d'une application comme étant "chauds" ou "froids" en fonction de l'intensité de leurs accès. Nous reviendrons plus formellement sur cette notion dans la suite. Il est à noter que cette description doit tenir compte de la taille des objets. En effet un objet deux fois plus gros qu'un autre à nombres d'accès égaux sera moins chaud.

Le nombre, la taille et l'ordre d'allocation et de désallocation des objets sont inconnus avant l'exécution et dépendent du jeu de données d'entrée du programme. Nous avons défini l'ensemble de ces caractéristiques des objets alloués pour une exécution comme étant le profil d'allocation de cette application pour un jeu de données. De plus les accès d'un programme aux différents objets qu'il alloue varient en fonction de son jeu de données d'entrée. On définit donc le profil d'accès au tas d'un programme de la manière suivante :

DÉFINITION - *profil d'accès au tas* : caractéristiques d'accès en lecture et en écriture d'une application à chacun des objets alloués lors d'une exécution. Un profil d'accès au tas est spécifique à un profil d'allocation et donc à l'exécution d'une application particulière pour un jeu de données d'entrée.

Cette approche de la chaleur d'un objet dans une exécution diffère de l'approche classique envisagée dans une hiérarchie mémoire où un objet "chaud" doit être gardé en cache, mais où la chaleur de l'objet se réfère à avec la fréquence à laquelle il est accédé plutôt qu'à son nombre d'accès absolu. En effet l'opération coûteuse pour accéder un objet dans ce cas consiste à le charger en cache, mais le nombre d'accès suite à ce chargement est peu influent sur les performances du programme car en mémoire rapide. Un objet peu accédé mais à intervalle régulier sera plus

intéressant à garder en cache tout au long de l'exécution qu'un objet beaucoup plus accédé sur une courte période de temps.

Or dans notre cas, l'accès aux bancs mémoires est direct et sans passage par un cache matériel. Il en résulte que l'importance d'un objet se rapporte au nombre de fois où il est accédé, la répartition temporelle de ces accès n'ayant pas d'impact..

Dans ces conditions l'importance d'un objet dépend :

- du profil d'accès de l'application
- des technologies mémoires mises en œuvre
- du critère d'évaluation choisi

En effet suivant l'aspect que l'on cherche à optimiser les objets importants peuvent ne pas être les mêmes. Le placement fait à l'allocation de l'objet influence le reste de l'exécution en fonction de son profil d'accès et peut affecter la durée de vie du système. Soit en faisant varier l'efficience énergétique de celui-ci, qui influe sur l'autonomie d'un système sur batterie, soit en plaçant des objets trop chauds dans une mémoire ayant une faible endurance limitant donc sa durée de vie. Suivent quelques scénarios illustrant l'influence d'un choix de placement sur un système.

Dans le cas d'un objet quelconque pouvant être placé dans une mémoire rapide ou dans une mémoire lente, le nombre d'accès à cet objet va influencer le temps d'exécution de l'application. Pour optimiser ce critère il conviendra de placer les objets les plus chauds dans la mémoire rapide.

Un objet uniquement accédé en lecture après son initialisation peut augmenter la durée de vie du système si elle est limitée par l'endurance de la mémoire. Dans ce cas, placer l'objet en question dans la mémoire à faible endurance permet de laisser de la place dans la mémoire plus endurante pour d'autres objets plus accédés en écriture.

Nous pouvons aussi tirer parti de l'asymétrie d'accès lecture/écriture aux objets pour guider leur placement dans des bancs mémoire présentant une asymétrie de latence.

Pour ce qui est d'un système embarqué constraint, une stratégie envisageable consiste à minimiser le coût énergétique des accès mémoires de l'application sur son exécution. Accélérer l'exécution par des choix de placement de manière à pouvoir passer le système dans un mode faible consommation est une autre approche pouvant être appliquée. Dans ces conditions, une exécution plus rapide permet de passer plus de temps dans ce mode d'économie d'énergie de donc *in fine* de moins consommer

6.1.2 Formulation

Nous avons vu que le problème de placement des données d'un programme concerne le choix d'un banc mémoire de destination entre plusieurs, présentant des performances différentes.

Nous avons également vu que les solutions proposées au problème de placement pour le code, les variables globales et les variables locales permettent d'améliorer significativement les performances de l'application. Ces performances pouvant au maximum égaler celles de la même application sur une architecture mémoire composée uniquement de la technologie mémoire la plus avantageuse suivant le critère ciblé.

Toutefois la mise en place de ces solutions requiert de disposer d'informations sur les données cibles au moment de la compilation, de manière à pouvoir prédire quelle donnée sera la plus intéressante à placer dans l'un ou l'autre des bancs. Ces informations peuvent être récoltées lors d'une phase de profilage mais doivent rester valides d'une exécution à une autre. Or nous avons vu au chapitre 2 que ces informations ne sont pas disponible avant l'exécution pour les objets alloués dynamiquement sur le tas.

On peut donc formuler le problème de placement des objets du tas en mémoire hétérogène de la manière suivante :

Pour une architecture mémoire hétérogène, une application et un jeu de données d'entrée, le *problème de placement* consiste à décider pour chaque objet alloué, dans quel banc mémoire il sera placé. On définit le placement mémoire comme la solution apporté à ce problème.

DÉFINITION - *placement mémoire* : ensemble de décisions répartissant les différents objets d'un profil d'allocation entre les bancs d'une architecture mémoire. D'une manière plus formelle, un placement mémoire est une fonction des objets vers les zones mémoires dédiées aux tas.

L'application d'une solution de placement ou d'une autre va influencer les performances de l'application à chaque accès qu'elle va faire sur un objet placé dans l'un des bancs mémoire de

l'architecture cible. Nous en concluons donc que de la même manière que l'allocation dynamique des objets impose une résolution du problème durant l'exécution, la résolution du problème de placement des objets du tas nécessite une résolution à l'exécution.

Contexte de résolution

Nous souhaitons apporter une méthode de résolution efficace au problème de placement que nous venons de décrire. Or nous avons vu au 3.2 que le domaine de l'embarqué influe grandement sur le design de ses architectures logicielles. En effet les logiciels du domaine de l'embarqué constraint se caractérisent par la réutilisation de code hérité et l'utilisation de librairies tierce partie dont le code n'est pas forcément accessible ou modifiable. Nous choisissons donc de ne pas modifier l'API standard d'allocation mémoire dynamique.

De plus le développeur d'application ne maîtrise pas forcément ou facilement plusieurs aspects essentiels à la résolution du problème de placement. Un développeur applicatif n'a pas forcément une connaissance approfondie des mécanismes sous-jacents à l'allocation mémoire dynamique. Or nous verrons dans les chapitres suivant qu'ils peuvent avoir un impact significatif sur le placement mémoire. De plus une bonne solution au problème de placement nécessite de bien connaître et comprendre le profil d'accès aux objets alloués par son application, mais également par les librairies qu'elle utilise. Enfin, puisque la solution de placement dépend des technologies mémoires utilisées et de l'architecture mémoire cible, il semble intéressant pour le développeur d'applications embarquées de pouvoir déléguer la résolution du problème de placement à une couche logicielle. En effet cette couche logicielle pourra s'adapter à l'architecture et aux caractéristiques technologiques plutôt que de lui imposer de résoudre ce problème pour chaque cible de son application.

Nous affirmons pour ces raisons que la solution que nous souhaitons apporter ne doit pas nécessiter d'adaptation du code source applicatif par le programmeur d'application embarquées.

6.2 Cas d'étude

De manière à apporter une solution au problème énoncé, dans le contexte définit précédemment nous discutons ici des hypothèses et restrictions qui découlent de ce contexte.

Comme dit ci-dessus, nous visons une solution permettant de ne pas modifier le code applicatif et ne nécessitant pas de connaissances particulières du problème de placement ou des détails architecturaux de la part du programmeur applicatif embarqué.

Mais la résolution du problème de placement mémoire en elle-même est liée au domaine de l'embarqué constraint. En effet les architectures matérielles du domaine sont spécifiques et n'embarquent généralement ni cache matériel ni unité de la gestion de la mémoire. Ces spécificités matérielles des systèmes considérés ont chacune un impact considérable :

Pas de MMU : la solution au problème de placement doit être formulée à l'allocation, car l'allocateur doit choisir l'adresse à laquelle l'objet est alloué. Contrairement à un système supportant la traduction d'adresse, un objet ne peut pas être déplacé d'un tas à un autre en fonction de l'exécution de l'application.

Pas de cache matériel : nous considérons une architecture matérielle où toutes les zones mémoire sont exposées sur le bus ou au travers d'une interface mémoire directe. Ajouter un cache matériel change non seulement la latence d'accès aux objets suivant si ils sont chargés dans le cache ou non mais de plus les accès aux bancs mémoire derrière le cache changent de granularité, passant d'une granularité de la taille d'un accès mémoire du processeur à celui d'une ligne de cache. Ainsi les résultats présentés dans ces travaux ne peuvent pas être transposés tels quels à d'autres systèmes dotés d'un cache matériel sans investigation supplémentaires.

6.2.1 Comparaison avec l'approche NUMA

Comme nous l'avons vu au chapitre 5 le problème d'allocation des objets dynamique entre les mémoires des différents coeurs d'une architecture NUMA présente des similarités avec le problème de placement des objets du tas. De nombreuses approches méthodologiques de ce domaine sont

inapplicables car elles reposent sur le mapping de fils d'exécution ou le déplacement d'objets à posteriori. Toutefois, les méthodes de mapping de données mises en œuvre peuvent être utilisées pour résoudre le problème de placement que nous considérons.

Cependant le contexte de résolution de notre problème diffère de l'approche de l'allocation sur machine NUMA sur plusieurs points. Tout d'abord le matériel, les noeuds de calculs reposant sur une hiérarchie mémoire, ne présentent pas d'hétérogénéité la plupart du temps et disposant d'une MMU. D'un autre côté les logiciels cibles ne sont pas les mêmes et l'étude de leur utilisation de la mémoire dynamique sort du cadre de cette thèse. Enfin le domaine et ses contraintes sont différentes, en effet si une approche courante pour l'approche NUMA est de modifier le code source de l'application, nous avons vu au chapitre 3 que cette approche est contre productive dans le domaine de l'embarqué constraint. Il est également à noter que la méthodologie NUMA se permet de reposer sur la présence d'un système d'exploitation ce qui n'est pas le cas de notre étude.

6.2.2 ISA et architecture mémoire

Nous avons vu l'importance pour la résolution du problème de placement des accès mémoire d'une application. Il va sans dire que l'architecture mémoire considérée a un impact sur la résolution du problème. Mais quel importance donner au choix du jeu d'instruction considéré ?

Nous faisons ici l'hypothèse que les différentes architectures de jeu d'instruction (ISA) utilisées par l'embarqué constraint n'influent pas de manière significative sur la résolution du problème de placement. En effet, nous nous intéressons au problème de placement des objets du tas uniquement. Or le nombre d'accès nécessaire à ces objets lors d'une exécution dépend plus de l'algorithme générant ces accès que de l'ISA du processeur les implémentant. De plus nos architectures cibles exécutent dans l'ordre et exécutent donc une instruction par cycle en dehors des attentes processeur.

Au contraire l'architecture du système mémoire a un impact important. En effet, différents programmes ayant des profils d'allocation et d'accès au tas différents il est peu probable qu'ils exposent les mêmes résultats pour la résolution du problème de placement. Ainsi deux programmes différents auront besoin de plus ou moins de mémoire (par exemple avec une faible latence mémoire) pour arriver au même impact sur leurs performances (en terme de temps d'exécution dans ce cas).

Nous fixons donc une ISA particulière arbitrairement dans nos expérimentations, MIPS32. Au contraire l'architecture mémoire ayant un impact fort nous avons bâti une solution permettant l'exploration de différentes architectures. Ainsi de manière à utiliser des technologies mémoires variées et à explorer l'architecture mémoire, nous avons choisi de mettre en œuvre une simulation du matériel.

Notre solution peut donc simuler de nombreux bancs mémoires. Ces bancs mémoires peuvent être raccordés directement au processeur, ou être accédés depuis le bus. Un banc contient le code de l'application, un autre les données (variables globales, constantes), une troisième la pile d'exécution.

À ces bancs s'ajoutent de 1 à n bancs dédiés aux objets du tas. Pour chacun de ces bancs il est possible de configurer indépendamment la latence de lecture et d'écriture. De plus nous modélisons le fonctionnement interne de chaque banc pour s'assurer de la pertinence des timings mémoires du simulateur. Ainsi, le simulateur permet d'instrumenter l'utilisation de la mémoire (nombre des lectures/écritures avec dates). Pour lier ces informations au niveau mémoire à l'information de l'application (profil d'allocation), les API de gestion de la mémoire dynamique sont également instrumentés.

De cette manière nous pouvons simuler l'exécution d'un programme et mesurer d'une manière fiable les effets de la résolution du problème de placement.

6.2.3 Performances mesurées pour l'influence du placement

Comme nous l'avons vu, résoudre le problème de placement correspond à choisir un banc mémoire de destination pour chaque objet. Cette décision a pour impact que les accès mémoire visant cet objet n'auront pas pour cible le même banc mémoire. Dans le cas d'une architecture mémoire hétérogène ces accès mémoire viseront donc des mémoires de technologies différentes. Notre simulateur permet de mesurer les différents accès générés vers les différents objets et peut

donc mesurer l'efficacité du placement sur toutes les grandeurs dépendant de ces accès (nombre, ordre, timing).

Nous avons proposé à la fin de la première section de ce chapitre quelques scénarios d'exemples où le placement mémoire influe sur différents paramètres physiques (vitesse d'exécution, endurance des mémoires, consommation énergétique). La mesure de ces différentes grandeurs peut être envisagées pour caractériser l'impact du placement mémoire des objets du tas sur le système. Toutefois elles ne présentent pas les mêmes implication méthodologiques.

La consommation énergétique des systèmes embarqués est une mesure physique classique et importante notamment pour évaluer leur durée de vie. Or les technologies que nous envisageons d'adopter pour les mémoires embarqués dans des systèmes embarqués contraints reposent sur des effets physiques différents comme vu au chapitre 4. La consommation énergétique d'un banc mémoire dépend aussi de l'intégration et de la taille du nœud technologique employé, paramètres que nous ne maîtrisons pas. De plus une simulation modélisant un haut niveau de détail matériel, par exemple en VHDL (VHSIC Hardware Description Langage – Very High Speed Integrated Circuit) serait trop lourde pour nous permettre d'étudier le problème dans un temps raisonnable.

Le temps d'exécution des systèmes embarqués contraint est une mesure de performance intéressante sur plusieurs aspects. En effet le système embarqué contraint travaille principalement lors de brefs pics d'activité et préserve son énergie le reste du temps. Réduire le nombre de cycles nécessaire à l'exécution de ces pics d'activité permet donc d'économiser de l'énergie.

L'étude de l'endurance des technologies mémoire ciblées peut aussi guider des choix de conceptions ou décider de choix logiciels visant à étendre la durée de vie du système.

Nous avons choisi dans cette étude de mesurer les performances du système en terme de temps d'exécution. Ce choix nous permet d'étudier l'impact de la résolution du problème de placement, tout en restant simple à mesurer. L'endurance est facile à mesurer dans cette modélisation, mais nous l'avons laissé à des travaux futurs.

6.2.4 Réduction du problème à deux tas

La littérature sur le placement de données en mémoire hétérogène considère généralement le temps d'exécution comme mesure de performances. Mais l'ensemble des études réalisées dans ce sens réduisent le problème de placement à deux mémoires cibles. Cette réduction est appliquée par toute la littérature du placement en mémoire Scratch Pad, notamment car les technologies mémoires NVRAM évoquées au chapitre 4 sont apparues après la construction de la majorité de l'état de l'art sur le sujet.

Néanmoins, disposer de plus de deux technologies mémoire dans l'architecture d'un système permettrait de mettre à profit les spécificités de chacune. Nous soutenons que cette approche est prometteuse et ouvre des possibilités intéressantes pour des systèmes embarqués contraints ou dans le domaine de l'IoT pour augmenter la durée de vie et l'efficience énergétique de ces systèmes. Grâce à une résolution intelligente du problème de placement des données, de tels systèmes pourraient tirer parti de technologies à faible endurance, ou compenser les coûts énergétiques en écriture de certaines technologies.

Pour proposer une résolution du problème de placement des objets du tas en mémoire hétérogène, nous prenons le parti de nous reposer sur la simplification du problème proposée par la littérature.

En effet, au delà de rendre cette première approche du problème plus abordable il est probable que les premières architectures mémoires utilisant les technologies abordées au chapitre 4 correspondent à cette simplification.

Nous traiterons donc dans ce travail d'une architecture mémoire hétérogène dont seulement deux bancs mémoire sont dédiées au tas de l'application.

Cette simplification et le choix de mesurer les performances en temps d'exécution nous permettent de définir l'architecture mémoire cible comme composée de mémoire rapide et lente. La *mémoire lente* est utilisée dans un banc dédiée au tas de l'architecture cible. En règle générale une technologie mémoire plus lente présentera une meilleure densité d'implantation, représentant la majorité de l'espace mémoire dédié au tas. La *mémoire rapide*, à l'inverse, est généralement d'implantation moins dense, mais répond plus vite. Suivant les technologies envisagées et la taille du nœud technologique ces deux type de mémoire peuvent consommer plus ou moins d'énergie

technologie	usage	latence en lecture (ns)	latence en écriture (ns)
STT-MRAM rapide	mémoire rapide	15	40
STT-MRAM dense	mémoire lente	25	200 - 400

FIGURE 6.1 – Description des technologies mémoire du scénario "MRAM"

technologie	usage	latence en lecture (ns)	latence en écriture (ns)
SRAM	mémoire rapide	2	2
ReRAM	mémoire lente	10	10 000

FIGURE 6.2 – Description des technologies mémoire du scénario "ReRAM"

pour leurs différentes opérations. On parlera du *tas rapide* (resp. *tas lent*) pour parler du tas composé uniquement de mémoire rapide (resp. lente).

6.2.5 Choix des technologies mémoire pour l'étude

Nous considérons dans cette thèse différentes technologies pouvant composer des architectures mémoires pour l'embarqué constraint. Néanmoins ce travail n'a pas pour ambition d'évaluer toutes les technologies mémoires disponibles pour cet usage. Nous en choisissons donc quelques unes pour étudier différents cas technologiques.

L'un des choix technologique se porte sur la technologie développée par notre partenaire industriel au début du projet, la startup eVaderis, qui en partenariat avec Global Foundries a proposé des technologies MRAM. Cette technologie présente différentes performances pour différents choix d'implantation. L'autre se base sur l'état de l'art provient de données de publications académiques [JASL+19].

Cet aspect du problème est le moins exploré dans ce projet, en effet, comme nous l'illustrerons au chapitre 8 les choix technologiques des mémoires implantées bornent les performances atteignables, mais n'ont que peu d'impact sur la manière de les atteindre, si ce n'est une influence plus ou moins grande de l'asymétrie entre lecture et écriture.

MRAM rapide contre MRAM dense

Comme évoqué précédemment cet exemple technologique est issu d'un partenariat industriel qui nous a permis d'évaluer des solutions crédibles du point de vue des technologies NVRAM réellement disponible sur le marché pour l'intégration en tant que mémoire interne du processeur.

Pour ce scénario, nous évaluons deux densités d'intégrations différentes pour une même technologie de MRAM. Ces différences d'implantation donnent une version exposant une meilleure densité alors que l'autre offre une plus grande vitesse.

Ces valeurs sont tirées des discussions avec notre partenaire industriel, de leur publication académique [LJBDG+16] ainsi que de la référence technique suivante : [Fou18]

SRAM contre ReRAM

Comme nous l'avons vu au chapitre 4, de nombreuses technologies peuvent prétendre à l'intégration pour les usages de NVRAM. En ce qui concerne l'embarqué constraint, notamment du à une fréquence en moyenne plus faible des coeurs, certaines technologies mémoires qui seraient inenvisageable en mémoire de travail autrement peuvent être utilisée de cette manière.

Pour illustrer ce cas nous considérons une technologie mémoire très lente en écriture mais pouvant être implanté très densément comme mémoire lente et de la SRAM en mémoire rapide. La technologie mémoire lente, proposée par Intel [JASL+19], pourrait ainsi autoriser l'exécution de programmes ayant besoin de beaucoup plus de mémoire si sa vitesse en écriture pouvait être masquée.

6.2.6 Hypothèse : mémoire non dédiée au tas, interférences des latences

Nous nous intéressons spécifiquement au comportement des objets du tas. Toutefois une majorité des accès générés par le processeur ne les concernent pas, mais concerne le code et des variables, locales ou globales.

Il en découle que la mesure des performances d'un placement pourrait être faussée par les autres accès mémoire. De plus sur un système mémoire hétérogène les données de l'application en dehors du tas doivent également faire l'objet de décisions de placement.

Nous avons vu au chapitre 5 que ces problèmes de placement ont déjà été formulés et que des solutions efficaces ont été proposées par la littérature. De plus les différentes méthodologies proposées correspondent à nos conditions d'expérimentation. En effet la réduction du problème à deux bancs mémoire et la prise en compte du temps d'exécution comme mesure de l'efficacité du placement sont des choix classiques de la littérature. Par exemple les auteurs de l'outil de profilage ProfDP [WCLL18] définissent l'hétérogénéité mémoire de cette manière : deux mémoires accessibles par le CPU ayant des latences différentes.

Nous n'avons pas ré-implémenté ces techniques pour le code, les variables globales et la pile d'exécution. Toutefois les travaux présentés au chapitre 5 démontrent que de telles approches arrivent à masquer en grande majorité les latences mémoire du programme, proposant des performances proche d'un programme dont toutes les données sont placées dans la mémoire rapide du système.

De plus nous souhaitons éviter toute interaction entre les latences mémoire du tas – issue de la résolution du problème de placement – et des latences mémoire dues au placement du reste des données applicatives en mémoire.

Nous isolons donc le problème du tas en exécutant nos applications avec le code, la pile et les variables globales positionnées en mémoire idéale. Nous pouvons ainsi mesurer fidèlement l'effet du placement mémoire sur le temps d'exécution de l'application sans perturbation.

Nous soutenons que cette hypothèse n'introduit pas de biais significatif dans les résultats. En effet la majorité des accès du processeur concerne la lecture du code. Si le processeur ne pouvait lire une instruction en un cycle il serait obligé d'attendre un ou plusieurs cycles pour chaque lecture d'instruction. Dans ces conditions soit la fréquence du processeur serait abaissée, soit un mécanisme matériel permettrait d'accéder à une instruction par cycle. Nous pouvons donc considérer qu'une partie de la mémoire destinée aux reste des données de l'application peut être approximée comme de la mémoire idéale.

De plus, l'empreinte du tas des applications considérées est en règle générale plus grande que la taille du code et des données de celles ci. Ainsi considérer que le code, les variables globales et les variables locales sont en mémoire idéale nous semble être une hypothèse raisonnable.

6.2.7 Exemple illustratif

Pour illustrer le problème de placement des objets du tas, nous proposons l'exemple simpliste suivant. Bien que ne représentant pas le comportement réel d'une application, nous allons utiliser cet exemple pour présenter d'une manière claire les différents concepts utilisés.

Dans un premier temps nous décrivons le code du programme d'exemple, puis les éléments importants dans son exécution ainsi que le profil en résultant. Nous illustrons dans un deuxième temps l'approche méthodologique décrite dans cette section.

Code de l'application

L'application à laquelle nous allons nous intéresser est une suite d'allocation, de désallocation, d'affectations et de copies concernant quelques objets.

La figure 6.3 (a) contient le pseudo code de notre application d'exemple. Cette application utilise des fonctions de la bibliothèque standard dont la déclaration est rappelée en (c). Lors de l'exécution sur simulateur de l'application, nous allons mesurer un certain nombre d'événements formant la liste des actions relatives aux objets de l'application présentée en (b) de la même figure. Nous nous servons alors de ces informations pour construire le profil de l'exécution, présenté à la figure 6.4. Nous considérons les actions ordonnées du gestionnaire mémoire dynamique, allocation / désallocation.

```

int main()
{
    int * A, B, C, D;
    A = malloc(42);
    memset(A, 0xFF, 42);
    B = malloc(96);
    memcpy(A, B, 42);
    free(A);
    C = malloc(50);
    D = malloc(30);
    free(B);
    memset(D, 0, 30);
    memset(C, 0, 50);
    memcpy(D, C, 15);
    free(D);
    free(C);
    return 0;
}

```

Liste des actions :

- alloc A 42
- write A 42
- alloc B 96
- read A 42
- write B 42
- free A
- alloc C 50
- alloc D 30
- free B
- write D 30
- write C 50
- read D 15
- write C 15
- free D
- free C

(b)

(a)

```

// allocates size bytes and returns a pointer to the allocated
// memory
void *malloc(size_t size);

// frees the memory space pointed to by ptr
void free(void *ptr);

// fill n bytes of memory pointed by dest with a constant byte c
void *memset(void *dest, int c, size_t n);

// copy memory area of n bytes from src to dest
void *memcpy(void *dest, const void *src, size_t n);

```

(c)

FIGURE 6.3 – Exemple illustratif du problème de placement des objets du tas : (a) pseudo-code applicatif, (b) liste des actions du programme pour la formulation du problème de placement (c) headers des fonctions standard utilisées

Objet	Profil d'allocation			Profil d'accès	
	Taille(octets)	Ordre d'allocation	Ordre libération	nb lectures	nb écritures
A	42	1	3	42	42
B	96	2	6	0	42
C	50	4	8	65	0
D	30	5	7	15	30

FIGURE 6.4 – Profil résultant de l'exécution de l'application d'exemple de la figure 6.3

Exécution de référence contre exécution rapide

Considérons l'exécution de ce programme dans les conditions suivantes. On considère pour cet exemple l'exécution sur les technologies mémoires suivantes :

- mémoire rapide (m_f) : latence symétrique d'un cycle, $lat(m_f) = 1$
- mémoire lente (m_s) : latence symétrique de 10 cycles, $lat(m_s) = 10$

L'exécution rapide de ce programme dépendra uniquement de la vitesse du processeur et des calculs requis pour l'exécution des instructions du programme (pas d'influence des latences mémoire). En effet, tous les accès mémoires répondent en un cycle. Dans ce cas la mémoire rapide est aussi idéale.

Notons T_{rapide} le temps d'exécution de l'exécution rapide. On nomme également Obj l'ensemble des objets décrits dans la figure 6.4

On peut alors calculer l'impact mémoire des accès au objets du tas si ils sont placés en mémoire lente plutôt que rapide.

Dans ces conditions :

$$T_{ref} = T_{rapide} + \sum_{\forall b \in Obj} (count_R(b) + count_W(b)) \times (lat(m_s) - lat(m_f))$$

Avec $count_R(b)$ (resp. $count_W(b)$) le nombre d'accès à b en lecture (resp. écriture). Dans cet exemple le placement mémoire peut donc faire varier le temps d'exécution d'au maximum $\Delta T_{max} = 2124$ cycles.

Solutions de placement

Une solution de placement consiste à donner un tas de destination pour chaque objet, en respectant la taille de la banc de destination. Pour cet exemple nous ignorons les détails d'implémentation du gestionnaire mémoire et nous pouvons ainsi calculer l'impact de deux placements différents pour une architecture mémoire. Soit une architecture mémoire contenant 100 octets de mémoire rapide et autant de mémoire lente que nécessaire. on peut imaginer les placements suivants :

- cas 1 : mémoire rapide : $\{A, C\}$ / mémoire lente : $\{B, D\}$
- cas 2 : mémoire rapide : $\{D, C\}$ / mémoire lente : $\{A, B\}$
- cas 3 : mémoire rapide : $\{B\}$ / mémoire lente : $\{A, B, D\}$

L'application de ces choix impliquerait un impact sur le temps d'exécution différent dans chaque cas, en fonction des objets présents dans le tas lent. On peut donc calculer pour chaque cas le nombre d'écritures dans le tas lent ajouté à l'exécution

- cas 1 : $(count_R(B) + count_W(B) + count_R(D) + count_W(D))$
- cas 2 : $(count_R(A) + count_W(A) + count_R(B) + count_W(B))$
- cas 3 : $(count_R(A) + count_W(A) + count_R(B) + count_W(B) + count_R(D) + count_W(D))$

Et de là calculer le temps en cycle ajouté au temps d'exécution rapide dans ces trois cas ΔT , correspondant au nombre d'accès décrit plus haut, multiplié par la différence de latence entre les deux mémoires :

- cas 1 : $\Delta T = 87 \times (10 - 1) = 783$
- cas 2 : $\Delta T = 126 \times (10 - 1) = 1134$
- cas 3 : $\Delta T = 171 \times (10 - 1) = 1539$

Ces résultats nous permettraient de calculer l'accélération par rapport à l'exécution de référence, et donc d'évaluer l'impact du placement mémoire. Ici, nous ne pouvons pas calculer l'accélération pour l'exécution de référence, faute d'une mesure de celle ci mais on peut calculer l'importance de ΔT par rapport à l'influence maximale sur les latences mémoires ΔT_{max} d'après la formule suivante :

$$I = \frac{(\Delta T_{max} - \Delta T)}{\Delta T_{max}}$$

- cas 1 : $I = 63\%$
- cas 2 : $I = 47\%$
- cas 3 : $I = 28\%$

On voit donc que, ici, le choix des objets à placer en mémoire rapide à un impact fort sur les performances de l'application en fonction du profil d'accès au tas. Il est toutefois à noter que cette exemple n'est pas totalement exact. En effet, il ne tient pas compte des différences d'exécution à l'intérieur de l'allocateur mémoire, et il ne mesure pas non plus le temps passé pour la prise des décisions de placement.

6.3 Méthodologie d'évaluation de la gestion mémoire dynamique

Nous avons restreint notre périmètre d'étude de la manière suivante. Pour le domaine de l'embarqué constraint nous nous proposons d'étudier le placement des objets du tas des applications entre deux bancs mémoire aux latences différentes, présentant possiblement une asymétrie lecture/écriture et en mesurant les performances en terme de temps d'exécution de l'application. Nous allons maintenant aborder la méthodologie à l'aide de laquelle nous allons pouvoir évaluer les différentes solutions de placement.

6.3.1 Exécutions sur architecture mémoire hétérogène

Pour comparer l'influence de placements mémoire, il faut que le reste de l'exécution soit identique. Dans les conditions qui nous concernent, des exécutions seront identiques si pour le même jeu de données, les mêmes instructions sont exécutées dans le même ordre et génèrent les mêmes sorties. Nous pouvons avoir deux exécutions identiques car le matériel cible est un microcontrôleur à un seul cœur, n'exécutant qu'un thread, n'ayant ni cache matériel et exécutant les instructions dans l'ordre.

Que se passerait t'il si nous placions les objets différemment, c'est à dire si nous changions la répartition des objets entre les différents bancs mémoire ? Le gestionnaire mémoire doit forcément exécuter des instructions différentes, et nous développerons cet aspect plus loin dans ce chapitre. Mais si nous nous intéressons à l'exécution du programme en dehors du gestionnaire mémoire, sur la même architecture mémoire avec le même jeu de données, quel impact peut avoir un placement mémoire différent ?

Nous listons deux cas où l'exécution en dehors du gestionnaire mémoire changerait dans ces conditions :

Premièrement, si le chemin d'exécution dépend de timings d'exécution de certaines parties du programme, un placement mémoire différent influant sur les latences mémoires, l'exécution pourrait générer des sorties différentes. Cependant, le cas le plus impactant pour ces considération est celui des applications temps réel dur qui est laissé en dehors de notre étude. Un autre exemple est celui d'une application multimédia qui peut être amenée à adapter le niveau d'encodage en fonction du temps dont elle dispose pour préparer l'affichage ou l'envoi d'une image. Néanmoins, pour éviter cet effet dans le cas de cet exemple, nous fixons ces paramètres sur un jeu de données pour son étude. Ce type d'approche permettant de produire le même profil d'allocation et d'accès aux objets du tas pour deux placements différents pour notre étude.

Deuxièmement, certains algorithmes ont des chemins d'exécution dépendant des adresses retournées par l'allocateur mémoire dynamique, par exemple l'implémentation d'une fonction retournant un code de hachage dans java pour un objet se contente de convertir son adresse. Mais dans ce cas, étant donné que la spécification n'impose pas que le code retourné soit le même entre différentes exécutions, il ne serait pas problématique de comparer deux exécutions ayant des placements différents. Nous n'avons toutefois pas trouvé d'exemple applicatif de ce genre dans la construction de notre suite de benchmarks. Nous nous permettons donc de ne pas prendre ces exemples en compte dans la définition suivante :

DÉFINITION - exécutions équivalentes : exécutions d'une application avec le même jeu de données, générant les mêmes sorties, mais appliquant des placements mémoires différents. Des exécutions équivalentes exécutent donc des séquences d'instructions identiques en dehors du placement et de l'allocation mémoire dynamique, dépendant de la stratégie de placement.

La méthodologie que nous allons donc utiliser pour évaluer les solutions de placement des objets du tas est donc de réaliser des exécutions équivalentes d'applications avec les mêmes jeux de données d'entrée et des placements mémoire différents. Mais plus encore que des solutions de

placement, nous allons devoir évaluer des stratégies de placement guidant la prise des décisions de placement.

DÉFINITION - *stratégie de placement* : règles sous-tendant la prise de la décision de placement.

Nous allons donc évaluer ces stratégies sur plusieurs applications avec plusieurs jeux de données. Mais comme nous l'avons vu au chapitre 3, le domaine de l'embarqué constraint est un domaine où le développement des applications est souvent considéré en parallèle de l'adaptation d'architectures matérielles (conception conjointe matériel-logiciel). Ainsi notre méthodologie d'évaluation aussi évaluer le comportement des applications sur différentes architectures mémoires. Ainsi il est possible de mettre en lumière l'impact sur les performances que peut avoir la conjonction d'une bonne stratégie de placement mémoire et d'une architecture mémoire paramétrée pour les besoins de l'application.

6.3.2 Interraction du placement avec l'allocation mémoire dynamique

Nous avons exclu le gestionnaire mémoire dynamique des définitions précédentes. En effet, même si l'exécution de l'application est équivalente, un placement différent implique une exécution différente de l'allocateur mémoire dynamique. Or la résolution du problème de fragmentation du tas peut avoir un impact fort sur les performances de l'application. Ainsi nous ne pouvons pas nous contenter de calculer l'impact des latences en fonction d'un placement mémoire à partir de la trace d'exécution d'un programme. Autrement dit, la résolution du problème de la fragmentation du tas évoqué au chapitre 2 n'est pas indépendante de la résolution du problème de placement.

Nous en concluons que pour évaluer l'influence d'une solution de placement, il est nécessaire de rejouer l'exécution de cette application pour cette solution de placement. De plus, le temps passé dans le gestionnaire mémoire dynamique doit être pris en compte, de manière à caractériser l'interaction entre le problème de fragmentation du tas et la résolution du problème de placement.

6.3.3 Évaluation des solutions de placement

Nous pouvons à ce point comparer deux exécutions équivalentes et en déduire une comparaison des solutions de placement qu'elles appliquent. Pour ce faire nous devons appliquer différents placements pour une application, sur une même architecture mémoire et pour le même jeu de données.

Il manque cependant un point de référence pour évaluer plus généralement les solutions de placement. En effet les solutions évaluées les unes contre les autres ne permettent pas d'évaluer le temps passé à choisir le placement mémoire. Cette comparaison ne permet de plus qu'une évaluation relative de l'impact de la résolution du problème de placement sur les performances du gestionnaire mémoire dynamique. Nous allons donc utiliser une exécution de référence à laquelle nous comparerons les performances des exécutions appliquant différentes stratégies de placement.

DÉFINITION - *exécution de référence* : exécution d'une application, pour un jeu de données, sur un seul tas lent géré par `DLmalloc`.

Du point de vue de l'exploration architecturale cette exécution représente les performances de l'application que l'on va essayer d'améliorer en ajoutant de la mémoire rapide à l'architecture mémoire. D'un point de vue plus général, si l'on décide de dédier une partie de la mémoire rapide au tas, cette exécution représente la base par rapport à laquelle on veut gagner en performance. Les résultats que nous présentons sont donc donnés sous la forme d'accélération du temps d'exécution de l'application par rapport à l'exécution de référence. On définit conjointement l'exécution de l'application représentant des performances idéales à priori non atteignables par la résolution du problème de placement. Correspondant à l'exécution de référence sur une architecture mémoire uniquement composée de mémoire rapide

DÉFINITION - *exécution rapide* : exécution d'une application, pour un jeu de données, sur un seul tas rapide géré par `DLmalloc`.

on définit également l'architecture de référence et l'architecture rapide :

- **DÉFINITION - *architecture de référence*** : architecture mémoire pour le tas d'une application uniquement composée de mémoire lente
- **DÉFINITION - *architecture rapide*** : architecture mémoire pour le tas d'une application uniquement composée de mémoire rapide

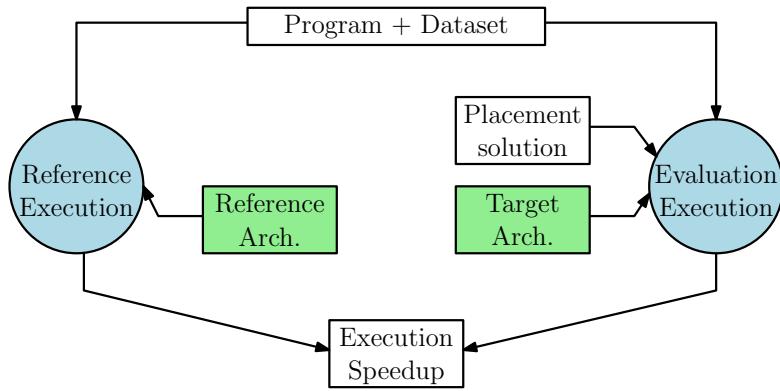


FIGURE 6.5 – Déroulement de l'évaluation d'une solution de placement des objets du tas

- DÉFINITION - *architecture idéale* : architecture mémoire pour le tas d'une application uniquement composée de mémoire idéale

L'évaluation d'une solution de placement consiste donc à comparer le temps d'exécution de l'application avec un jeu de données en entrée appliquant la solution de placement à l'exécution de la même application avec le même jeu de données sur une architecture composée uniquement de mémoire lente. Il est à noter que l'exécution de référence n'est pas pénalisée par le surcoût de résoudre le problème de placement et qu'ainsi l'accélération obtenue prend en compte les coûts de la résolution du problème de placement. Cette approche est présentée en figure 6.5.

6.4 Couche logicielle d'allocation multi-tas

6.4.1 Allouer des objets : Hiérarchie vs Hétérogénéité mémoire

Nous avons jusqu'ici dans ce chapitre expliqué comment nous abordons le problèmes de placement des objets du tas. Nous abordons maintenant la réponse à la question suivante : "Comment allouer des objets en mémoire hétérogène ?"

Dans une hiérarchie mémoire classique le cache matériel a pour but d'abstraire le fait que les données sont situées en mémoire lente en chargeant automatiquement les données accédées en mémoire rapide. Dans ces conditions l'allocateur mémoire essaye de grouper les données accédées en même temps pour limiter les fautes de caches et améliorer les performances de l'application.

Mais ce cas ne se présente pas dans les conditions que nous considérons – absence de cache matériel. Ainsi nous pouvons séparer des objets qu'une méthode d'allocation classique aurait groupé suivant d'autres critères.

Nous en déduisons qu'il est possible d'allouer des objets dans deux tas différents, chaque tas ayant ses propres structures de données décrivant son état et gérant un banc mémoire différent. Suit donc la description des mécanismes logiciels permettant le placement mémoire et l'allocation multi tas.

6.4.2 Architecture logicielle pour l'allocation multi-tas

La figure 6.6 décrit notre architecture logicielle. Nous avons vu ci-dessus que séparer les objets entre différents tas ne posait pas de problème. Nous allons donc les distinguer en fonction de leur profil d'accès, où de ce que nous pourrons anticiper sur leur profil d'accès au moment de leur allocation. Cette architecture logicielle est basée sur le choix de préserver l'interface standard de `malloc`. Ainsi l'utilisation de cette architecture ne devrait pas impliquer de charge de développement pour le développeur d'application embarqué, et être capable d'exploiter du code hérité. De plus cette architecture permet de réutiliser une solution standard au problème de fragmentation du tas, c'est à dire un algorithme d'allocation mémoire efficace dans le cas général. Un composant logiciel est chargé de prendre les décisions de placement :

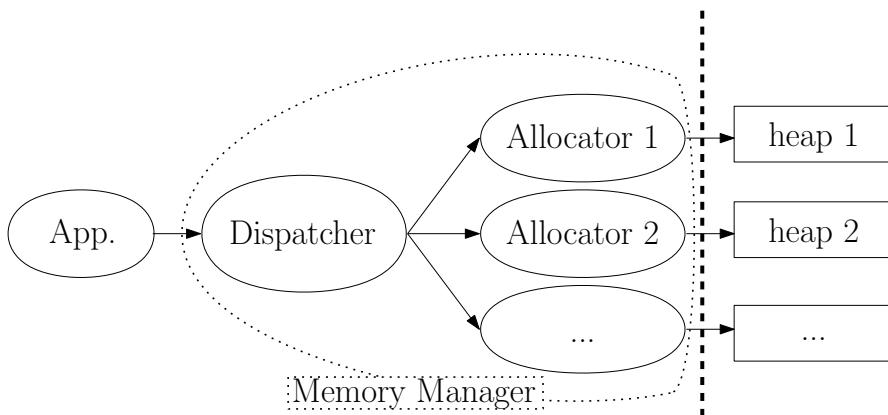


FIGURE 6.6 – Architecture logicielle pour la résolution du problème de placement

DÉFINITION - *dispatcheur* : couche logicielle exposant l'API standard `malloc / free` et prenant les décisions de placement. Le dispatcheur transmet ensuite la requête d'allocation au gestionnaire mémoire dynamique gérant le tas choisi.

Les décisions prises par le dispatcheur correspondent à la résolution du problème de placement et on peut donc préciser que c'est le dispatcheur qui implémente la stratégie de placement. Il est à noter que ces stratégies peuvent être pré-calculées (voir chapitre 8) ou prise durant l'exécution :

- Stratégie *en ligne* : partie ou totalité des informations permettant la prise de décision sont obtenue durant l'exécution, la décision est prise au moment de la requête d'allocation mémoire.
- Stratégie *hors ligne* : la décision est prise relativement à une exécution préalable, le dispatcheur ne fait qu'appliquer une décision prise au préalable.

Cette architecture logicielle implique que le dispatcheur peut prendre une décision inapplicable. En effet si il essaie de placer un objet dans un tas donc le banc mémoire est saturé il se peut que l'allocation mémoire échoue. Ce cas peut se produire soit car le tas de destination de l'objet soit plein, soit car il est trop fragmenté et ne dispose donc pas d'assez d'espace contigu pour satisfaire la requête. Dans ce cas le dispatcheur essayera d'allouer dans l'autre tas :

DÉFINITION - *fallback* : action du dispatcheur si l'application de la stratégie n'est pas réalisable car le tas ciblé est plein. Le fallback consiste à allouer dans un autre tas pouvant satisfaire la requête.

Nous avons dans cette partie décris de quelle manière nous allons allouer les objets dans une architecture mémoire non hétérogène. Le domaine de notre étude et l'existence d'algorithmes d'allocation mémoire efficaces pré-existants ont guidé notre choix de gérer différents tas, un par banc mémoire cible.

6.5 Architectures mémoire hétérogènes pour le tas

Nous avons vu comment nous allons évaluer les solutions de placement, et quelles performances nous allons mesurer. L'architecture logicielle que nous avons décrite ci-dessus permet de mettre en œuvre cette résolution du problème de placement en respectant les conditions du domaine, c'est à dire sans nécessiter de modification de code source applicatif.

Mais comme nous l'avons évoqué plus haut le domaine de l'embarqué constraint favorise la conception conjointe matériel-logiciel des systèmes. Ainsi le design d'un système et l'optimisation de ses performances passe souvent par une phase d'exploration architecturale. Par là nous entendons que les IP mémoires utilisées dans la conception de ces plateformes sont paramétrées, notamment en fonction de l'espace mémoire requis par l'application.

Pour une application donnée, nous allons donc étudier le problème de placement dans un tas dont la taille maximale correspond aux besoins de l'application en terme d'allocation mémoire dynamique. Nous avons ainsi fait le choix de paramétrier la taille totale du tas des architectures mémoires considérées par la taille totale de l'empreinte mémoire des applications cibles.

Nom	% Mémoire rapide	% Mémoire lente
architecture de référence	0	100
architecture 5%	5	95
architecture 10%	10	90
architecture 25%	25	75
architecture 50%	50	50
architecture 75%	75	25
architecture rapide	100	0

FIGURE 6.7 – Architectures mémoires hétérogènes – taille totale paramétrée par l’empreinte mémoire de l’application

Un autre aspect à prendre en compte dans le choix des architectures mémoires que nous allons étudier est de savoir si elles sont représentatives des architectures mémoires du domaine. Nous ciblons notre étude sur le temps d’exécution et comme vu au chapitre 4 les technologies mémoires ou leurs implantations présentant les latences les plus faibles sont les plus coûteuses en surface de silicium. Nous définissons donc les architectures intéressantes comme ayant moins de mémoire rapide que de mémoire lente.

Nous allons donc réaliser nos expériences sur les architectures mémoires suivantes, sachant que la taille totale de l’architecture est déterminée par l’empreinte mémoire des applications cibles. Par soucis de lisibilité nous attribuons un nom à chacune de ces architectures, présentées dans la figure 6.7

6.6 Conclusion

Dans ce chapitre nous avons construit le cas d’étude que nous allons utiliser dans la suite de ce manuscrit pour montrer l’impact de l’allocation des objets du tas en mémoire hétérogène. Nous avons défini la notion de stratégie de placement, et avons proposé une architecture logicielle et un ensemble d’architecture mémoires pour réaliser ce placement. Par la suite nous allons donc proposer et évaluer des stratégies de placement permettant de résoudre le problème énoncé.

Mais dans un premier temps il nous reste à justifier des benchmarks que nous allons utiliser pour réaliser cette étude. Nous verrons donc dans la suite quelles applications nous allons étudier, ainsi que leurs caractéristiques au regard du problème de placement.

Chapitre 7

Étude expérimentale du comportement des applications

Nous avons exposé le problème de placement et proposé une méthodologie permettant son étude au chapitre précédent. Nous allons maintenant devoir choisir quelles applications considérer pour cette étude. Comme nous le verrons dans ce chapitre pour justifier une démarche de résolution complexe nous estimons qu'il est nécessaire que l'application fasse un usage significatif de l'allocation mémoire dynamique. Une fois cette nécessité justifiée, nous pourrons présenter des applications pertinentes pour notre cas d'étude, spécifique au domaine de l'embarqué. Mais dans un premier temps nous allons présenter des choix de simulation et d'implémentation relatifs à l'étude de ces applications. Nous pourrons par la suite construire une métrique permettant d'évaluer l'importance des objets vis à vis du problème de placement. Puis nous terminerons par la caractérisation des applications basée sur cette métrique de manière à pouvoir en tirer des stratégies de placement.

7.1 Infrastructure logicielle nécessaire à l'étude

7.1.1 Plateforme

Choix de la solution de simulation

Il existe de nombreuses solutions de simulation, notamment pour l'étude de l'architecture du sous système mémoire.

Notre approche d'un problème logiciel complexe influe toutefois sur celles que nous pouvons envisager. En effet une approche classique est de simuler en détail le fonctionnement de la mémoire.

On peut ici citer en exemple NVSim [DXXJ12] qui permet une étude approfondie de l'organisation de la mémoire, permettant le prototypage et l'étude de l'implantation de nœuds mémoire. Un autre outil répandu pour cet usage est CACTI [TMAJ08].

Une autre approche largement répandue correspond à l'étude de l'architecture mémoire dans son ensemble, en utilisant des composants mémoire dont le comportement est caractérisé. Notre approche se classe dans cette seconde catégorie. Le simulateur GEM5 [BBBR+11] est la solution la plus répandue pour cette approche. Le problème soulevé par cette solution de simulation est qu'elle fait l'hypothèse de la présence d'une hiérarchie mémoire. Nous avons écarté cette solution de simulation bien qu'elle soit efficace pour la complexité d'adaptation à notre cas d'étude. En effet, l'architecture mémoire par défaut de ce simulateur présente un cache mémoire matériel et donc un modèle de mémoire Scratch Pad n'est pas facilement interfaçable sans passer par le bus.

On peut également noter que de nombreuses solutions de simulations existent pour étudier le comportement logiciel au dessus d'un système mémoire contenant des technologies différentes. Toutefois, ces solutions de simulation visent le plus souvent des applications HPC, reposant sur une simulation au niveau système d'exploitation ([VEPAG12; SWVC+15]). Ces solutions d'émulation présentent de bonnes performances et permettent d'étudier efficacement les comportements logiciels, malheureusement elles reposent sur le fait que la cible logicielle soit dotée d'un système d'exploitation, ce qui n'est pas le cas de notre étude.

Nous avons donc fait le choix de construire une plateforme de simulation sur un niveau d'abstraction correspondant à l'étude des mécanismes logiciels de placement. Dans la suite nous présentons la technologie de simulation utilisée, SystemC/TLM, ainsi que les principaux éléments de la simulation.

Technologie de simulation

Cette plateforme est basée sur le cœur de simulation SystemC [Sys]. Plus qu'un cœur de simulation, SystemC est un pseudo-langage de description de systèmes hardware. Il permet la précision d'un langage de description matériel comme VHDL, précis au cycle près et au bit près. Mais il ne s'agit pas d'un langage de description hardware au sens propre. SystemC est un ensemble de classes C++, et permet de décrire des composants ou des systèmes de manière moins contraignante qu'une description matérielle. En effet il est possible de décrire un composant en ne modélisant que ses fonctionnalités évitant ainsi d'implémenter tout ou partie de son fonctionnement matériel. Cette solution de simulation est donc particulièrement adaptée à la conception de systèmes sur puce, facilitant des phases de prototypage et accélérant la conception conjointe matériel-logiciel. En effet par rapport aux solutions de simulations plus lourdes de description matérielle il permet de rapidement permettre le début du développement logiciel. Pour la modélisation et la communication entre les composants de la simulation, nous utilisons SystemC/TLM qui est un niveau d'abstraction du système basé sur des transactions entre les composants. Ce niveau de modélisation nous permet de réaliser des simulations complexes (programmes allouant de l'ordre de 100 000 objets) pour une durée de simulation raisonnable (quelques heures). Ce choix nous permet aussi de ne pas modéliser le fonctionnement interne de tout les composants mais de le faire au cycle près pour les mémoires étudiées.

Chaque lecture ou écriture mémoire est donc une transaction, et chaque composant mémoire modélise la latence induite par sa technologie. Comme nous l'avons dit au chapitre précédent nous évaluons les performances sur le temps d'exécution du programme. De plus, le simulateur de jeu d'instructions est précis au cycle près. Nous n'avons pas modélisé la contention du bus qui n'ajoute donc pas de délai. Toutefois, l'architecture considérée inclue deux interfaces directes entre le cœur de calcul et la SPM, une pour les données et une pour le code. Ainsi, la majorité des accès mémoire que nous observons est réalisée hors du bus.

L'ensemble de ces choix, que nous allons détailler ci dessous, nous permet de considérer que les résultats de simulations sont suffisamment précis pour modéliser et étudier le système mémoire et les latences qu'il induit dans le cadre de la gestion mémoire dynamique.

Architecture matérielle

La figure 7.1 présente l'architecture matérielle modélisée, simulée et étudiée dans le cadre de cette thèse.

Nous considérons que l'ensemble de la plateforme est un système sur puce et que les bancs mémoire intégrés à la mémoire Scratch Pad sont des modules mémoires paramétrables, potentiellement de technologies différentes.

Cette architecture matérielle est inspirée de l'architecture proposée par notre partenaire industriel durant le début de cette thèse, la startup eVaderis. Notre étude portant sur le placement mémoire et la communication étant faite directement entre le cœur et la mémoire Scratch Pad, ces éléments sont modélisés précisément comme décrit ci-dessous. Néanmoins le bus et les autres périphériques modélisés n'étant pas critiques du point de vue de notre étude, ils sont modélisés de manière fonctionnelle uniquement.

Les autres périphériques considérés sont un port série permettant d'afficher les sorties du programme embarqué, ainsi qu'un composant d'aide à la simulation. Ce composant permet par exemple, en y réalisant un accès mémoire d'enregistrer le début d'une allocation mémoire, ou la libération d'un objet. Il expose sur le bus différents registres, chacun appelant une callback dans le simulateur qui peut ainsi récupérer les informations utiles au profilage. La mémoire ainsi exposée n'insère pas de délai et l'instrumentation se limite donc à l'ajout d'un accès mémoire dans le programme. Ce composant est utilisé pour le profilage, ainsi que pour les entrées sorties du programme.

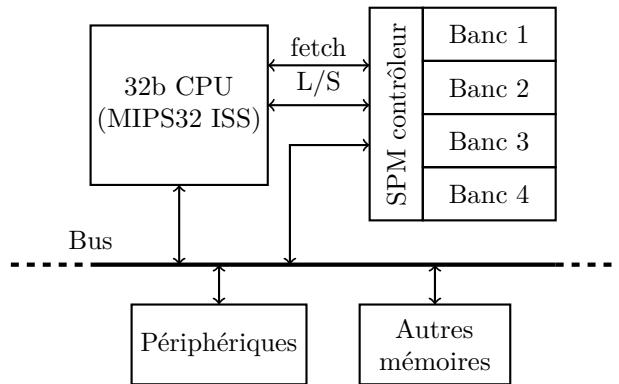


FIGURE 7.1 – Plateforme matérielle simulée

Architecture de jeu d'instructions

Le cœur de calcul simulé dans notre plateforme correspond au jeu d'instructions MIPS32. L'implémentation que nous utilisons, précise au cycle près et simulant un pipeline à trois étages est celle du projet SOCLIB [Con+]. Ce cœur de calcul présente des performances équivalentes à un processeur 32 bit basse consommation, et est donc relativement limité en puissance de calcul.

Nous avons intégré ce cœur de calcul à notre simulation au travers d'un wrapper SystemC/TLM, ce qui nous a permis de l'implémenter avec les spécificités suivantes :

- Pas de cache matériel : correspondant à une architecture mémoire Scratch Pad, notre plateforme ne possède pas de cache matériels
- Requêtes mémoires non bloquantes : la couche de simulation englobant le simulateur de jeu d'instructions ne se bloque pas en attendant la réponse d'une requête mémoire. Ainsi, sur une lecture le processeur attend la réponse jusqu'à ce que le système mémoire lui fournisse la réponse. Si une instruction a plusieurs cycle d'exécution, elle continue donc de s'exécuter durant cette attente.
- Découplage des écritures mémoires : le simulateur de jeu d'instruction n'attend qu'une écriture soit finie que si il doit réécrire dans le même banc mémoire ou relire la valeur qu'il vient d'y écrire. Ainsi, si plusieurs requêtes d'écritures visant des bancs différents sont émises, le processeur ne verra aucun délai et considérera les écritures comme réalisées. Mais si pendant que ces écritures sont réalisées en mémoire il a besoin de lire ou d'écrire dans un banc mémoire occupé il devra attendre la fin de la transaction précédente.

Nous estimons que cette implémentation permet de simuler fidèlement les éléments importants de notre étude, à savoir, le temps d'exécution influencé par les latences mémoires des données placées dans différents bancs mémoire aux latences différentes.

Architecture mémoire hétérogène

Nous simulons pour cette plateforme deux types de composants mémoires différents. La mémoire Scratch Pad d'une part regroupe plusieurs bancs mémoire et est accessible au travers de plusieurs interfaces. D'autre part d'autres bancs mémoire modélisent des mémoires accessibles depuis le bus uniquement.

Les bancs mémoire auxiliaires situés sur le bus représentent des mémoire de stockage ou des buffers de périphériques. Elles sont utilisées pour stocker des données en entrée ou en sortie des applications et non comme mémoire de travail, ainsi que le code de démarrage du processeur.

La mémoire Scratch Pad est utilisée comme mémoire de travail. Le nombre de bancs mémoire qu'elle contient est un paramètre d'exploration de la simulation. Comme nous l'avons vu au chapitre précédent, elle contient un banc mémoire dédié au code un autre aux données globales et à la pile d'exécution. Nous simulons un ou deux bancs mémoire supplémentaires dédiés au tas.

Chaque banc mémoire simulé modélise la latence en lecture et en écriture d'une technologie cible. Comme justifié au 6.2, les bancs mémoire dédiés au code et aux données (hors tas) simulent une mémoire idéale répondant au processeur sans introduire de latence mémoire. Chacun de ces bancs est accessible au travers de trois interfaces :

- "fetch" : interface de code, allant chercher en mémoire les instructions du programme à exécuter. Cette interface relie directement la mémoire Scratch Pad au CPU.
- "L/S" : interface de données, générant des lectures ou des écritures résultant de l'exécution des instructions du programme. Concerne tout autant les accès à la pile que les accès aux variables globales ou au tas. Cette interface relie directement la mémoire Scratch Pad au CPU.
- "bus" : interface permettant l'accès des différents bancs mémoire depuis le bus. Elle n'est utilisée qu'à la marge dans cette étude mais permettrait par exemple d'observer l'impact d'un transfert DMA depuis un périphérique vers la mémoire Scratch Pad.

L'accès des différentes interfaces aux différents bancs se fait au travers d'une matrice d'interconnexion. La politique d'arbitrage d'accès entre les différentes interfaces est une politique de priorité fixe en fonction des différentes interfaces. Ainsi, l'interface du code est plus prioritaire que celle des données, la moins prioritaire étant l'interface provenant du bus.

De plus, la mémoire Scratch Pad doit être capable de répondre simultanément sur ces trois interfaces (si les accès ciblent des bancs différents)

Ce design, proposé par notre partenaire industriel, a fait l'objet d'une validation fonctionnelle et de tests unitaires fournis dans les sources du projet.

7.1.2 Allocation multi-tas

Nous avons choisi dans cette étude (voir chapitre 6) d'utiliser plusieurs tas entre lesquels un dispatcheur réparti les requêtes. Cette approche sépare la résolution du problème de placement et du problème d'allocation. Nous pouvons donc réutiliser un allocateur mémoire de la littérature pour résoudre ce second problème.

Nous allons ici justifier le choix de cet allocateur mémoire, et présenter l'influence de la résolution du problème de placement sur l'allocation mémoire en elle-même. Nous expliquerons par la suite de quelle manière nous avons adapté cet algorithme à ces nouvelles conditions.

L'algorithme de gestion de la mémoire dynamique que nous avons choisi d'utiliser pour gérer chaque tas en dessous du dispatcheur est présenté au chapitre 2 : il s'agit du DLMalloc [LG96]. Satisfaisant dans le cas général, il est basé sur un ensemble de listes contenant des blocs vides de différentes tailles (segregated free list), avec différentes optimisations.

Il est notamment utilisé dans la libc GNU utilisée par le système d'exploitation linux, mais aussi intégré à newlib [New], une implémentation de la bibliothèque standard C destiné à l'embarqué que nous avons porté vers notre plateforme de simulation.

L'adaptation pour l'utilisation de cet algorithme en multi-tas correspond à la duplication des structures de données de description du tas pour permettre la sélection du tas de destination. À part la factorisation des structures de données, et les mécanismes attenant, par exemple le changement du tas actif ou la recherche du tas d'un objet à libérer, aucune modification n'est apportée à l'algorithme qui alloue donc toujours les objets de la même manière.

Multi-tas : changement des conditions d'utilisation de l'algorithme d'allocation

Nous venons de voir que l'adaptation de l'algorithme d'allocation à nos conditions reste très légère. Néanmoins, gérer deux tas sur un même espace mémoire au lieu d'un entraîne une autre conséquence. Les conditions normales d'utilisation d'un allocateur, comme vu au chapitre 2, impliquent que celui-ci ne doit pas échouer. Par échouer nous entendons ici retourner un pointeur NULL à l'application, entraînant généralement l'échec de celle-ci, voir du système tout entier. Si un OS était présent, l'allocateur mémoire lui demanderait plus de mémoire à ce moment-là, mais dans nos conditions, sur machine nue, ce n'est pas le cas.

Or, nous séparons la mémoire dédiée aux objets du tas en deux zones mémoires, gérées par deux tas distincts. Il en découle que si le dispatcheur essaye d'allouer dans un tas n'ayant plus de place, l'algorithme ne va échouer qu'après une recherche coûteuse en temps. La figure 7.2 décrit les différents cas possibles, l'allocateur étant prévu pour aller vite dans le cas n° 1 et éviter le cas n° 3.

Il est important de noter ici que la notion d'échec pour l'allocateur mémoire n'a pas le même sens dans le cas du multi-tas. En effet, échouer à allouer dans un système à un seul tas signifie l'échec du programme, voire du système. Dans le cas du multi-tas, échouer à allouer dans un tas

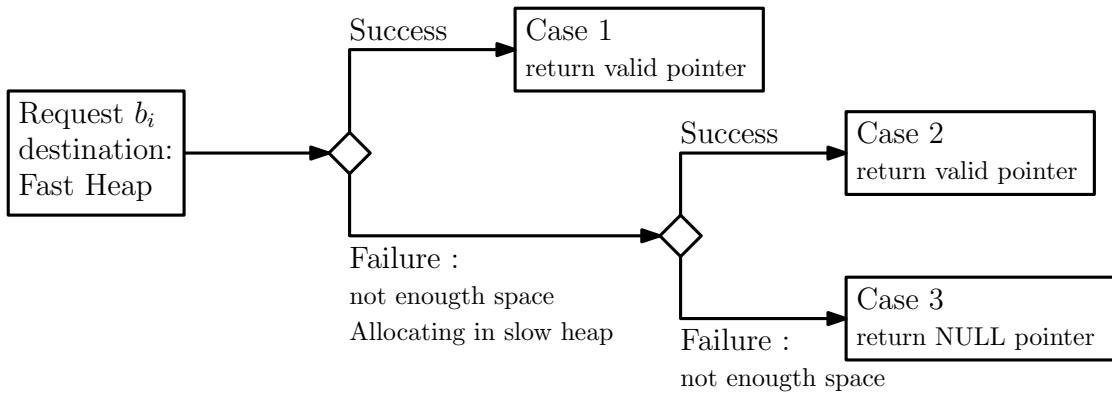


FIGURE 7.2 – Mécanisme de fallback pour l'allocation mémoire multi-tas

signifie seulement qu'il va falloir essayer d'allouer dans un autre tas (cas n° 2). Le cas d'échec réel, conduisant à retourner l'adresse zéro à l'application n'arrive que si aucun des tas du système ne peut satisfaire la requête (cas n° 3). De plus les performances d'un allocateur mémoire dynamique sont moins bonnes quand le tas est surchargé et fragmenté, rendant cet impact négatif encore plus important. Sachant que nous dimensionnons la taille du tas à la taille de l'empreinte mémoire de l'application, il devient dès lors évident que la majorité des exécutions peuvent être amenées à subir cet impact. Nous proposons donc une adaptation de l'algorithme de gestion mémoire dynamique l'adaptant aux conditions d'utilisation multi-tas (cas n° 2).

Nous avons modifié l'algorithme DLmalloc de telle manière qu'il garde dans la structure de données de chaque tas l'information de "plus grand bloc disponible" dans ce tas. Cette information permet de rapidement se rendre compte que l'allocation n'est pas possible dans ce tas, sans parcourir toute la structure de données décrivant son état.

DÉFINITION - *early fail* : mécanisme d'allocation mémoire permettant de refuser une requête d'allocation que le tas ne peut pas satisfaire sans avoir à parcourir l'intégralité de la structure de données le décrivant.

Cette modification permet d'adapter l'algorithme d'allocation mémoire aux conditions du multi-tas. Néanmoins, le calcul de cette information représente un coût important pouvant à lui seul représenter une dégradation significative des performances. Nous avons donc choisi d'implémenter cette modification de manière paresseuse. Nous attendons donc que l'allocation échoue une fois dans un tas pour calculer la taille du plus grand bloc de ce tas. Nous vérifions aussi à la libération d'un objet que la taille du bloc formé après fusion avec ses voisins n'augmente pas cette valeur.

L'implémentation que nous avons mise en œuvre est évaluée au chapitre 8 et permet de garder un temps passé dans l'allocateur mémoire raisonnable.

7.2 Cas d'étude logiciels

Dans cette section nous allons justifier le choix des applications cibles pour notre étude. Partant des suites de benchmarks académiques pour l'embarqué nous n'avons pu retenir que les applications utilisant significativement la gestion mémoire dynamique. Nous allons donc justifier et illustrer ce choix à l'aide de l'application jpeg. Par la suite nous discuterons de chaque application sélectionnée, ainsi que des différents jeux de données utilisés.

7.2.1 Suites applicatives académiques pour l'embarqué

MiBench

La suite de benchmarks MiBench [GREAMB01] présente de nombreuses applications représentatives du domaine de l'embarqué. Elle couvre de nombreux cas d'utilisation de tels systèmes. Les applications qu'elle contient couvrent les domaines suivants : contrôle industriel et automobile, réseau, sécurité, multimédia, utilitaires de texte et télécommunications.

Ces applications ont des usages très variables des ressources de la plateforme et sont représentatifs de cas réels de l'embarqué. Toutefois l'allocation mémoire dynamique ne fait pas partie des préoccupation des auteurs. Nous n'en avons tiré qu'une application, Dijkstra, comme nous allons le voir plus loin.

Mediabench

L'autre suite de benchmarks académique que nous avons utilisé est la suite Mediabench [FSTW09]. Il s'agit d'une suite d'applications multimédia destinées à l'embarqué. De cette suite nous avons tiré deux applications multimédia : l'encodage vidéo H263 et la compression jpg2000. Contrairement à la suite Mibench, plus d'application de cette suite auraient pu être intégrés, mais nous souhaitons éviter de sur-représenter les applications multimédia dans notre étude.

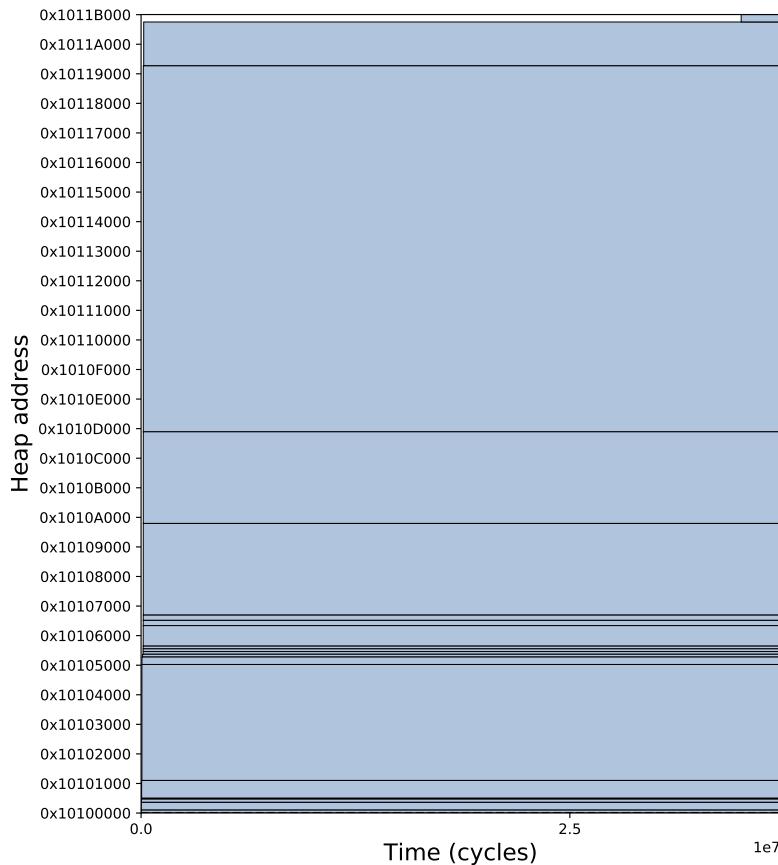


FIGURE 7.3 – Objets du tas : **jpeg**, exécution de référence, jeu de données 0

7.2.2 Sélection applications embarquées à étudier

De l'ensemble des applications de ces deux suite de benchmarks nous tirons donc seulement trois applications. Pour l'expliquer nous allons tout d'abord illustrer l'utilisation de la mémoire dynamique d'une application couramment utilisée dans la littérature. L'algorithme de compression jpeg est une application de la suite Mibench, notamment utilisée pour l'étude des mécanismes logiciels de gestion de mémoire hétérogène pour l'embarqué. Cet exemple montre que dans le cas d'une application ne faisant pas un usage significatif de la gestion mémoire dynamique, notre approche n'est pas nécessaire.

Exemple d'une application utilisant peu la gestion mémoire dynamique

L'application jpeg est une bonne illustration d'une application dont l'utilisation de la gestion mémoire dynamique est non significative. D'une part il existe des implémentations ne l'utilisant pas

– notamment à destination du domaine de l'embarqué [Gel10]. D'autre part, le profil d'allocation et le profil d'accès de l'application ne présente pas d'aspect dynamique significatif (à part de la taille des objets alloués) dépendant du jeu de données.

La figure 7.3 présente le comportement du tas de cette application. Chaque requête émise par l'application y est représenté par un rectangle dont les dimensions correspondent à son occupation spatio-temporelle du tas. L'axe des abscisses représente donc le temps en cycles processeur simulés, et en ordonnée les adresses du tas. Chaque rectangle occupe donc une surface correspondant aux adresses allouées durant sa durée de vie.

Dans le cas de jpeg nous remarquons en premier lieu que l'ensemble des objets sont alloués au début de l'exécution et libérés à la fin (ou non libérés). Ce comportement d'allocation mémoire ne donne donc lieu à aucune réutilisation de la mémoire du tas. Il est intéressant de noter qu'une exécution de l'application n'alloue que 21 objets, et ce nombre ne varie pas en fonction du jeu de donnée. De plus ces objets sont toujours alloués dans le même ordre. Ces conditions sont suffisamment simples pour qu'un développeur puisse résoudre le problème de placement à la main.

Nous n'avons donc pas pris en compte les applications exhibant des comportements simplistes de ce type pour une étude centrée sur la gestion mémoire dynamique.

7.2.3 Applications cibles

Nous présentons ici les applications embarquées utilisées pour cette étude et les suites de benchmarks dont elles sont issues. Pour chaque application, la description que nous en donnons contient les éléments suivants :

- empreinte mémoire du tas
- nombre d'objets
- ratio de lecture/écriture du tas
- ratio de réutilisation de la mémoire du tas : définie comme la taille totale allouée divisée par l'empreinte mémoire.

Autres applications

N'ayant pu mettre en œuvre qu'un nombre limité d'applications issues de suites de benchmarks de la littérature, nous avons cherché dans les usages du domaine de l'embarqué des exemples d'utilisation de la gestion mémoire dynamique. Nous avons de cette manière ajouté deux applications à nos cibles logicielles. D'une part une application permettant de sérialiser et dé-sérialiser des fichiers json, couramment utilisés dans le domaine de l'embarqué, que ce soit pour le stockage de configurations ou pour l'échange de données. D'autre part nous avons utilisé une application issue d'une bibliothèque de sécurité destinée au domaine de l'embarqué contraint. Ces applications nous semblent en accord avec les usages et les besoins de l'industrie notamment pour le développement de l'IoT.

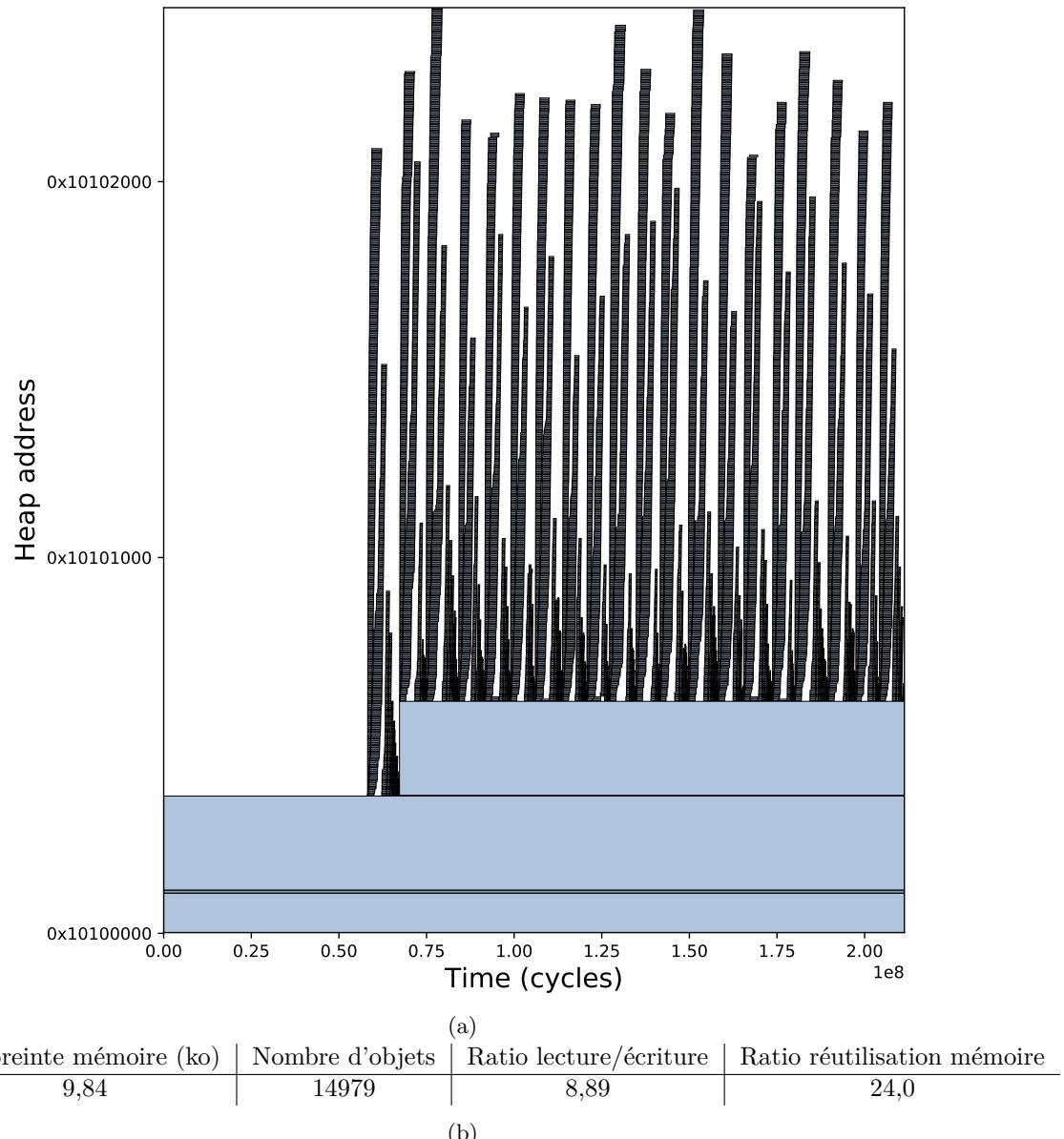


FIGURE 7.4 – Application **Dijkstra** (MiBench) : exécution de référence, jeu de données 0.
 (a) Visualisation de l'occupation du tas et (b) description des caractéristiques du tas de l'application

Dijkstra Pour cette implémentation de la recherche de plus court chemin, les données décrivant un graphe sont stockés dans une matrice d'adjacence. L'application utilise ensuite de nombreuses fois l'algorithme éponyme, calculant le plus court chemin entre chaque paire de noeuds du graphe. La figure 7.4b présente les caractéristiques du tas de l'exécution du jeu de données fourni avec l'application. Le graphe d'occupation du tas relatif à l'exécution de référence de ce jeu de données est présenté en figure 7.4a. Les différents pics d'allocation correspondent aux différentes itérations de l'algorithme.

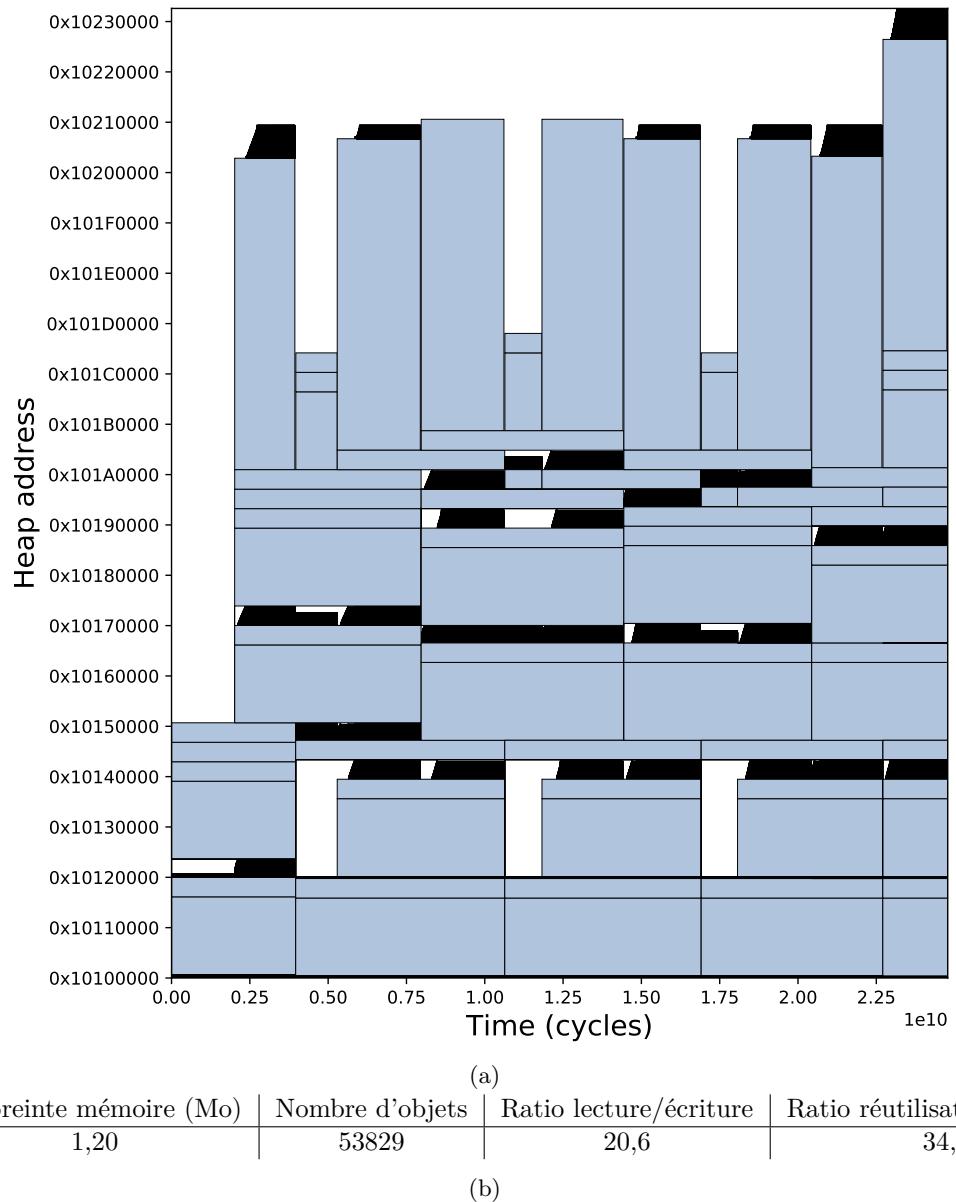


FIGURE 7.5 – Application **H263** (Mediabench) : exécution de référence, jeu de données 0.
(a) Visualisation de l'occupation du tas et (b) description des caractéristiques du tas de l'application

H263 : L'encodage H263 a été développé comme un standard de compression vidéo destiné à des échanges à faible débits pour la visioconférence. La figure 7.5b décrit ses caractéristiques et la figure 7.5a présente une visualisation du tas de l'application. Un même schéma d'allocation se répète plusieurs fois dans le temps, il correspond à l'encodage d'images successives. Il est important de noter que les zones apparaissant noires sur la figure correspondent à l'empilement de nombreux objets d'une faible taille.

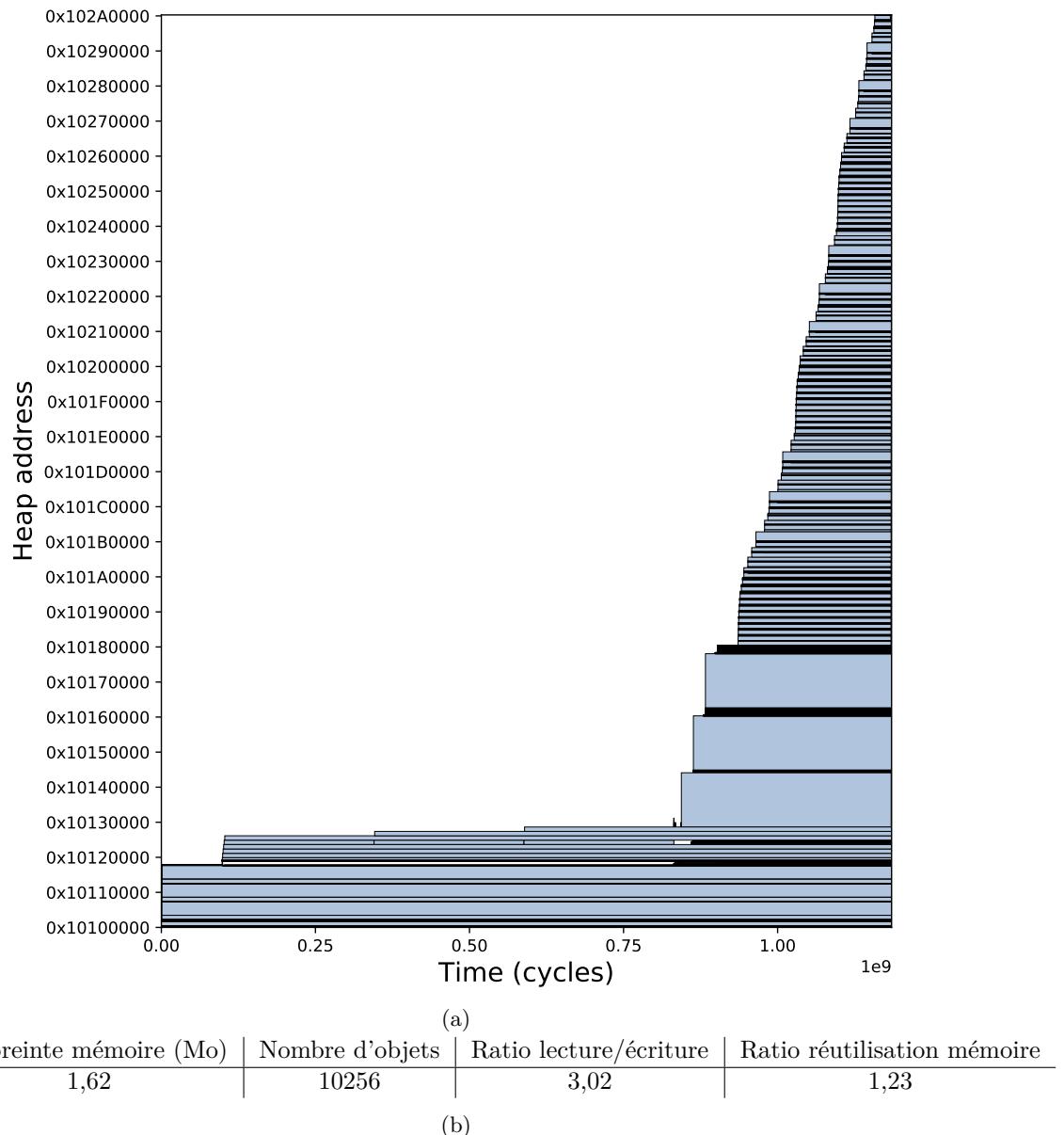
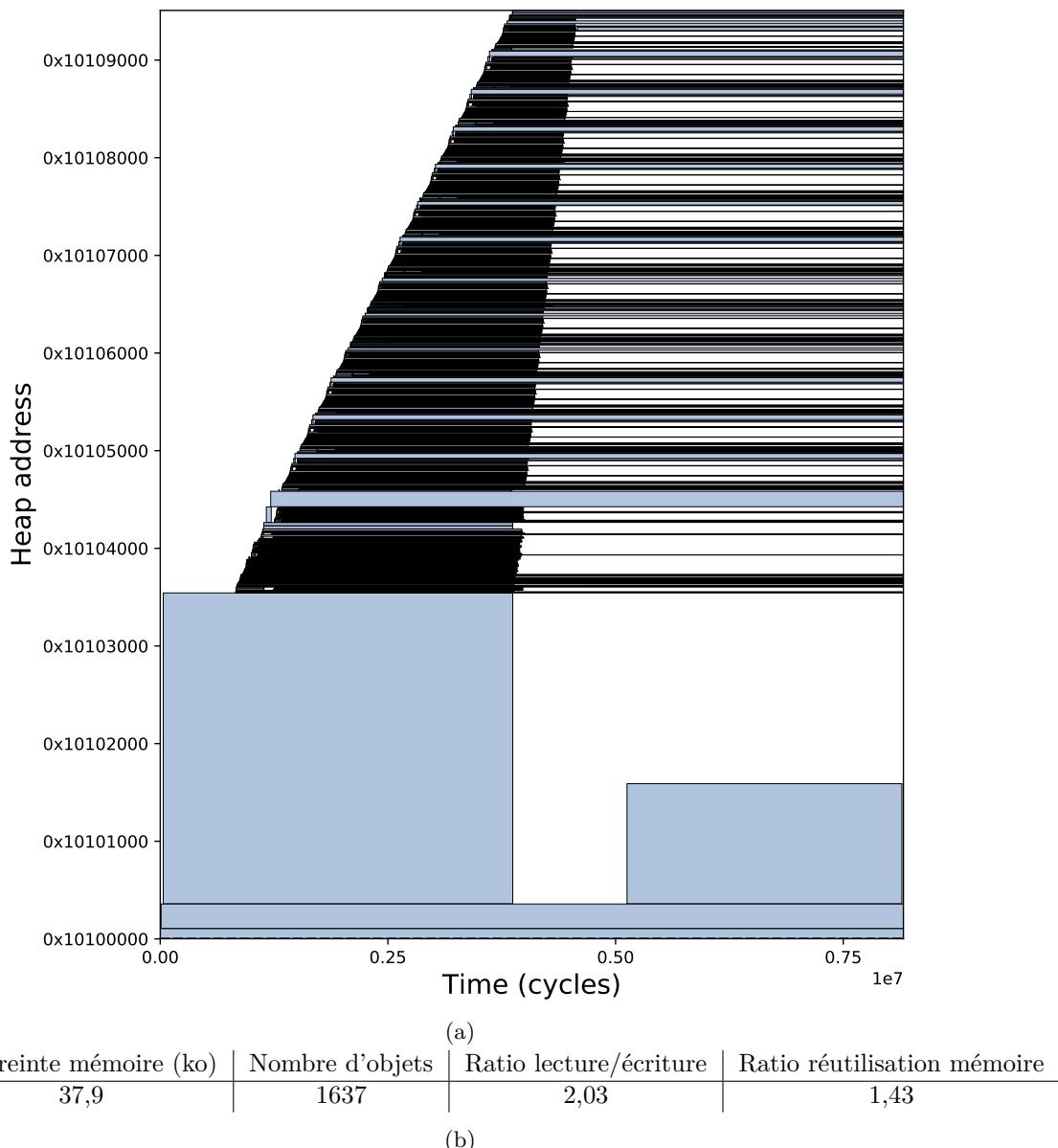


FIGURE 7.6 – Application **Jpeg 2000** (Mediabench) : exécution de référence, jeu de données 0.
(a) Visualisation de l'occupation du tas et (b) description des caractéristiques du tas de l'application

Jpeg 2000 : Le format d'image Jpeg 2000 correspond à une norme de compression d'image pouvant travailler avec ou sans pertes. Il est basé sur la transformée en ondelettes, et présente un meilleur taux de compression que JPEG. La figure 7.6 présente les caractéristiques de l'exécution de référence et le tracé de son l'occupation du tas. Cette application présente le comportement typique d'une rampe pour l'allocation mémoire dynamique, accumulant un nombre croissant d'objets alloués jusqu'à avoir fini la tâche de l'application. Bien que présentant un taux de réutilisation de la mémoire plutôt faible, ce type de comportement est fonction des jeux de données d'entrée qui influe sur le nombre d'objet ainsi que sur leur taille.

FIGURE 7.7 – Application **json** : exécution de référence, jeu de données 0.

(a) Visualisation de l'occupation du tas et (b) description des caractéristiques du tas de l'application

Analyse syntaxique json : Json (Java Script Object Notation) est un format de fichier permettant de transmettre et stocker des valeurs sérialisées. L'utilisation de ce format est courante pour l'embarqué, et passé généralement par l'utilisation d'une bibliothèque tierce partie. Nous avons donc utilisé une librairie tierce partie libre [Gab12] destinée aux usages de l'embarqué et codée en langage C. Autour de cette librairie, nous avons bâti une application réalisant l'analyse syntaxique d'un fichier json, construisant en mémoire une représentation de celui-ci. Cette représentation est ensuite parcourue et l'application filtre la majorité des clefs, le résultat est ensuite sérialisé vers un buffer de sortie. Par soucis de commodité nous désignerons cette application json dans ce manuscrit.

Cette application, bien qu'ayant un profil d'accès au données relativement peu intensif, présente un profil d'allocation complexe. La figure 7.7b présente les caractéristiques de l'exécution de référence de l'application pour notre jeu de données de base, et la figure 7.7a présente une visualisation du tas de cette exécution.

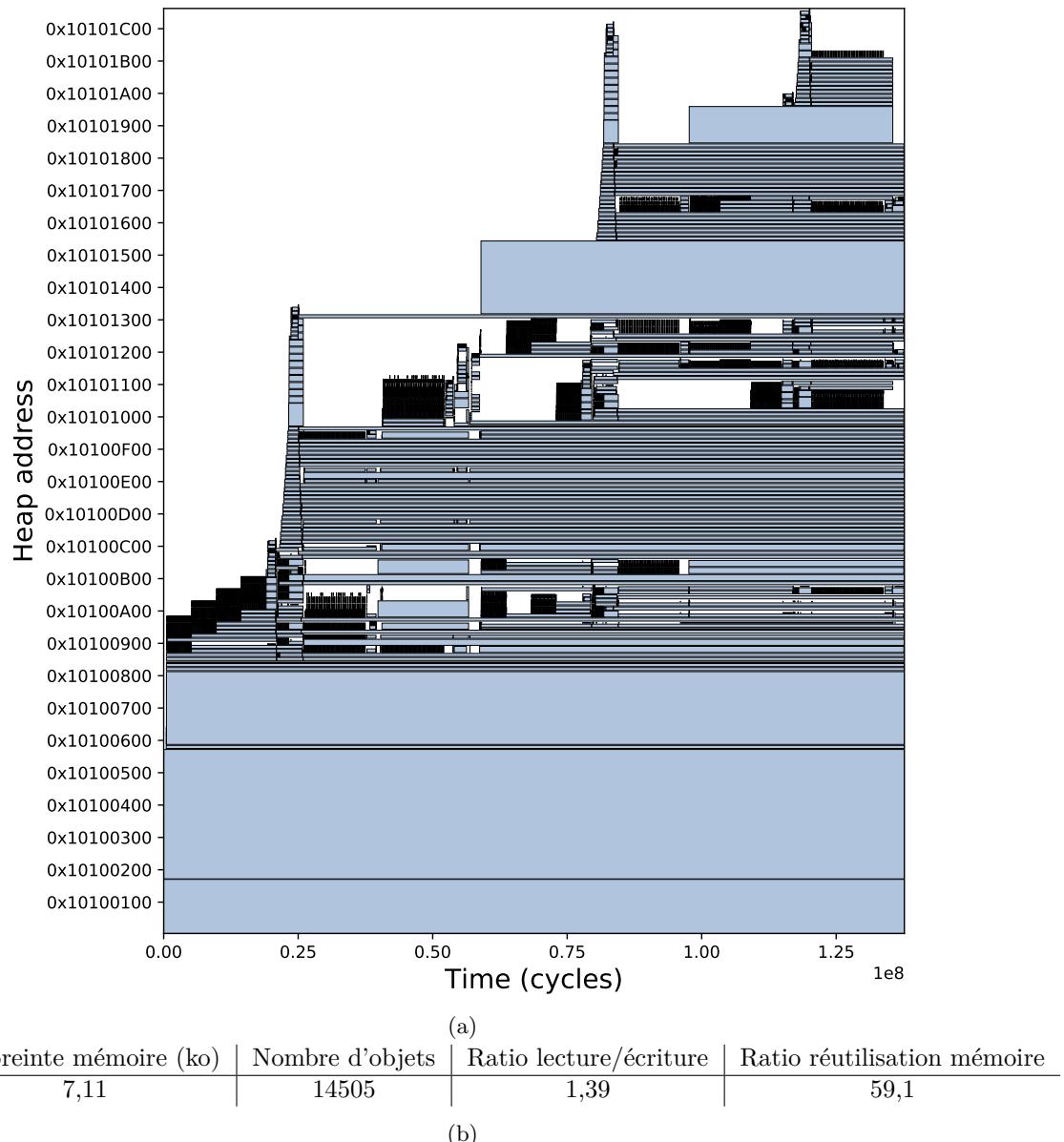


FIGURE 7.8 – Application **Ecdsa** (mbedtls) : exécution de référence, jeu de données 0.
(a) Visualisation de l'occupation du tas et (b) description des caractéristiques du tas de l'application

Ecdsa : La suite mbedtls est une bibliothèque logicielle de chiffrement implémentant les protocoles SSL et TLS. Elle est développée par ARM et est largement utilisée, notamment pour la sécurité des applications IoT. L'un des algorithmes de signature numérique, utilisant la cryptographie sur les courbes elliptiques, fait une utilisation intensive de la gestion mémoire dynamique. Pour des raisons d'offuscation des opérations liées à la sécurité, la librairie n'alloue pas d'objets au sens commun du terme mais utilise le tas pour stocker des nombres entier de grandes taille. L'application utilisée génère une clef de chiffrement et encore un message, puis vérifie la que signature numérique est correctement réalisée. La figure 7.8 présente les caractéristiques du tas de l'application et ainsi que son occupation mémoire durant l'exécution.

7.2.4 Jeux de données

Les benchmarks de la littérature utilisés ne proposent pas ou peu de variété dans les données d'entrée des applications. En effet, l'influence sur le comportement de l'application vis à vis de

7.3. CONCEPTION D'UNE MÉTRIQUE EFFICACE : FRÉQUENCE D'ACCÈS PAR OCTET

son accès au code ou aux variables non dynamiques n'est que peu influencé par le jeu de données d'entrée. Dans de nombreux cas l'analyse statique du code permet de décider de quelles variables ou quelles sections de code méritent d'être placées dans la mémoire rapide. Pour notre cas d'étude cependant nous savons que le profil d'allocation et le profil d'accès au tas d'une application est fonction du jeu de données d'entrée. Nous avons donc bâti des jeux de données variés pour chaque application dont la description et les caractéristiques sont fournies en annexe.

Nous avons donc évalué le comportement des applications et l'impact du placement pour huit jeux de données distincts. Pour chaque application, chaque jeu de données fait varier les paramètres impactant le travail de l'application. Pour jpeg 2000 par exemple nous avons fait varier la taille des données d'entrée, mais aussi les images (plus ou moins de hautes fréquences, couleurs). Pour json, nous avons fait varier la largeur de l'arbre json à analyser, sa profondeur et son contenu (longues chaînes de caractères, tableaux d'entiers, ...)

7.3 Conception d'une métrique efficace : Fréquence d'accès par octet

Pour répondre efficacement au problème de placement nous souhaitons être capable de prendre la bonne décision au moment de l'allocation. Pour ce faire nous étudions le comportement des applications à posteriori. Néanmoins, il n'est pas trivial de décider quel objet est important pour la résolution du problème de placement juste à partir du profil d'allocation et du profil d'accès de l'application. Nous avons donc élaboré une métrique nous permettant de classer les objets par chaleur, relativement à la place qu'ils occupent dans le tas. L'utilisation de cette métrique nous permet de comparer les objets pour déterminer lequel aurait le plus d'impact sur les performances placé dans le tas rapide.

Le but d'une telle métrique est de permettre un classement qui, utilisé pour prendre des décisions de placement pour une exécution équivalente présenterait une amélioration des performances significative par rapport à l'exécution de référence. Nous n'avons toutefois pas pour but de calculer la valeur de cette métrique durant l'exécution.

Nous pouvons classer les informations à prendre en compte en deux catégories : les aspects influençant les performances de l'application, et les aspect présentant un impact sur l'efficacité et ou l'occupation du tas.

Pour les impacts directs sur les performances de l'application, nous prenons deux informations en compte. Dans un premier temps, nous considérons le nombre de l'accès à l'objet, en séparant lecture et écriture. En effet nous avons vu que c'est l'hétérogénéité de l'accès aux objets qui permet de tirer partie du problème de placement pour améliorer les performances de l'application. Nous prenons également en compte les latences des technologies mémoires. En effet les latences en lecture et en écriture des bancs mémoire dans lesquels le dispatcheur va placer les objets influencent les décisions de placement.

Mais nous souhaitons également prendre en compte l'occupation du tas. En effet, nous ne souhaitons placer des objets dans le tas rapide, limité en taille, que si les gains en performances que ce placement apporte sont plus important que l'impact négatif sur le tas de l'objet. Nous prenons donc en compte pour un objet sa taille et sa durée de vie.

Il est à noter que, comme vu au chapitre 2, l'allocateur mémoire dynamique n'a pas de représentation du temps. Nous souhaitons tout de même de prendre en compte la durée de vie de l'objet représentant le temps durant lequel l'objet occupe l'espace qui lui est alloué dans le tas. Dans ces conditions nous définissons la notion d'overlap (recouvrement) pour les objets du tas :

DÉFINITION - *overlap* : nombre d'objets alloués durant la durée de vie d'un objet, incluant l'objet lui-même.

En effet, un objet peut avoir une durée de vie longue, si aucun objet n'est alloué pendant cette durée de vie, les adresses qu'il occupe n'auront jamais été occupées pour l'algorithme de gestion mémoire dynamique pendant une allocation. A l'inverse, un objet ayant une durée de vie courte, mais durant laquelle sont alloués de nombreux objets influe de nombreuse fois sur la résolution du problème d'allocation mémoire. De plus cet objet à la durée de vie plus faible laissera potentiellement le tas plus fragmenté lors de sa libération, d'autres objets ayant été alloués durant sa durée de vie, impactant par là l'allocation de futures requêtes. Nous préférons donc cette notion d'overlap à l'intuition d'occupation temporelle du tas.

Nous décrivons l'architecture mémoire cible pour le tas de la manière suivante :

$$M = \{m_0, m_1, \dots, m_n\}$$

Dans le cadre de cette thèse $|M| = 2$, et le tas est composé de mémoire rapide et de mémoire lente. Nous définissons donc

$$M = \{m_f, m_s\}$$

avec m_f (resp. m_s) le banc composée de mémoire rapide (resp. lente)

Pour l'exécution de référence d'une application avec un jeu de données spécifique on obtient un profil d'allocation et un profil d'accès. On nomme B l'ensemble des objets de cette exécution de référence et on note :

- $count_R(b)$: nombre d'accès en lecture à l'objet b durant l'exécution
- $count_W(b)$: nombre d'accès en écriture à l'objet b durant l'exécution
- $size(b)$: taille de l'objet b en octets

Nous construisons donc la fréquence d'accès par octet de la manière suivante :

$$freq_{octet}^{acces}(b) = \frac{count_R(b) \times (lat_R(m_s) - lat_R(m_f)) + count_W(b) \times (lat_W(m_s) - lat_W(m_f))}{size(b) \times overlap(b)}$$

Avec $overlap(b)$ le nombre d'objets alloués pendant la durée de vie de b (b inclus).

Cette métrique prend en compte l'importance des accès à un objet, pondéré par la différence de latence entre les technologies mémoire ciblées. Elle permet également de prendre en compte l'asymétrie potentielle de ces technologies. L'impact négatif d'un objet sur l'occupation du tas est aussi prise en compte, tant au niveau de la place requise par l'objet que de son interaction avec l'allocation d'autres objets.

Nous présenterons des résultats d'évaluation de cette métrique au chapitre 8. Nous allons maintenant étudier la manière des applications d'allouer des objets et d'y accéder.

7.4 Comportement d'allocation des applications

Nous avons déterminé quelles informations nous semblaient importantes sur les objets pour jauger leur importance. Nous les avons synthétisées dans la métrique de fréquence d'accès par octet. Mais leur utilisation pour une stratégie en ligne n'est toujours pas immédiat. En effet, nous ne pouvons pas mesurer les informations nécessaires à son calcul durant l'exécution notamment le nombre d'objets alloués pendant la vie d'un objet.

Nous allons donc caractériser hors ligne le comportement de nos applications à partir des profils des exécutions de références pour les lier aux données accessible au moment de l'allocation. De cette manière nous pouvons faire émerger des règles applicables en ligne pour guider le choix de placement par le dispatcheur. Autrement dit, ne pouvant calculer la fréquence d'accès par octet en ligne, nous ne pouvons espérer l'utiliser que si nous trouvons des informations mesurables en ligne qui lui sont corrélées.

Les informations disponibles en ligne au moment de la requête d'allocation faite au dispatcheur par l'application sont limitées. La première information disponible à l'appel du dispatcheur est la taille de la requête d'allocation. En effet celle ci est l'unique paramètre de l'API d'allocation mémoire. Une autre information peut être facilement récupérée durant l'exécution, il s'agit de l'adresse de retour vers l'application. Cette adresse, stockée dans le registre du processeur destiné à cet effet nous permet d'obtenir sans calcul l'adresse du site d'allocation. Des travaux précédents ont déjà mis en avant la pertinence de cette information pour discriminer les objets intéressants pour la résolution du problème de placement [DUB05 ; WCLL18].

Nous allons donc dans la suite étudier le profil du tas des applications cibles. En calculant à posteriori la fréquence d'accès par octet des objets du profil puis en visualisant ces valeurs par classe de taille nous pouvons essayer de distinguer des caractéristiques d'allocations des applications liées à la taille des objets alloués.

Dans un second temps, nous réaliserons cette même étude en considérant l'information du site d'allocation de chaque objet.

7.4.1 Étude de la fréquence d'accès des objets en fonction de leur taille

Les figures 7.9, 7.10, 7.11, 7.12 et 7.13 présentent les résultats de l'analyse des objets par classe de taille des exécutions de référence. Pour chaque application, la fréquence d'accès de chaque objet de l'exécution de référence est calculée puis présentée en fonction de la taille de l'objet. Dans le premier graphe, chaque objet est représenté par un point ayant pour coordonnées sa taille (en abscisse) et sa fréquence d'accès par octet (en ordonnée), Le tracé des points n'est pas opaque, permettant de rendre visibles plusieurs objets ayant la même taille et la même fréquence d'accès par octet. Le second et le troisième graphe de chaque application travaillent sur des classes de taille logarithmiques, pour permettre la représentation des très grands écarts de taille que peut avoir à traiter un gestionnaire mémoire dynamique. Le second graphe présente la valeur moyenne de la fréquence d'accès par octet au sein de chaque classe de taille. Le troisième quant à lui est un histogramme de distribution des objets par classe de taille, facilitant la lecture des deux autres graphiques.

L'analyse du deuxième graphe de chaque figure ne présente pas de corrélation entre la taille des objets et leur fréquence d'accès. Nous en déduisons qu'il n'y a pas de lien direct entre ces deux informations. On distingue néanmoins pour les applications H263, Jpeg 2000 et Ecdsa des groupes d'objets de taille identiques présentant tous une fréquence d'accès par octet haute. Mais la taille de la requête d'allocation ne semble pas pouvoir être un critère de distinction suffisant étant donné que d'autres objets de la même taille sont souvent alloués par les applications.

On constate également que l'application Dijkstra, bien que présentant un taux de réutilisation de la mémoire du tas élevé, n'alloue pas des objets de tailles différentes.

Le jeu de données impacte le profil d'accès et le profil d'allocation des applications. Mais cet impact ne touche pas forcément l'intégralité des objets alloués par l'application. Les analyses des objets pour les autres jeux de données d'entrée sont présentées en annexe III. Nous pouvons néanmoins tirer les conclusions suivantes. La taille de certains objets est insensible au changement du jeu de données. En effet certains sont directement liés à la taille des données d'autres à la complexité de l'algorithme à traiter ces données. Un jeu de données différent peut changer le chemin d'exécution en plus d'avoir (ou non) une taille différente. Nous déduisons de cette étude qu'une méthode basée sur la taille des objets pour prédire leur fréquence d'accès par octet n'est pas pertinente.

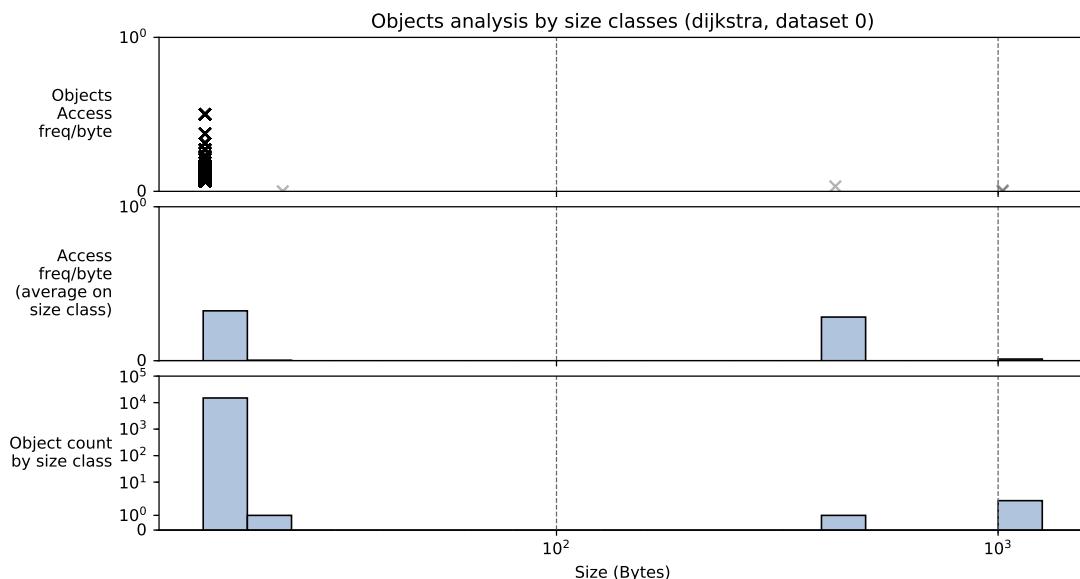


FIGURE 7.9 – **Dijkstra** : analyse des objets par classe de taille – exécutions de référence. De haut en bas : Fréquence d'accès par octet en fonction de la taille de l'objet, Fréquence d'accès par octet moyenne par classe de taille et histogramme de distribution des objets par classe de taille

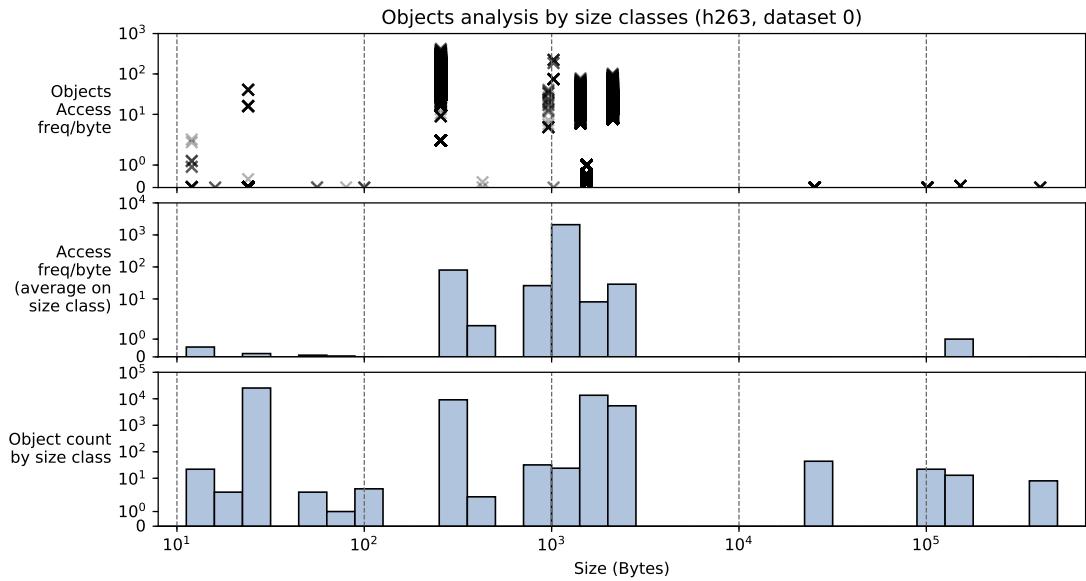


FIGURE 7.10 – H263 : analyse des objets par classe de taille – exécutions de référence.
De haut en bas : Fréquence d'accès par octet en fonction de la taille de l'objet, Fréquence d'accès par octet moyenne par classe de taille et histogramme de distribution des objets par classe de taille

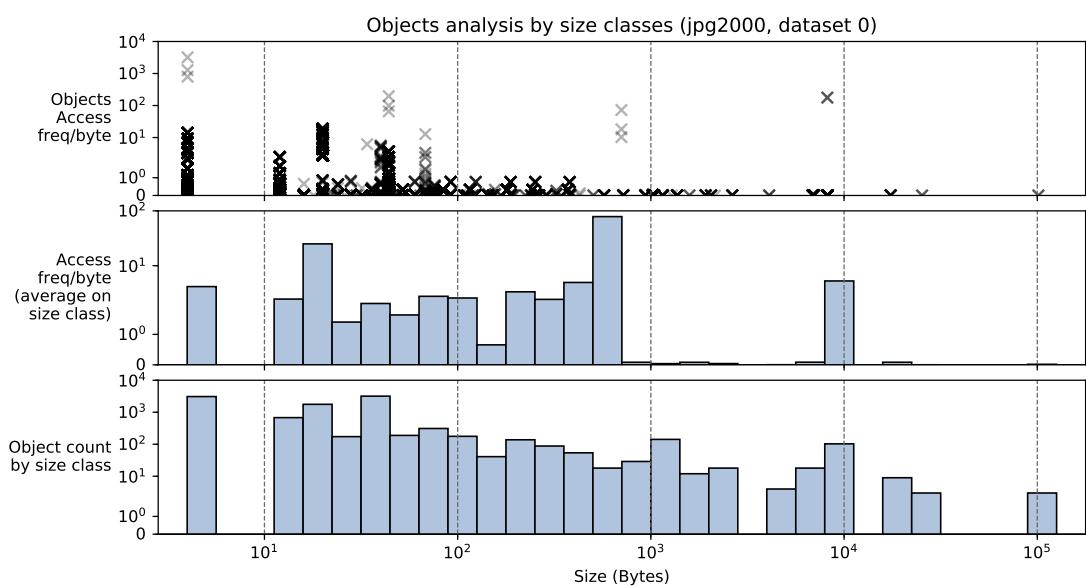


FIGURE 7.11 – Jpg2000 : analyse des objets par classe de taille – exécutions de référence.
De haut en bas : Fréquence d'accès par octet en fonction de la taille de l'objet, Fréquence d'accès par octet moyenne par classe de taille et histogramme de distribution des objets par classe de taille

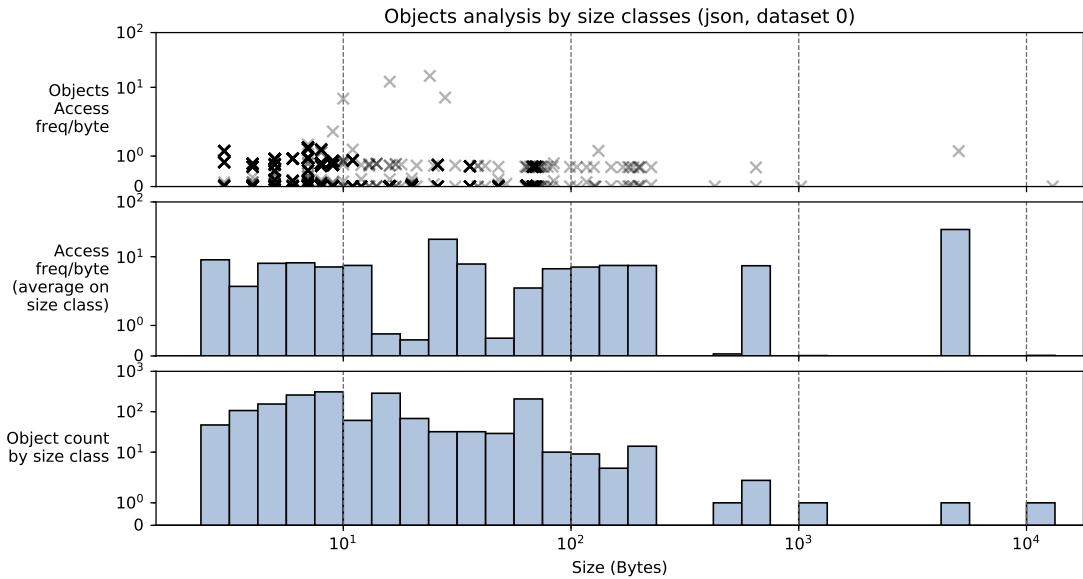


FIGURE 7.12 – Json parser : analyse des objets par classe de taille – exécutions de référence.

De haut en bas : Fréquence d'accès par octet en fonction de la taille de l'objet, Fréquence d'accès par octet moyenne par classe de taille et histogramme de distribution des objets par classe de taille

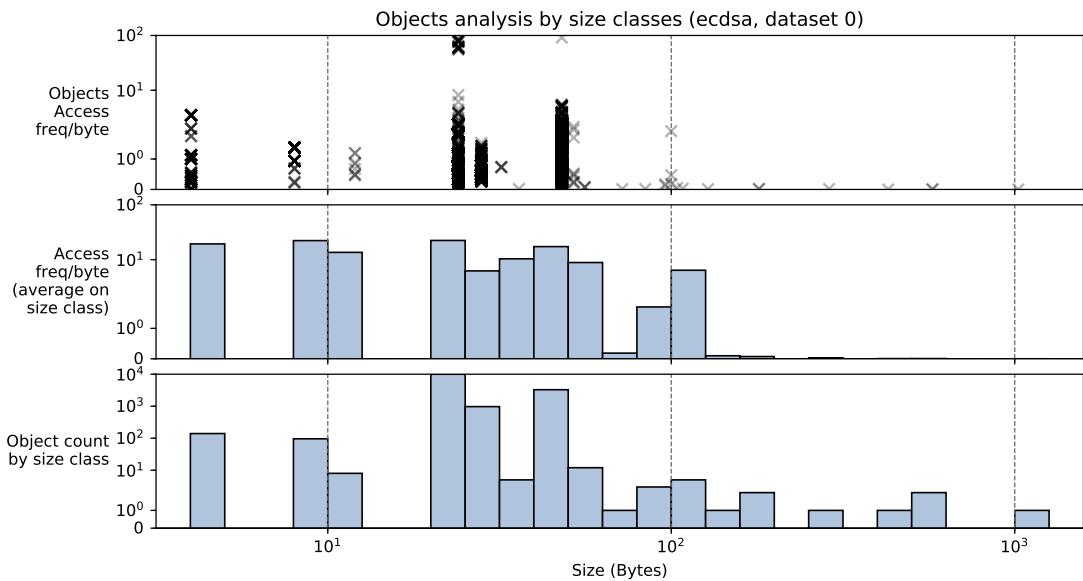


FIGURE 7.13 – Ecdsa : analyse des objets par classe de taille – exécutions de référence.
 De haut en bas : Fréquence d'accès par octet en fonction de la taille de l'objet, Fréquence d'accès par octet moyenne par classe de taille et histogramme de distribution des objets par classe de taille

7.4.2 Étude de la fréquence d'accès des objets en fonction de leur site d'allocation

Nous avons également étudié la fréquence d'accès par octet des objets de nos benchmarks en fonction de leur site d'allocation.

De la même manière que précédemment nous avons étudié la fréquence d'accès par octet des objets, mais cette fois en les groupant par site d'allocation. Les figures 7.14, 7.15, 7.16, 7.17 et 7.18 présentent donc l'analyse des objets par site d'allocation. Les trois graphiques sont présentés pour chaque application. Le premier présente la fréquence d'accès par octet de chaque objet, les différents sites d'allocations étant listés en abscisse. Les sites d'allocations sont regroupés par fonction (le nom de la fonction est indiquée) et ordonnés dans le sens des adresses croissantes. Le deuxième graphique agrège ces informations en représentant la moyenne de fréquence d'accès par octet des objets alloués pour chaque site d'allocation. Comme précédemment le dernier graphe est un histogramme de distribution des objets mais cette fois par site d'allocation.

Le premier point que nous constatons est que certaines applications ont un faible nombre de sites d'allocations. Ainsi l'application d'une stratégie d'allocation basée sur cet aspect peut ne pas être facilement applicable pour le cas général.

Malgré cette considération et contrairement à l'étude par classe de taille, certains sites d'allocation se dégagent nettement comme allouant des objets ayant en moyenne une fréquence d'accès par octet bien supérieure à celle d'autres sites.

Par exemple pour l'application H263, la fonction `MotionEstimation` qui possède deux sites semble allouer des objets qu'il serait intéressant de placer dans le tas rapide. Au contraire la fonction `Predict_P` qui alloue sensiblement le même nombre d'objets que chaque site de la fonction `MotionEstimation` n'alloue que des objets ayant une faible fréquence d'accès par octet.

Si l'on regarde l'application Jpeg 2000, un certains nombres de fonctions se dégagent aussi. Certains sites d'allocation sont des candidats intéressants, par exemple `jpc_bitstream_sopen` ou `jas_cmshapmatlut_set_isra.11`. Mais d'autres allouent des objets qui n'auront pas forcément la même fréquence d'accès par octet, par exemple les sites d'allocations de `jas_create_matrix`.

L'utilisation de jeux de données différents a un impact sur les objets alloués et leur fréquence d'accès par octet. Toutefois, le groupement par site est plus résilient à cette perturbation. par exemple pour l'application H263, si nous sélectionnons les sites allouant en moyenne les objets ayant la plus haute fréquence d'accès par octet, ils restent les mêmes dans le même ordre, avec l'exception d'un site n'apparaissant pas dans l'exécution du jeu de données 0. Ces données sont présentées en annexe III.

Pour les applications Dijkstra et Ecdsa cependant une grande majorité des objets étant alloués par un seul site, ce critère de distinction entre les objets peut ne pas être suffisant à première vue.

Malgré l'absence de certains sites pour certaines exécutions, et malgré le fait que certaines applications n'aient pas assez de distinctions entre leurs sites d'allocation, nous soutenons que cette information permet de dresser un profil applicable en ligne pour la résolution du problème de placement.

Sur le plan méthodologique, il est par contre important de sélectionner suffisamment de jeux de données et qu'ils soient représentatifs de l'utilisation cible de l'application.

En effet au delà de faire varier le profil des objets alloués ou leur nombre, certains jeux de données génèrent des chemins d'exécution différents.

Discussion : nombres de sites d'allocation et pile d'appel

La limitation apparue dans l'étude des applications Dijkstra et Ecdsa en terme de site d'allocation peut être levée dans certains cas. En effet dans l'exemple de l'application Ecdsa, l'utilisation de l'allocation mémoire dynamique est réservée à la réservation de mémoire pour le module MPI – Multiple Precision Integer, nombres entiers à précision variable, permettant de gérer de très grands entiers, comme requis pour le chiffrement. L'utilisation classique du tas étant une voie d'attaques par canaux cachés, la librairie de sécurité mbed tls [ARM08] implémente et gère son tas par ailleurs. Cela dit l'utilisation du tas pour cet usage permet de dégager des gains de performance significatifs comme nous le montrerons au chapitre suivant. Mais comme le montre la figure 7.16, la majeure partie des objets de l'application sont alloués par un seul site d'allocation. Dans le cas d'une utilisation du tas depuis un seul site d'allocation, il est envisageable

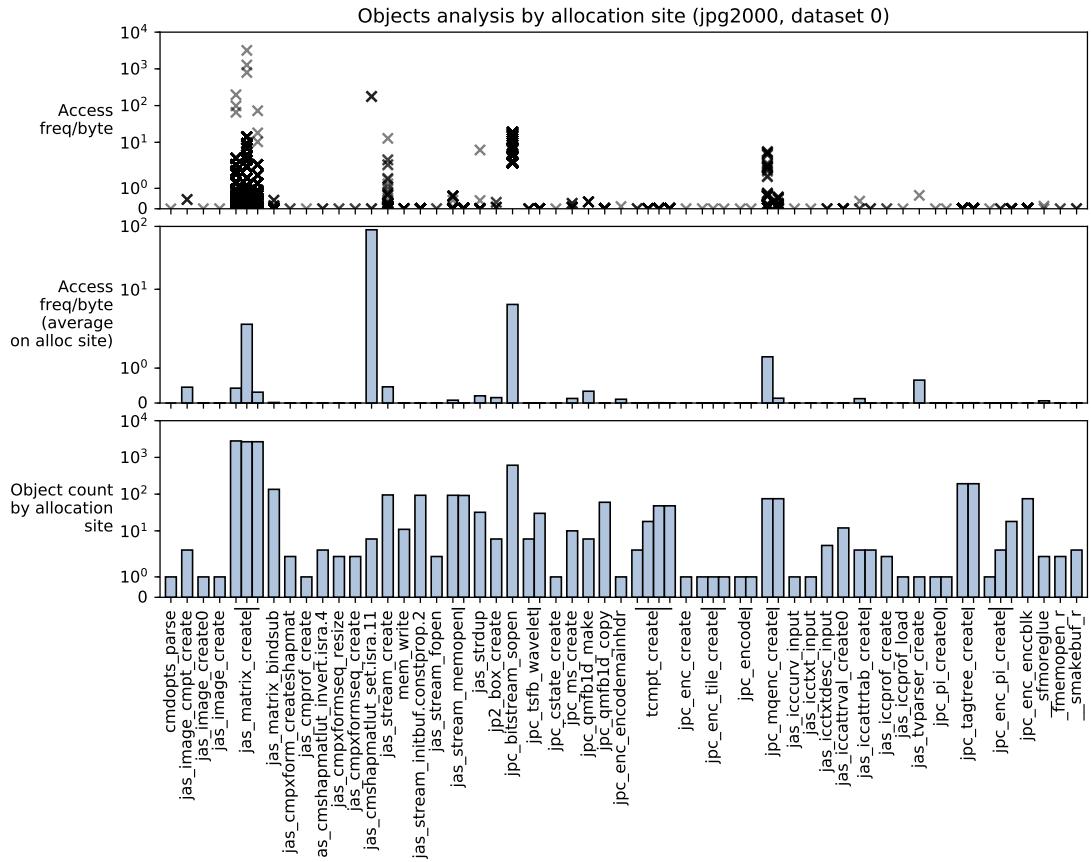


FIGURE 7.14 – Jpg2000 : analyse des objets par site d’allocation – exécution de référence
De haut en bas : fréquence d'accès par octet en fonction du site d'allocation, fréquence d'accès par octet moyenne en fonction du site d'allocation et nombre d'objets par site d'allocation.

de remonter la pile d'appel. Il serait alors possible de distinguer les fonctions appelantes de ce site d'allocation pour discriminer les objets en fonction de leur fréquence d'accès par octets.

Cette piste laissée pour des travaux futurs est néanmoins prometteuse pour lever la limitation d'un profil par site d'allocation ne contenant pas assez de sites. Le coût d'une telle analyse à réaliser durant l'exécution peut néanmoins être important et dépend de l'interface binaire proposée par le jeu d'instruction.

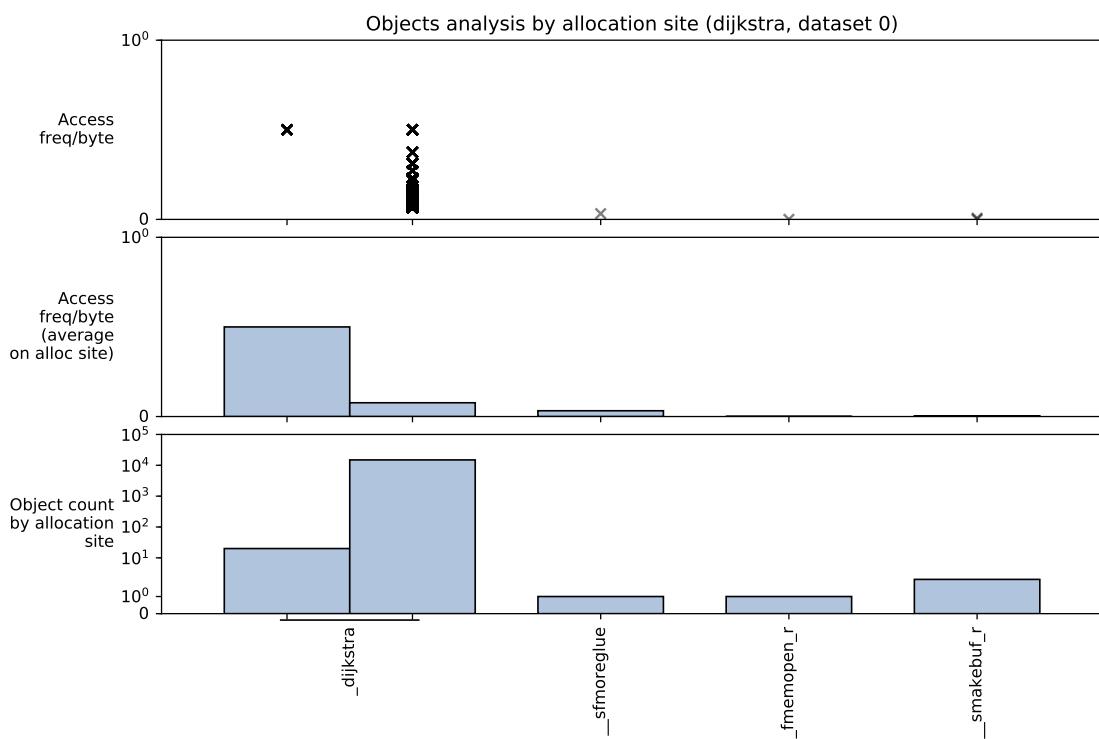


FIGURE 7.15 – Dijkstra : analyse des objets par site d’allocation – exécution de référence
De haut en bas : fréquence d'accès par octet en fonction du site d'allocation, fréquence d'accès par octet moyenne en fonction du site d'allocation et nombre d'objets par site d'allocation.

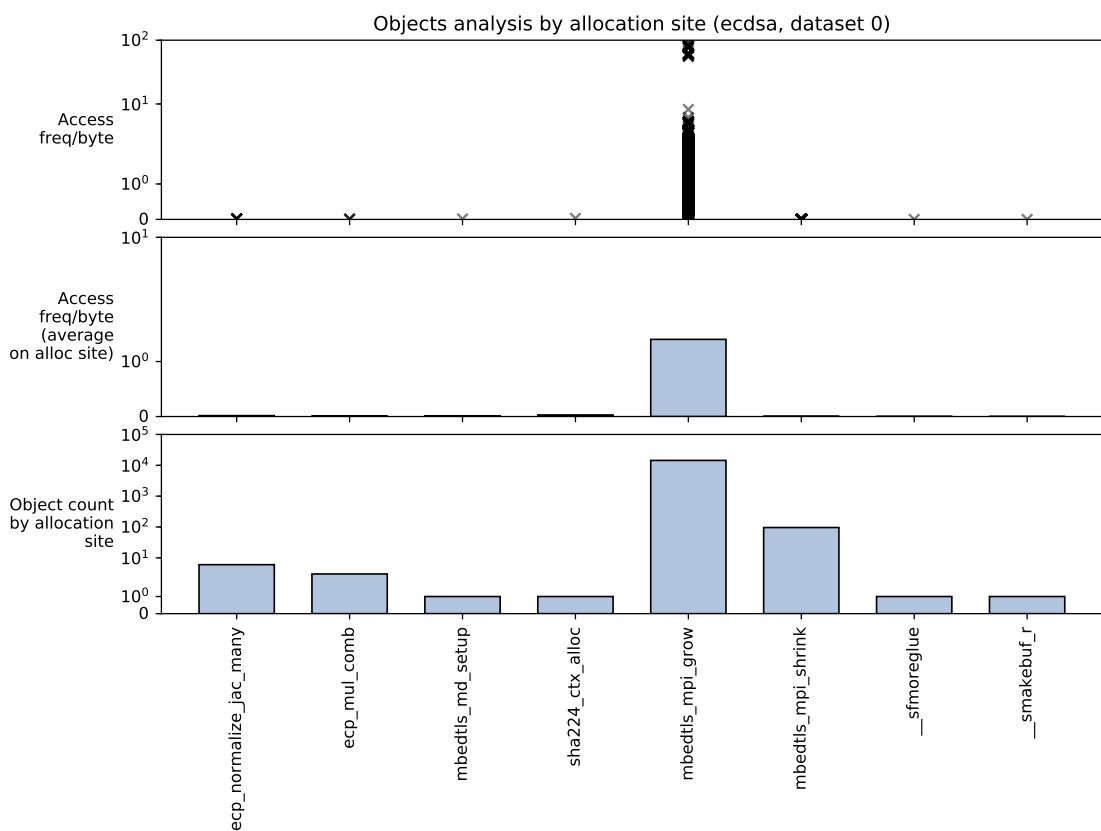


FIGURE 7.16 – **Ecdsa : analyse des objets par site d’allocation** – exécution de référence
De haut en bas : fréquence d’accès par octet en fonction du site d’allocation, fréquence d’accès par octet moyen en fonction du site d’allocation et nombre d’objets par site d’allocation.

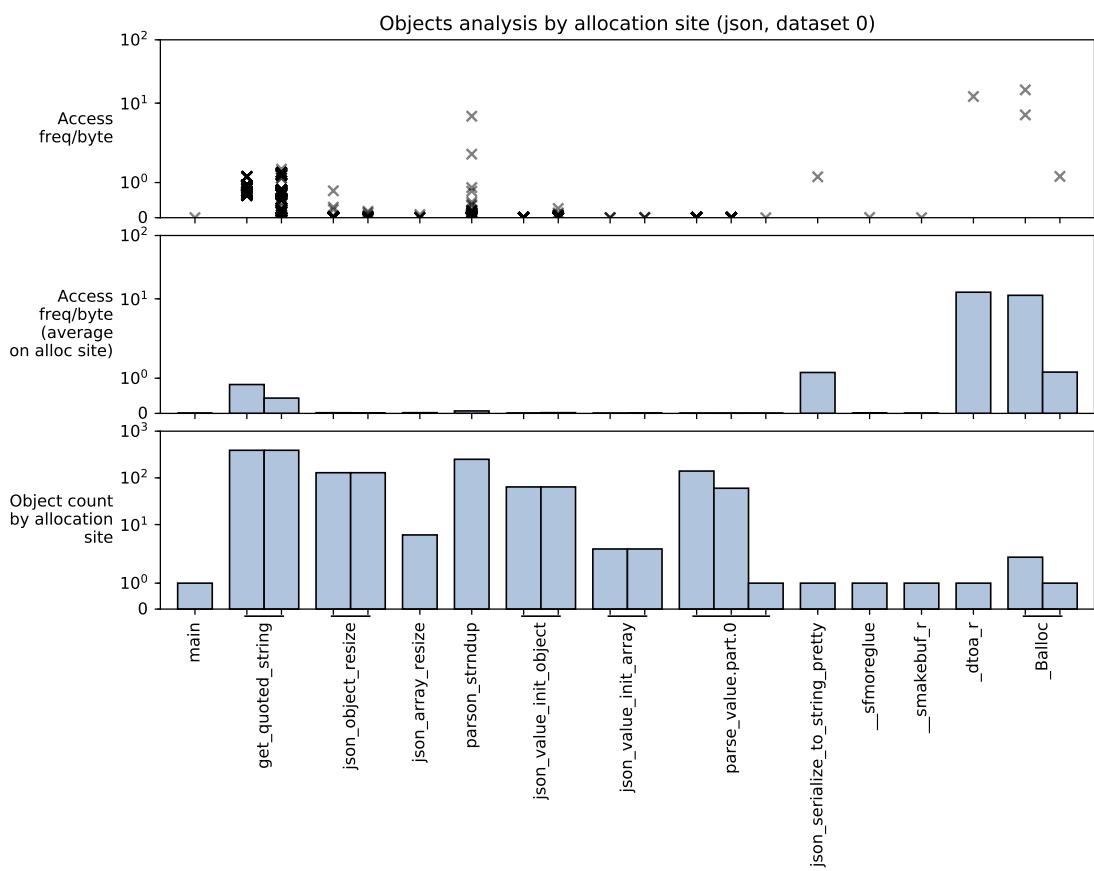


FIGURE 7.17 – **Json parser : analyse des objets par site d’allocation** – exécution de référence

De haut en bas : fréquence d'accès par octet en fonction du site d'allocation, fréquence d'accès par octet moyenne en fonction du site d'allocation et nombre d'objets par site d'allocation.

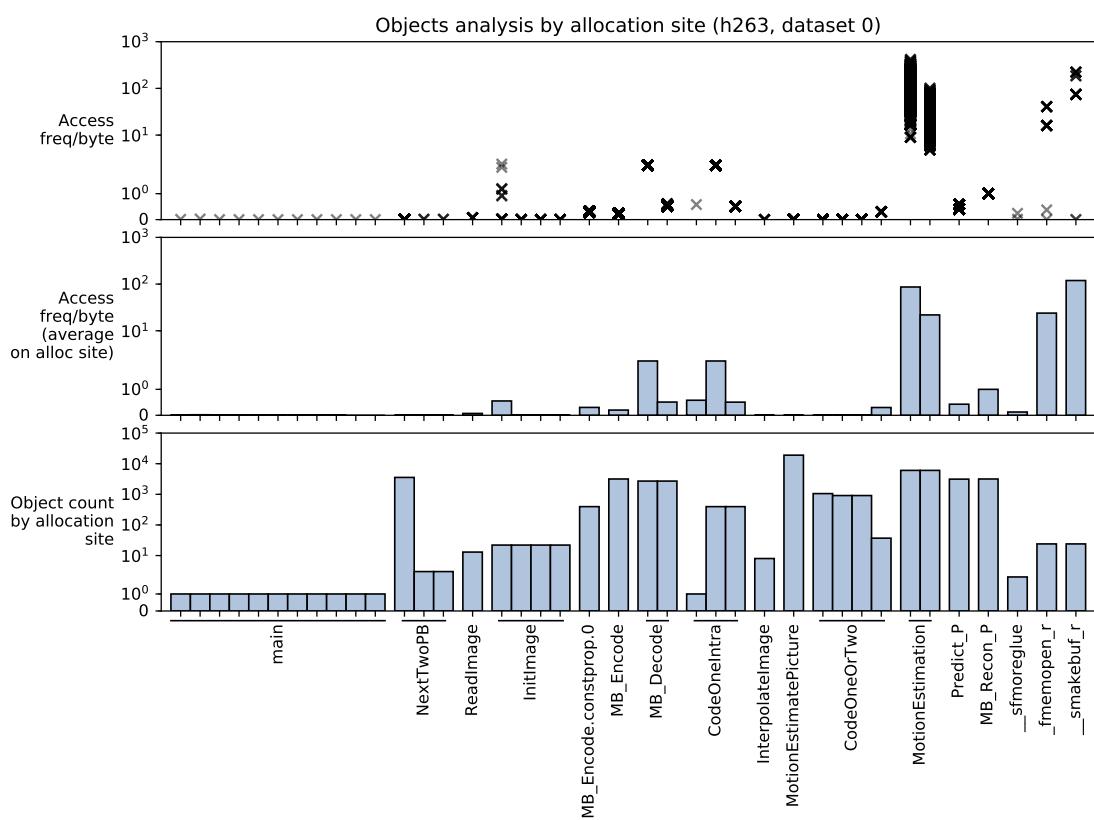


FIGURE 7.18 – **H263 : analyse des objets par site d'allocation** – exécution de référence
De haut en bas : fréquence d'accès par octet en fonction du site d'allocation, fréquence d'accès par octet moyen en fonction du site d'allocation et nombre d'objets par site d'allocation.

7.5 Conclusion

Nous avons présenté dans ce chapitre notre simulateur, ainsi que la manière dont nous avons implémenté l'allocation mémoire dynamique multi-tas. Nous avons également présenté les applications que nous avons retenues comme application cible de notre étude.

Dans le but d'élaborer des stratégies de placement nous avons mis en place la métrique de fréquence d'accès par octet. Cette métrique est basée sur le profil d'accès au tas (nombre d'accès en lecture et en écriture), sur le profil d'allocation (nombre, taille et ordre d'allocation des objets) ainsi que sur les technologies mémoires cibles (latences en lecture et écriture). Elle nous permet d'évaluer la chaleur d'un objet, et donc son importance pour la résolution du problème de placement.

Nous avons ensuite utilisé cette métrique pour évaluer à posteriori les traces de l'exécution des applications cibles. Nous avons cherché à déterminer quelles informations disponibles pendant l'exécution de l'application nous pourrions utiliser pour résoudre le problème de placement.

De cette étude nous concluons que le site d'allocation est une information qui peut être utilisée pour décider du tas de destination d'un objet. Néanmoins cette approche nécessite une phase de profilage.

Nous allons maintenant évaluer empiriquement l'impact sur les performances que peut avoir la stratégie de placement et évaluer les résultats d'une stratégie hors ligne guidée par la métrique de fréquence d'accès par octet.

Chapitre 8

Impact de la stratégie de placement sur les performances

Nous avons étudié et mis en avant la manière dont les applications allouent des objets et accèdent les objets alloués. Cette diversité dans le profil d'allocation et dans le profil d'accès au tas offre des opportunités pour améliorer les performances de l'application dans le cas où ces objets sont alloués dans un système mémoire hétérogène. L'approche que nous avons choisi pour tirer parti de ces opportunités est de résoudre le problème de placement dans une couche logicielle dédiée en amont de plusieurs tas : le dispatcheur. Chaque tas est géré par un algorithme de gestion de la mémoire dynamique classique adapté à ces conditions. Pour étudier l'intérêt de cette approche il est nécessaire d'estimer les gains atteignables en performances.

Nous allons donc mettre en œuvre plusieurs stratégies de placement de manière à mettre en lumière l'impact du problème de placement et de sa résolution. Ces différentes stratégies bien qu'elles ne soient pas toutes applicables telles quelles, nous permettent de caractériser pour les applications cibles non seulement les gains atteignables mais également quelles informations sont importantes pour dégager ces gains.

Ce chapitre contient donc les contributions suivantes : dans un premier temps nous proposons et évaluons une stratégie naïve tentant de tirer parti de l'hétérogénéité mémoire ainsi que son interaction avec le reste du système, notamment l'allocateur mémoire dynamique. Puis nous proposons une formulation mathématique du problème faisant abstraction de la fragmentation ainsi qu'une méthode de résolution par programmation linéaire en nombres entiers. Sur cette résolution nous construisons deux stratégies hors ligne bornant l'optimal pour évaluer les gains potentiels de notre approche. Dans un dernier temps nous évaluons la métrique de fréquence d'accès par octet, présentée au chapitre 7. Pour ce faire nous construisons une stratégie hors ligne basée sur cette métrique, dont l'impact sur les performances de l'application nous renseigne sur la qualité des décisions de placement qu'elle génère.

Au delà de la caractérisation de l'impact sur les performances d'une stratégie de placement, nous souhaitons aussi mettre en lumière l'importance de mettre en œuvre une stratégie de placement intelligente.

8.1 Stratégie baseline en ligne : Fast First

Le problème de placement est-il si difficile à résoudre ? En tant que programmeur, plusieurs idées viennent rapidement en tête pour tirer parti de l'hétérogénéité du système mémoire présenté dans notre cas d'étude.

Les considérations générales que l'on peut tenter d'appliquer en première approche sont les suivantes : si une partie de la mémoire est plus rapide, essayons de plus l'utiliser, de la garder remplie. De plus, introduisant une nouvelle couche logicielle, nous essayons de limiter le surcoût qu'elle induit. En se basant sur l'architecture logicielle présentée au 6.4, il est donc facile d'élaborer une stratégie guidée par ces considérations.

Cette stratégie, que nous évaluons comme une solution naïve au problème de placement essaye donc de remplir au plus vite le tas rapide.

8.1.1 Formulation

Cette Stratégie peut donc être formulée de la manière suivante :

DÉFINITION - Stratégie *Fast First* : pour toute requête d'allocation émise par l'application, essayer d'allouer dans le tas rapide. Si le tas rapide ne peut pas satisfaire la requête, allouer dans le tas lent.

La stratégie ainsi formulée est triviale à implémenter et permet de saturer et de garder saturé le tas situé en mémoire rapide. De plus le surcoût du à l'ajout du dispatcheur dans l'architecture logicielle reste faible. Néanmoins cette stratégie repose sur les adaptations de l'allocateur mémoire dynamique pour le multi-tas présentées au 7.1 comme nous allons le détailler ci-après.

8.1.2 Évaluation

Nous allons dans cette section évaluer les performances de cette stratégie sur les architectures mémoires présentées au 6.5.

Pour nos applications cibles, nous évaluons le gain de performance en pourcentage de gain par rapport à l'exécution de référence, c'est à dire l'exécution à un seul tas situé en mémoire lente et sans dispatcheur, les résultats sont présentés sur la figure 8.1.

Les résultats pour des faibles proportions de mémoire rapide sont mauvais. Par exemple, sur l'architecture 5% (présentée au chapitre 6), les applications dijkstra et json sont moins performantes que l'exécution de référence.

Néanmoins les applications arrivent à tirer parti de l'ajout de mémoire rapide à l'architecture et les performances augmentent donc pour se rapprocher des performances sur un tas composé exclusivement de mémoire rapide plus la proportion de mémoire rapide augmente dans l'architecture mémoire.

8.1.3 Discussion

La stratégie "Fast First" permet donc d'obtenir des gains de performances : par exemple pour 50% de mémoire rapide sur le scénario "*MRAM*", l'application jpg2000 voit son temps d'exécution réduire de 12,5%. Toutefois ces gains sont incertains pour les faibles proportion de mémoire rapide dans le tas. Ecdsa ne voit son temps d'exécution réduire que pour des architectures contenant plus de 25% de mémoire rapide sur le même scénario. Et pour l'autre scénario ("*ReRAM*"), même si la réduction du temps d'exécution est positive dès 5% de mémoire rapide il en faut plus de 25% pour que l'application s'exécute significativement plus vite. En effet même si l'implémentation de la stratégie est légère et que l'*early fail* permet de ne pas exécuter une recherche de bloc vide inutilement, certaines applications passent une part importante de leur temps dans l'allocateur mémoire dynamique et donc le fait d'essayer d'allouer systématiquement dans le tas rapide pénalise trop les performances.

De plus il est intéressant de remarquer que cette première stratégie n'exploite que l'hétérogénéité de l'architecture mémoire. Cependant, elle ne tient pas compte des différences entre les accès aux différentes objets. En d'autres termes une stratégie plus efficace, notamment pour une faible proportion de mémoire rapide, serait de mieux choisir les objets destinés au tas rapide.

Un dernier aspect important qui doit être pris en compte est l'interaction entre mécanismes de placement et mécanismes d'allocation mémoire. En effet, un allocateur mémoire dont le tas est saturé et ou très fragmenté présente des performances dégradées. Dans le cas de la stratégie "Fast First", la volonté de saturer en permanence le tas rapide en dégrade les performances d'allocation.

Impact des optimisations de l'allocation mémoire pour le multi-tas

La stratégie Fast First ne donne pas de bons résultats pour une faible proportion de mémoire rapide. Mais avec le mécanisme d'*early fail* elle arrive néanmoins à améliorer les performances dans la majorité des cas, comme présenté plus haut.

Néanmoins cette stratégie est très agressive vis à vis du gestionnaire mémoire dynamique du tas rapide. Nous pouvons donc estimer l'impact de ces optimisations sur l'efficacité de l'allocation mémoire dynamique.

La figure 8.2 permet de comparer les performances avec et sans les optimisations multi-tas présentées au 6.4. Nous présentons ici le temps passé dans le gestionnaire mémoire de l'exécution

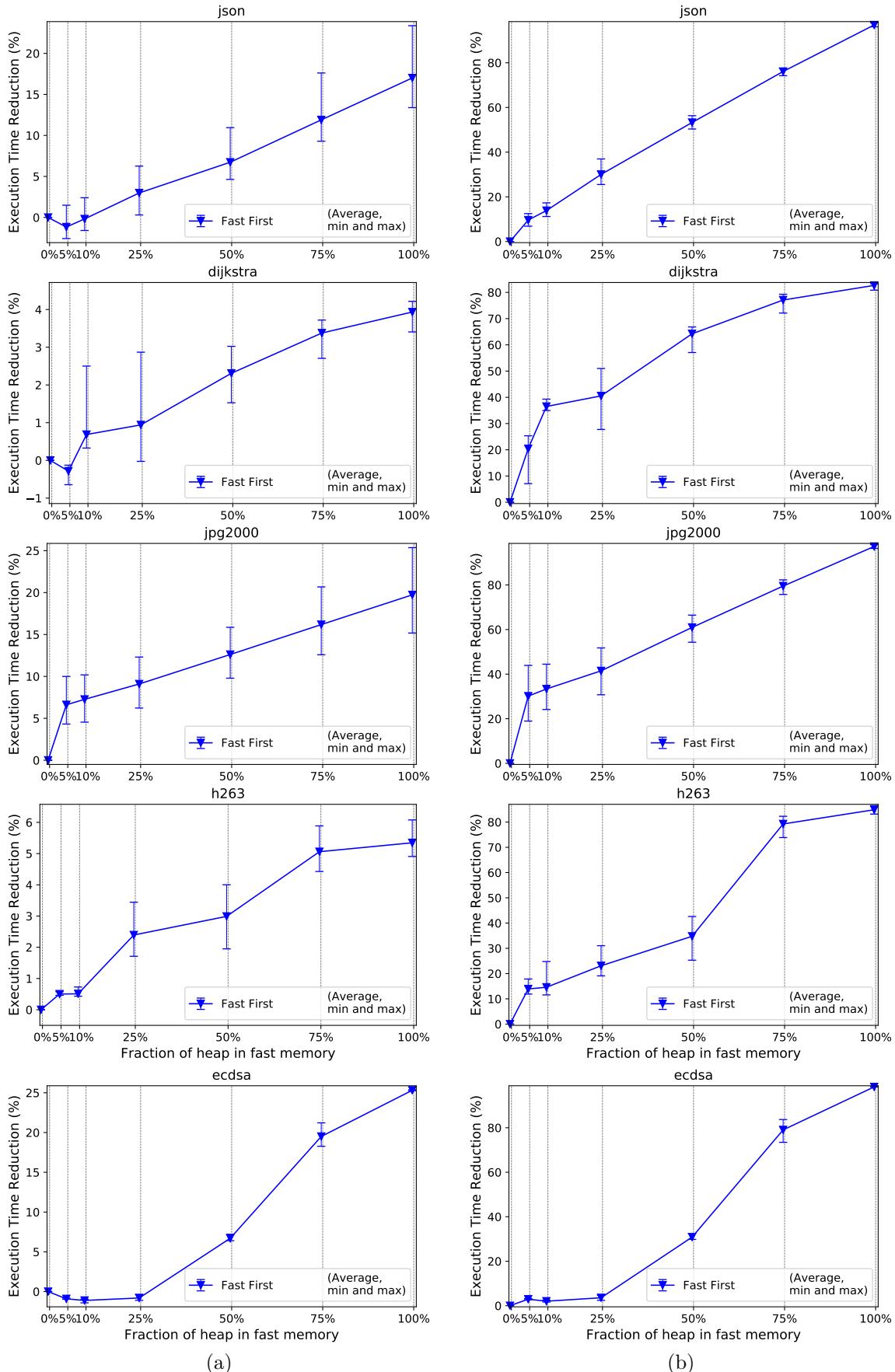


FIGURE 8.1 – Évaluation des performances de la stratégie **Fast First** en fonction de l'architecture mémoire : (a) scénario "*MRAM*" (b) scénario "*ReRAM*" – jeux de données 0 à 7

App	Without optimisation			With optimisation	
	Ref. Exec.	Time (cycles)	Time increase (%)	Tile (cycles)	Time increase (%)
Ecdsa	16535486	19741181	19,4	18618654	12,6
Dijkstra	12012570	15728469	30,9	14305431	19,1
Json	1307940	1659276	26,9	1533270	17,2
Jpg2000	9335023	11631876	24,6	11115146	19,1
H263	46565492	59478729	27,7	59155755	27,0

FIGURE 8.2 – Augmentation du temps passé dans le gestionnaire mémoire, avec et sans optimisation "early fail" pour les différentes applications.

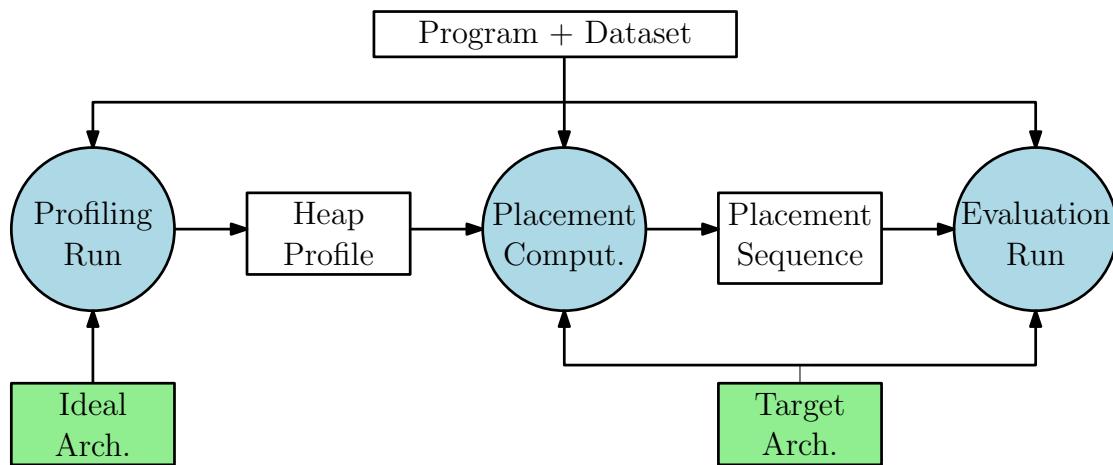


FIGURE 8.3 – Résolution du problème de placement hors ligne

de référence c'est à dire n'incluant aucun des mécanismes multi-tas. Nous comparons ensuite à ce temps le temps passé dans le gestionnaire mémoire de l'exécution de la stratégie "Fast First" avec et sans optimisation early fail. Ainsi le temps mesuré inclue l'algorithme de décision de placement ainsi que le temps passé dans l'allocateur mémoire. Les colonnes "time increase" (augmentation du temps passé dans le gestionnaire mémoire) sont calculées par rapport à l'exécution de référence. Toutes les exécutions présentées sont réalisées sur l'architecture contenant 5% de mémoire rapide. L'optimisation early fail limite donc le surcoût de l'allocation mémoire multi tas. Le surcoût sans optimisation est de 25,9% en moyenne et passe à 19% en moyenne avec l'optimisation.

8.2 Résolution hors ligne

8.2.1 Approche

Une résolution naïve du problème de placement donne de mauvais résultats si l'on ne prend pas en compte les interactions entre le problème de placement mémoire et l'allocation mémoire dynamique et si l'on ne tient pas compte de l'hétérogénéité dans le profil d'accès au tas.

Toutefois on pourrait espérer obtenir de bons résultats une fois ces aspects pris en compte par une stratégie de placement plus élaborée.

Pour répondre à cette interrogation et estimer les performances atteignables de notre approche nous proposons de calculer des solutions optimales hors ligne au problème de placement pour des exécutions particulières.

Ces solutions, rejouées à posteriori comme le montre la figure 8.3, permettent de mesurer précisément l'influence du placement mémoire des objets du tas sur les performances d'une application.

8.2.2 Notion de stratégie hors ligne

Pour ce faire nous allons donc réaliser une exécution sans dispatcheur. Cette exécution de référence nous permettra de récolter le profil d'allocation et le profil d'accès au tas de l'application pour un jeu de donnée particulier grâce à l'instrumentation de la plateforme de simulation.

Cette approche repose sur l'hypothèse que notre application, rejouée avec le même jeu de donnée générera non seulement le même profil d'allocation mais également le même profil d'accès au tas comme présenté au 6.3. Ainsi les objets, faisant la même taille, seront alloués et désalloués dans le même ordre, et de plus seront accédés de manière identique. En effet nous faisons l'hypothèse que les applications considérées ne tiennent pas compte des adresses où les objets sont alloués pour leur exécution.

On peut ainsi résoudre le problème hors ligne en se servant des informations de profil du tas. Par la suite, nous évaluons cette résolution hors ligne par l'exécution de l'application avec le même jeu de donnée en appliquant les choix de placement pré-calculés.

Cette méthode nous permet d'estimer les gains atteignable si nous sommes capables de calculer les choix de placement optimaux hors ligne. En effet l'application de ces choix n'est pas coûteuse en terme de temps processeur durant l'exécution d'évaluation, correspondant juste à aller chercher en mémoire pour chaque objet son tas de destination.

L'application d'une solution pré-calculée hors ligne peut donc être formulée ainsi :

DÉFINITION - Stratégies hors ligne : pour chaque requête d'allocation émise par l'application, allouer l'objet dans le tas décidé avant l'exécution.

Il est à noter que si une solution calculée hors-ligne n'est pas applicable elle sera adapté en ligne par le mécanisme de fallback présenté au chapitre 6.3.

8.2.3 Limitations

Ces solutions, et les performances qu'elles atteignent, ne sont néanmoins pas des solutions applicables ou envisageables en tant que tel pour répondre au problème de placement dans des systèmes embarqués réels. Même si la limitation méthodologique de la phase de profilage peut être acceptable dans le domaine de l'embarqué contraint, les solutions que nous calculons de cette manière sont des solutions particulières à une exécution de référence, et donc spécifiques pour le jeu de donné utilisé. Les stratégies hors ligne présentées par la suite dans ce chapitre ont donc pour but d'estimer les performances atteignables mais ne sont pas des solutions applicables.

8.3 Stratégie hors ligne optimale

Nous avons mis en œuvre une première stratégie de placement naïve n'exhibant pas de bons résultats. De manière à déterminer si la stratégie est en cause ou si l'approche mise en œuvre est inefficace, nous proposons ici une stratégie permettant de borner les performances du placement optimal et ce avec un surcoût faible à l'exécution. Toutefois cette stratégie n'est applicable qu'à une execution particulière. Nous allons donc formuler le problème de placement pour une exécution de référence et lui apporter une solution de manière à déterminer les performances atteignable avec les décisions de placement optimales de manière à évaluer les performances de la stratégie naïve.

8.3.1 Formulation Mathématique

Système mémoire

Nous modélisons le système mémoire décrit au chapitre 6. Les bancs mémoires dédiées au tas présentent des performances variées. Pour un banc mémoire m nous définissons les constantes suivantes :

- $cap(m)$: capacité mémoire en octets
- $lat_R(m)$: latence en lecture du banc mémoire (en cycles processeur)
- $lat_W(m)$: latence en écriture du banc mémoire (en cycles processeur)

Le sous système mémoire étudié M est donc un ensemble de bancs mémoires non idéales tel que :

$$M = \{m_0, m_1, \dots, m_n\}$$

Dans le cadre de cette thèse $|M| = 2$, et le tas est composé de mémoire rapide et de mémoire lente. Nous définissons donc

$$M = \{m_f, m_s\}$$

avec m_f (resp. m_s) le banc composée de mémoire rapide (resp. lente)

Objets du tas

La résolution du problème de placement nécessite la modélisation de ces deux aspects, permettant de caractériser les objets du tas.

Pour l'exécution d'une application avec un jeu de donnée on définit donc l'ensemble des objets alloués par l'application B on a donc $\forall b \in B$:

- $count_R(b)$: nombre d'accès en lecture à l'objet b durant l'exécution
- $count_W(b)$: nombre d'accès en écriture à l'objet b durant l'exécution
- $size(b)$: taille de l'objet b en octets

Bien que variables pour des jeux de donné d'entrée différents, ces valeurs sont des constantes du problème que nous formulons car nous le résolvons pour une exécution particulière.

Modélisation du temps pour l'allocateur mémoire

Comme vu au chapitre 2.4 l'allocateur mémoire n'a pas de représentation du temps, et se contente de satisfaire des requêtes dans l'ordre où elles arrivent. Nous définissons donc pour une exécution une séquence d'action *Actions* du gestionnaire mémoire dynamique (allocation ou désallocation) permettant de les ordonner.

Nous pouvons donc définir pour chaque objet $b \in B$:

- $alloc_index(b)$: index de l'allocation de b dans *Actions*
- $free_index(b)$: index de la désallocation de b dans *Actions*

Tel que $\forall b, b' \in B$:

$$\left. \begin{array}{l} alloc_index(b) = alloc_index(b') \\ free_index(b) = free_index(b') \end{array} \right\} \Leftrightarrow b = b'$$

Ainsi à tout instant de l'exécution de l'application on peut définir la dernière action de l'allocateur mémoire, donc $\forall (a, b) \mid a \in Actions, b \in B$

$$alive(b, a) = \begin{cases} 1 & \text{if } alloc_index(b) \leq a < free_index(b) \\ 0 & \text{otherwise} \end{cases}$$

Allocation mémoire dynamique

Pour résoudre hors ligne le problème de placement entre les tas il est nécessaire de prendre en compte la manière dont un allocateur mémoire dynamique place les objets dans le tas considéré. Néanmoins les détails de la solution au problème de fragmentation sont étroitement liés à l'implémentation de l'allocateur mémoire dynamique. Modéliser le comportement d'un allocateur tel que le DLMalloc se révèle trop complexe pour permettre notre étude.

Il est à noter que la définition de l'empreinte mémoire proposée au chapitre 2 concerne un seul tas. Pour l'étendre à plus nous considérerons que l'empreinte mémoire d'une application correspond à la somme de l'empreinte mémoire de chacun de ses tas.

Nous faisons également l'hypothèse que l'allocateur mémoire que nous utilisons garde la même empreinte mémoire pour un profil d'allocation donné tant qu'il dispose de suffisamment de mémoire pour allouer tout les objets.

Étant donné un profil d'allocation P , pour un allocateur mémoire donné, on définit $footprint(P)$ comme l'empreinte mémoire de l'application requise en octets par cet allocateur mémoire pour allouer tous les objets de P sur une architecture mémoire idéale.

Problème de placement

Le rôle du dispatcheur est de choisir dans quelle banc mémoire $m \in M$ allouer chaque objet $b \in B$ pour l'exécution d'une application. Étant donné que chaque requête d'allocation doit être satisfaite et allouée dans un seul tas, trouver une solution au problème de placement est équivalent

à trouver une fonction de B dans M . Une telle solution garantit que toutes les requêtes émises par l'application seront allouées dans un banc mémoire.

Toutefois, cette description du problème ne permet pas d'assurer que la solution au problème de placement soit valide.

Pour être valide pour une exécution sur le système mémoire $M = \{m_0, \dots, m_n\}$, et étant donné un allocateur mémoire, la fonction de B dans M doit respecter les contraintes suivantes :

$$\forall i \in [0; n] \text{ } footprint(\{b_i | alloc(b_i) = m_i\}) \leq cap(m_i)$$

Ainsi une solution respectant ces contraintes garanti que l'allocateur mémoire aura la place dans les différentes bancs mémoires pour allouer les objets qui y sont destinés. Il est important de noter que ces contraintes ne prennent pas en paramètre la taille des requêtes mais l'empreinte mémoire de l'allocateur mémoire pour allouer un sous ensemble des objets de l'application. En effet, l'allocateur mémoire doit encore résoudre le problème de fragmentation pour le sous ensemble de B destiné à chaque banc mémoire et la taille requise pour ce faire dépend non seulement de l'algorithme de l'allocateur mémoire et de ces choix d'implémentation, mais aussi du sous ensemble des objets visés (taille et ordre des requêtes).

Estimation de l'influence du placement sur le temps d'exécution

Nous voulons estimer le temps passé par le processeur durant l'exécution à attendre une lecture ou une écriture mémoire. Par définition, si le sous système mémoire est uniquement composé de mémoire idéale le processeur n'aura jamais besoin d'attendre une réponse de la mémoire. Nous pouvons donc calculer l'influence du système mémoire et des décisions de placement sur le temps d'exécution par rapport au temps d'exécution sur une architecture idéale.

Soit T_{ideal} le temps d'exécution de l'application cible avec un jeu de données particulier sur une architecture mémoire idéale $M_{ideal} = \{m\} \mid lat_R(m) = lat_W(m) = 1$.

Pour un placement mémoire $D = D_0, \dots, D_n$ des objets B d'une exécution, valide sur l'architecture mémoire $M = \{m_0, \dots, m_n\}$ On peut donc estimer le temps d'exécution :

$$T_{exec}(D, M) = T_{ideal} + \sum_{i=0}^n \sum_{b_j \in D_i} (count_R(b_j) \times lat_R(m_i) + count_W(b_j) \times lat_W(m_i))$$

Solution optimale

De cette formulation du problème, la solution optimale de placement consiste à trouver un placement mémoire $D_{opt} = D_0, \dots, D_n$ des objets B d'une exécution, valide sur l'architecture mémoire $M = \{m_0, \dots, m_n\}$ tel que :

$$\#D' \mid T_{exec}(D', M) < T_{exec}(D_{opt}, M)$$

8.3.2 Résolution par programmation linéaire en nombre entier

Nous avons formulé le problème mathématique de placement mémoire des objets du tas dans le cas général. Nous allons ici le formuler pour permettre sa résolution par un solveur ILP.

Nous proposons ci-dessous une formulation ILP du problème de placement réduit à deux tas et évalué en terme de performances en temps d'exécution. Nous soutenons néanmoins que cette démarche est applicable à des architectures mémoires hétérogènes plus complexes (plus de deux bancs) et sur de nombreux autres critères d'évaluation / métriques de performances (endurance, énergie). Ces possibilités d'élargissement de périmètre seront toutefois laissé hors de cette étude pour être abordées au chapitre 10.

Solvabilité du problème formulé : relaxation de la fragmentation

Les solutions au problème de placement que nous souhaitons calculer doivent toutefois satisfaire des contraintes de validité explicitées plus haut, à savoir que l'algorithme de gestion de la mémoire considéré doit être capable d'allouer dans le banc mémoire cible les objets qui y sont destinés.

Dans cette étude nous avons fait le choix d'utiliser un algorithme de gestion de la mémoire classique, généralement utilisé dans le domaine de l'embarqué pour des solutions industrielles, présenté au chapitre 2 : l'allocateur DLMalloc. Les détails d'implémentations d'un tel algorithme sont nombreux et modéliser l'empreinte mémoire de l'allocation d'un ensemble d'objets représente une complexité trop grande pour être prise en compte dans le processus de résolution de notre problème ILP.

Nous faisons donc le choix d'abstraire en partie le problème formulé pour pouvoir le résoudre en un temps acceptable.

Ainsi nous formulons les contraintes de validité des solutions en fonction de la somme de la taille des objets plutôt que de l'empreinte mémoire résultant de l'allocation de ces objets par un algorithme de gestion de la mémoire.

Il sera discuté dans la suite des implications d'une telle relaxation.

Variables

Étant donné l'architecture mémoire $M = \{m_f, m_s\}$ avec m_f (resp. m_s) le banc composé de mémoire rapide (resp. lente) et l'ensemble $B = \{b_0, \dots, b_p\}$ des objets d'une exécution de référence, on définit une partition de B , $D = \{D_f, D_s\}$ correspondant à la solution de placement que nous calculons.

On définit alors la fonction suivante :

$$mem(b_i) = \begin{cases} m_f & \iff b_i \in D_f \\ m_s & \iff b_i \in D_s \end{cases}$$

Nous pouvons alors définir les variables binaires du problème ILP :

$$x_i = \begin{cases} 1 & \iff mem(b_i) = m_f \\ 0 & \iff mem(b_i) = m_s \end{cases}$$

Fonction objectif

Nous définissons par la suite en fonction de ces variables du problème une fonction objectif à minimiser :

$$\begin{aligned} f_{obj} = & \sum_{i=0}^p x_i \times (count_R(b_i) \times lat_R(m_f) + count_W(b_i) \times lat_W(m_f)) + \\ & \sum_{i=0}^p (1 - x_i) \times (count_R(b_i) \times lat_R(m_s) + count_W(b_i) \times lat_W(m_s)) \end{aligned}$$

Cette fonction correspond à l'impact de la solution de placement sur le temps d'exécution d'une application, mais est indépendante du temps processeur dépendant de l'ISA et non modélisé ici.

Contraintes

Il est nécessaire de formuler des contraintes au problème ILP, sans quoi le solveur se contenterait de placer tous les objets en mémoire rapide. Les contraintes du problème relaxé sont donc qu'en négligeant la fragmentation interne et externe du tas rapide, la taille cumulée des objets placés dans ce tas ne peut excéder la capacité du banc mémoire cible. Toutefois les objets à considérer dans le calcul de ces contraintes varient au cours du temps. En effet la taille d'un objet occupant une part du tas rapide ne doit être prise en compte qu'entre son allocation et sa désallocation.

Il en découle un ensemble de contraintes devant être satisfaites par la solution au problème en première approche :

- chaque contrainte somme la taille des objets placés dans le tas rapide à un instant donné.
- il y a autant de contrainte que d'actions du système de gestion de la mémoire dynamique telles que décrite dans la modélisation du temps pour l'allocateur mémoire au début du 8.3

On formule ces contraintes sous la forme suivante, $\forall a \in Actions$:

$$\sum_{i=0}^p size(b_i) \times x_i \times alive(b_i, a) \leq size(m_f)$$

Sachant que les valeurs de $alive(b_i, a)$ pour tout b_i sont connues à la construction du problème. On peut toutefois facilement réduire le nombre de ces contraintes. En effet une désallocation ne pose pas de problème n'occupant pas plus d'espace dans le tas rapide. De plus si une série d'actions du système de gestion de la mémoire est constituée uniquement d'allocation, alors les contraintes successives englobent les précédentes. Nous pouvons donc formuler les contraintes sous la forme précédente uniquement pour les actions de l'allocateur mémoire étant une allocation précédent une désallocation, correspondant à un maximum local d'utilisation de mémoire dynamique. Il faut aussi prendre en compte la dernière action de l'allocateur si elle se trouve être une allocation.

Implications de la relaxation de la fragmentation

Pour que le problème formulé puissent être résolu dans un temps acceptable par un solveur ILP nous avons choisi d'abstraire le problème de fragmentation mémoire. Ignorer la fragmentation implique que la solution optimale calculée par le solveur sera optimiste sur l'espace mémoire utilisé du tas par les objets destinés au tas rapide. Ainsi la solution générée entraînera du fallback durant son exécution. Par là, nous entendons que le tas rapide sera saturé et que certains objets que la solution de l'ILP aurait placé en mémoire rapide n'y tiendront pas. En effet comme expliqué au chapitre 2 résoudre le problème de fragmentation est non trivial et l'allocateur mémoire dynamique ne peut pas utiliser 100% du tas de manière efficace.

Nous appliquons donc une méthode de résolution nous permettant de borner l'optimal par une sous-approximation et une sur-approximation de celui-ci.

Borner l'optimal

Comme nous l'avons expliqué ci-dessus la solution calculée par le solveur ILP entraînera du fallback. Toutefois, toute execution simulée sur l'architecture mémoire cible est une sous-approximation de l'optimal de la réduction du temps d'exécution.

Notre approche va donc être de trouver un moyen de limiter le fallback pour obtenir une meilleure sous approximation des performances optimales. Nous évaluons donc des solutions ILP calculées sur une taille de tas artificiellement réduite. Les meilleures performances des solutions évaluées de cette manière sont retenues comme sous-approximation des performances optimales. L'idée derrière cette approche étant de compenser la fragmentation ignorée dans le problème ILP.

De manière à borner l'optimal nous devons également calculer une borne supérieure pour la réduction du temps d'exécution. Nous choisissons d'utiliser la solution de placement calculée par l'ILP sans compensation de la fragmentation. Cette solution est optimale dans les conditions où elle est formulée. Mais son execution entraînant de la fragmentation, et donc le mécanisme de fallback, ses performances sont sous-optimales. Nous appliquons donc cette solution de manière exacte lors d'une execution. Pour ce faire, nous relaxons la taille du banc de mémoire rapide, permettant d'allouer dans le tas rapide tous les objets que le solveur ILP y a placé. Cette execution génère une empreinte mémoire pour le tas rapide supérieure à la taille de l'architecture cible. Elle n'est donc pas applicable à l'architecture cible. Toutefois, le temps d'exécution mesuré pour cette exécution est plus rapide ou égal à l'optimal. Cette exécution est donc une sur-approximation de l'optimal en réduction du temps d'exécution.

De cette manière et comme le montre l'évaluation ci dessous nous arrivons à resserrer les bornes autour de l'optimal d'une manière satisfaisante.

Nous formulons donc les deux stratégies suivantes :

DÉFINITION - stratégie ILP : Application d'une solution de placement ne tenant pas compte de la fragmentation calculée par ILP. Cette solution compense la fragmentation en solvant le problème sur une taille de tas rapide plus petite.

DÉFINITION - stratégie ILP Upper Bound : Application d'une solution de placement ne tenant pas compte de la fragmentation calculée par ILP. Cette solution ignore la fragmentation à l'exécution et présente donc des performances meilleures ou égales à celles du placement optimal.

8.3.3 Évaluation

Nous nous proposons donc d'évaluer les gains atteignables avec notre approche, ainsi que l'efficacité de la stratégie Fast First. Nous évaluons l'impact sur le temps d'exécutions des applications étudiées pour une architecture mémoire hétérogène dédiant deux bancs mémoire au tas, l'un rapide, l'autre lent.

Dans les systèmes de l'embarqué contraint à faible consommation, les technologies mémoires plus rapides sont plus coûteuses, en surface de silicium notamment, et donc on considère que le programme à, en règle générale, moins d'espace dans la mémoire rapide que dans la lente.

Nous allons donc évaluer nos différentes stratégies sur plusieurs architectures mémoires présentant des proportions de mémoire rapide différentes, mais les proportions de mémoire rapide intéressantes architecturalement pour le domaine se situent sous la barre des 50% de mémoire rapide.

La figure 8.4 présente les résultats de l'évaluation pour les différentes applications, sur les deux ensembles de technologies mémoire considérées (colonnes (a) et (b)). Chaque graphe présente en abscisse la proportion de mémoire rapide dans l'architecture mémoire dédiée au tas, en pourcentage de l'empreinte mémoire de l'application. Nous lisons en ordonnée l'accélération obtenue sur le temps d'exécution total de l'application comparé à l'exécution sur une architecture mémoire du tas composée uniquement de mémoire lente, avec un système de gestion de la mémoire dynamique non-adapté au multi-tas.

Les courbes de chaque graphe représentent donc les performances des différentes stratégies évaluées :

- Fast First : stratégie naïve définie au 8.1
- stratégie ILP : sous-approximation des performances de l'optimal, avec compensation de la fragmentation par résolution du problème sur une architecture mémoire sous-évaluée.
- stratégie ILP Upper Bound : sur-approximation des performances de l'optimal, ignorant la fragmentation à l'exécution.

Il faut donc voir ces deux dernières courbes comme bornant l'optimal de performance atteignable avec notre approche.

Estimation de l'optimal

Un des premiers résultats confirmés par la lecture de ces graphes est que notre stratégie visant à borner l'optimal donne des résultats satisfaisants. En effet, pour une partie des applications, sur la majorité des architectures, la différence de performances entre la stratégie ILP et la stratégie ILP Upper Bound est négligeable. Pour les applications où les deux courbes ne sont pas confondues la différence ne dépasse jamais 5% du temps d'exécution de l'application et est en majorité inférieure à 2%. Les plus gros écarts constatés, sur Dijkstra, avec une architecture composée de 50% de mémoire rapide par exemple restent en dessous de 20% de l'écart de performance entre un tas composé uniquement de mémoire lente et un tas composé uniquement de mémoire rapide.

Performances atteignables

L'intervalle des performances atteignables par rapport à l'exécution de référence correspond aux performances de la même application sur un seul tas composé de mémoire rapide. Cet intervalle dépend donc en premier lieu des différentes technologies mémoires choisies pour la mémoire rapide et lente – dans le cadre d'une évaluation basée sur le temps d'exécution. Nous présentons ces intervalles de performance par application pour chaque scénario technologique dans la figure 8.5.

Partant de ce point de vue, les performances de la stratégie de placement optimale que nous avons borné entre la stratégie ILP et la stratégie ILP Upper Bound nous renseignent sur la proportion de cet intervalle qui est atteignable en fonction de la proportion de mémoire rapide du tas.

Dans un premier temps, les résultats nous permettent d'espérer atteindre un niveau de performances proche des performances de l'exécution rapide) avec une architecture mémoire hétérogène pour le tas.

L'allure des courbes est spécifique à chaque application, mais ne présente que peu de variation en fonction des technologies mémoires utilisées.

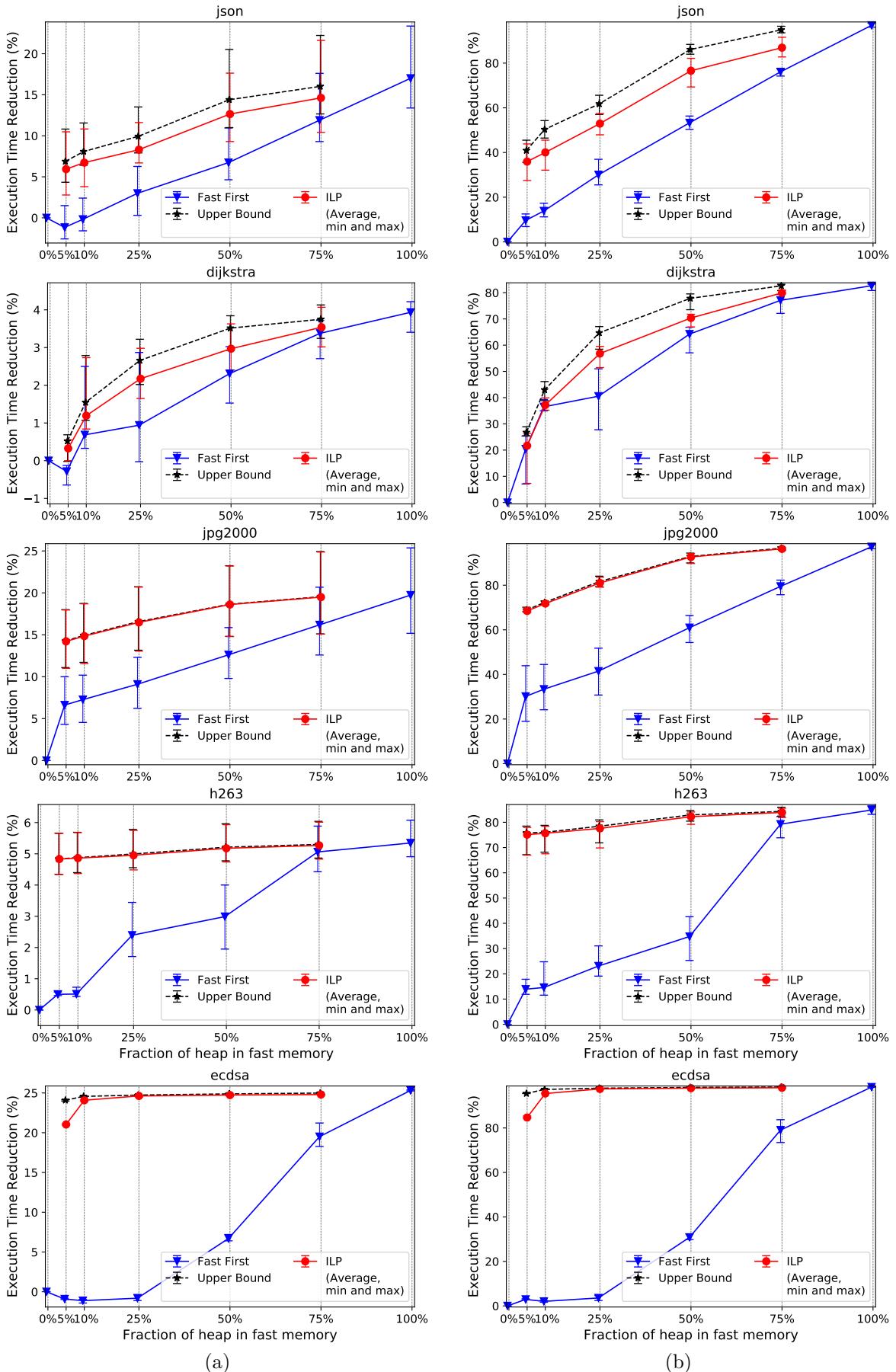


FIGURE 8.4 – Évaluation des gains optimaux contre la stratégie Fast First : (a) scénario "MRAM" (b) scénario "ReRAM" – jeux de données 0 à 7

	json	dijkstra	jpg2000	h263	ecdsa
Scen. " <i>MRAM</i> " (1/3 vs 2/30)	17%	4%	20%	5%	25%
Scen. " <i>ReRAM</i> " (1/1 vs 4/4000)	97%	83%	96%	83%	98%

FIGURE 8.5 – intervalle des performances atteignables par scénario technologique pour chaque application

Si l'on s'intéresse plus précisément aux architectures mémoires contenant peu de mémoire rapide qui sont les plus pertinentes pour le domaine de l'embarqué, on constate que dans tout nos benchmarks, 25% de mémoire rapide suffisent pour gagner entre la moitié et la quasi-totalité des gains potentiels.

Pour certaines applications comme H263 et Jpg2000 5% de mémoire rapide permettent d'atteindre 80% de l'amélioration de performance d'une exécution sur un tas composé uniquement de mémoire rapide.

Nous affirmons donc qu'une stratégie de placement efficace, si elle existe, permet de tirer profit efficacement de l'hétérogénéité mémoire pour les objets du tas.

Stratégie Fast First

Nous avons donc maintenant un objectif de performances atteignables auquel comparer la stratégie naïve Fast First. En s'intéressant à la courbe bleue dans les graphes de la figure 8.4 on peut faire plusieurs observations.

Tout d'abord, ajouter de la mémoire rapide ne permet pas toujours d'améliorer les performances avec une telle stratégie. Dans certains cas, placer 5%, voire 10% de l'espace mémoire du tas en mémoire rapide dégrade les performances de l'application. Ces dégradations de performances sont dues au fait que la stratégie essaie d'allouer beaucoup d'objets dans le tas rapide et malgré les optimisations réalisées pour adapter l'allocateur au multi tas que nous présenterons au début du chapitre suivant, le surcoût n'est pas nul. Mais surtout, au-delà de cette dégradation de performance, cette stratégie ne choisit pas quelles objets placer en mémoire rapide en fonction de leur importance dans le profil d'accès au tas de l'application, mais uniquement en fonction de l'ordre d'allocation des objets. Il est donc raisonnable de penser que le tas rapide se sature d'objets potentiellement peu intéressants si ceux-ci ne sont pas alloués au début de l'application. Ainsi, le surcoût de l'allocation mémoire multi-tas n'est pas compensé par des gains suffisant en terme de latence d'accès aux objets situés dans le tas rapide, aboutissant dans des gains en performances beaucoup moins importants pour les architectures mémoire possédant peu de mémoire rapide. Or ce sont ces architectures que nous considérons les plus représentatives du domaine de l'embarqué contraint.

Nous en concluons que pour exploiter avec succès l'hétérogénéité mémoire pour le placement des objets du tas une stratégie de placement intelligente est nécessaire. Nous entendons ici par intelligente qui prenne les décisions de placement en fonction de l'importance de l'objet dans le profil d'accès au tas de l'application.

Influence du jeu de donnée

L'étude des objets par taille et par site d'allocation que nous avons réalisée au chapitre 7 nous montre que pour différents jeux de données d'entrée, les profils d'allocation et d'accès au tas sont différents. Mais qu'en est-il du comportement des stratégies et des performances ? La figure 8.6 illustre les résultats de l'application jpg2000 pour deux jeux de données différents. Les courbes de performances que présentent la stratégie "Fast First" comme l'approximation de l'optimal sont quasi identiques, à quelques détails près pour les deux jeux de données. Mais au-delà de l'allure des courbes identiques, les performances atteignables changent en fonction du jeu de donnée d'entrée.

Au-delà de l'illustration que nous présentons ici ces résultats peuvent être trouvés pour toutes les applications en annexe. Les conclusions que nous tirons de l'analyse des variations de performances des stratégies en fonction du jeu de données d'entrée sont identiques. À savoir : le comportement des stratégies est identique et produit des courbes de performances à l'allure identique. Toutefois les différents jeux de données ne présentent pas le même degré d'hétérogénéité

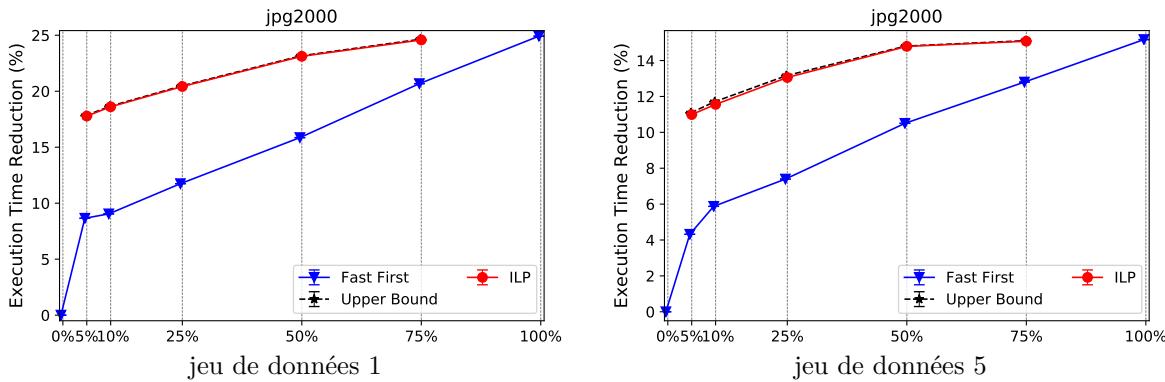


FIGURE 8.6 – Impact du jeu de donnée sur les performances, application Jpg2000

dans leur accès au tas. Ainsi une stratégie n'a pas de garantie d'atteindre toujours la même réduction du temps d'exécution de l'application. En effet, en fonction du jeu de données, les accès aux objets alloués varient faisant varier également les performances de la stratégie.

8.3.4 Discussion : impact sur l'exploration architecturale

Dans cette étude nous avons fait le choix de nous intéresser au domaine de l'embarqué, et plus spécifiquement de l'embarqué contraint avec les conséquences en terme d'architecture matérielles, en terme de logiciel et en terme de méthodes explicitées au chapitre 3.

Dans ce cadre il est intéressant de mettre en avant l'implication de nos conclusions dans une démarche de conception conjointe matériel-logiciel.

L'impact de la stratégie mis en lumière ci-dessus est intéressant du point de vue des performances de l'application. Mais du point de vue d'une adaptation du matériel (ou d'une IP matérielle) au logiciel cible, on peut constater que dans le cas de toutes les applications considérées sauf Dijkstra, la stratégie naïve Fast First, pour atteindre le même niveau de performance que l'optimal sur une architecture à 5% de mémoire rapide aura besoin d'au moins 50% de mémoire rapide, représentant un impact rendant une telle approche inapplicable dans ces conditions.

Il nous semble donc d'autant plus important d'avoir une stratégie de placement efficace dans le domaine que nous considérons.

8.4 Stratégie hors ligne "fréquence d'accès par octet" : évaluation de la métrique

Nous avons proposé et évalué la stratégie Fast First par rapport aux performances d'un placement optimal. Nous déduisons de ces résultats mitigés qu'une "bonne" stratégie de placement est nécessaire. De plus, une telle stratégie doit tenir compte de l'importance des objets dans le profil d'accès au tas de l'application, particulièrement si elle dispose d'une faible proportion de mémoire rapide pour le tas.

Au chapitre 7 nous avons élaboré la métrique de fréquence d'accès par octet pour déterminer l'importance des objets d'une exécution et former un critère de distinction de ceux-ci pour le placement multi-tas.

Nous allons donc utiliser cette métrique pour construire une solution de placement hors ligne. Cette étude nous permettra donc de juger si en suivant cette métrique il est possible de construire des solutions de placement efficaces.

En effet la stratégie ILP prend "les bonnes décisions" pour optimiser le temps d'exécution de l'application. Toutefois elle ne nous permet pas d'inférer sur quels critères mesurables il faut formuler la stratégie de placement pour prendre ces décisions ou s'en approcher.

8.4.1 Annotation des objets

Partant d'une exécution de référence nous allons donc évaluer chacun de ses objets à l'aide de sa fréquence d'accès par octet.

Rappel : fréquence d'accès par octet

La fréquence d'accès par octet telle que définie au 7.3 peut être calculée en fonction des technologies mémoires ciblées, du profil d'allocation et du profil d'accès au tas de l'exécution de référence.

$$freq_{octet}^{acces}(b) = \frac{count_R(b) \times (lat_R(m_s) - lat_R(m_f)) + count_W(b) \times (lat_W(m_s) - lat_W(m_f))}{size(b) \times overlap(b)}$$

Avec $overlap(b)$ le nombre d'objets alloués pendant la durée de vie de b (b inclus).

8.4.2 Stratégie

Construction de la solution

Selon ce critère nous pouvons donc classer les objets de l'exécution de référence du plus important au moins important.

Nous adoptons pour la construction de cette solution hors ligne la même simplification que pour la résolution du problème ILP à savoir ignorer la fragmentation. La différence est qu'au lieu de formuler un problème ILP et d'utiliser un solveur pour placer les objets, nous utilisons le classement des objets par fréquence d'accès par octet décroissante comme heuristique d'importance. Partant de ce classement nous construisons une solution de placement de manière gloutonne. Nous construisons donc la solution en plaçant les objets dans cet ordre – qui n'est pas l'ordre d'allocation et de désallocation des objets.

Pour respecter les contraintes de capacité des bancs mémoires de l'architecture cible, nous avons besoin de formuler des contraintes dont nous pouvons vérifier la satisfaction pendant la construction de la solution. De la même manière que pour la construction du problème ILP, et pour les mêmes raisons, nous ignorons ici la fragmentation.

Nous pouvons donc définir le tas de destination des objets dans l'ordre de notre heuristique. Nous vérifions hors ligne que pour toutes les actions de l'allocateur entre l'allocation et la désallocation de l'objet ajouté à la solution la somme de la taille des objets placés dans le tas rapide ne dépasse pas la capacité du banc mémoire. Si c'est le cas le tas de destination de l'objet considéré est défini comme le tas rapide dans la solution au moment de sa construction.

Si un objet ne peut être destiné au le tas rapide pendant ce parcours la solution définit son tas de destination comme étant le tas lent. Ainsi la solution construite hors ligne vise à avoir en mémoire rapide à chaque instant de l'exécution d'évaluation les objets ayant la plus haute fréquence d'accès par octet déjà alloués et pas encore désalloués par l'application.

Compensation de la fragmentation

Comme nous l'avons expliqué au 8.3 pour la stratégie ILP, ignorer la fragmentation produit une solution de placement moins performante.

Nous allons donc pour cette stratégie également compenser la relaxation de la fragmentation. Cette solution est donc calculée pour différentes tailles de tas rapide réduites. Ces solutions sont ensuite évaluées sur l'architecture mémoire cible et la meilleure est retenue.

8.4.3 Évaluation

Nous avons précédemment évalué la stratégie Fast First et les performances d'un placement optimal. Nous allons donc pouvoir évaluer la stratégie hors ligne basée sur la fréquence d'accès par octet par rapport à ces stratégies.

La figure 8.7 présente les mêmes courbes que précédemment, à savoir les résultats de la stratégie Fast First et de la sur et sous-approximation de l'optimal pour chaque application, et

ce sur deux choix de technologies mémoires (a) et (b). En addition la courbe verte présente les résultats de la stratégie hors ligne basée sur la métrique de fréquence d'accès par octet.

Le gain en performance sur le temps d'exécution se lit en ordonnée et le pourcentage du tas situé en mémoire rapide sur l'axe des abscisses.

Dans tout les cas où le placement optimal dégage des performances intéressantes la stratégie fréquence d'accès par octet fait de même. Les performances de cette stratégie, bien que légèrement inférieures à celles de la sous approximation de l'optimal suivent les mêmes tendances, notamment pour les architectures mémoires ayant une faible proportion de mémoire rapide.

Ainsi nous concluons que la stratégie basée sur la métrique de fréquence d'accès par octet prends les bonnes décisions, validant l'intérêt de la métrique pour la construction d'une stratégie qui serait cette fois en ligne

Conclusion

Dans ce chapitre nous avons évalué une stratégie en ligne, Fast First, et deux stratégies hors ligne, ILP et Fréquence d'accès par octet, pour répondre au problème de placement en mémoire hétérogène des objets du tas.

Ces stratégies nous permettent d'affirmer qu'il est possible de tirer parti de l'hétérogénéité du système mémoire pour améliorer les performances de l'application en plaçant les objets du tas entre mémoire rapide et lente.

Grâce à l'estimation de l'optimal de performance atteignable par cette approche, nous pouvons également affirmer que les gains en performance sont influencés par trois facteurs : le profil du tas, les technologies mémoires mises en jeu ainsi que la stratégie de placement.

Dans un premier temps différentes applications avec différents jeux de données présentent gains en performances atteignables très variés.

Nous avons également vu que les technologies mémoire déterminent aussi les gains possibles.

Mais surtout l'approximation de l'optimal des performances, comparé à l'évaluation de la stratégie Fast First met en lumière l'importance de la stratégie de placement en elle même pour l'influence sur les performances.

Les placements générés à partir de la métrique de fréquence d'accès par octet abordée au chapitre précédent étant une bonne approximation des placements optimaux nous allons maintenant pouvoir utiliser ces informations pour élaborer une stratégie en ligne efficaces.

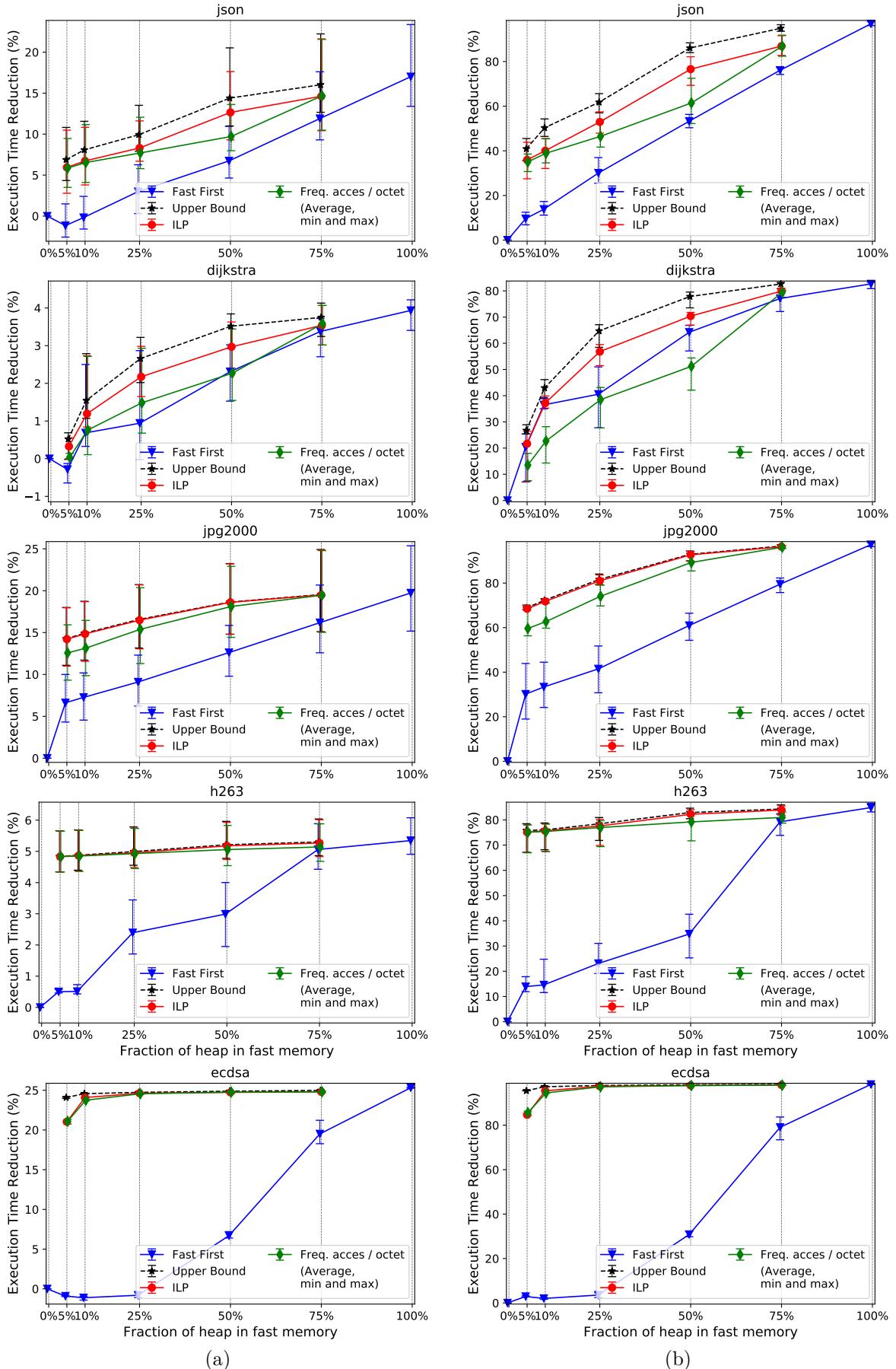


FIGURE 8.7 – Évaluation de la métrique "Fréquence d'accès par octet" : (a) scénario "*MRAM*" (b) scénario "*ReRAM*" – jeux de données 0 à 7

Chapitre 9

Conception et implémentation d'une stratégie efficace

Nous avons évalué au chapitre précédent l'impact que peut avoir la résolution du problème de placement sur les performances de l'application. Nous avons également montré que les gains en performances sont liés à la stratégie de placement retenue, à l'architecture mémoire (technologie et taille des bancs mémoires) en plus d'être liés au profil d'allocation et au profil d'accès de l'application.

Contrairement aux résultats précédents, où la résolution du problème hors ligne n'est applicable qu'à des exécutions équivalentes nous proposons ici une stratégie de résolution en ligne du problème.

Cependant nous considérons acceptable de proposer une résolution en ligne spécifique à une application et donc basée sur un profil préalablement calculé. En effet dans le domaine de l'embarqué, du à la méthodologie de développement et au besoin d'efficacité pour pallier au manque de ressources, le profilage est une solution généralement acceptée. Ainsi la résolution du problème que nous proposons doit fonctionner sur un jeu de données inconnu et donc des profils d'allocation et d'accès inconnus, pour une application donnée.

Nous avons également évalué au chapitre précédent la métrique de fréquence d'accès par octet construite au chapitre 7 qui permet de mesurer la chaleur des objets à posteriori. Or nous avons constaté que les objets, groupés par site d'allocation présentent des fréquences d'accès par octet proches. Nous présentons donc dans ce chapitre la méthodologie de construction d'un profil par site d'allocation et son utilisation en ligne pour la résolution du problème de placement pour l'exécution avec un jeu de donnée différent.

Cette approche permet de dégager des résultats proche de l'optimal pour une partie de nos applications cible, mais souffre de limitations dans le reste des cas. Nous présentons ensuite les différentes pistes que nous avons suivies pour améliorer cette stratégie. Nous évaluons une version prenant en compte le remplissage du tas pour adapter la longueur du préfixe du profil considéré. Nous évaluons également une stratégie dont le profil a été enrichi par les résultats de l'ILP.

9.1 Profilage des sites d'allocation d'une application

La stratégie que nous décrivons ici vise à être applicable à n'importe quelle exécution d'une application mais reste spécifique à une architecture mémoire. Pour ce faire nous nous basons sur l'observation que nous avons faite au chapitre 7. En effet, un site d'allocation qui alloue des objets chauds pour une exécution allouera en règle générale des objets chauds, même pour un jeu de données différent.

Ainsi, contrairement aux stratégies de résolution hors ligne présentées dans les chapitres précédents, l'évaluation sur le ou les jeu de données de profilage ne présente pas d'intérêt. Nous allons donc pour la construction de cette stratégie et son évaluation séparer les jeux de données à notre disposition en deux ensembles : ceux utilisés dans le profilage et ceux réservés à l'évaluation de la stratégie.

9.1.1 Exécutions non-équivalentes et couverture des sites d'allocation

L'exécution d'une application avec deux jeux de données différents produit des profils d'allocation et d'accès au tas différents. De plus, rien ne garantit que le chemin d'exécution sera identique entre les différentes exécutions. Or, dans notre définition des exécutions équivalentes nous utilisons le fait que pour un jeu de donnée d'entrée spécifique le flux d'exécution du programme est le même entre les deux exécutions considérées. Au contraire pour des jeux de données différents, aucune garantie de ce type n'existe. Ainsi rien ne peut garantir que l'ensemble des différents sites d'allocation qui vont être appelés durant l'exécution d'évaluation auront été considérés durant la phase de profilage.

Exemple : H263

Nous pouvons ici citer l'exemple de l'application h263. la figure 9.1 présente l'analyse de fréquence d'accès par octet de l'exécution de référence pour les jeux de données 0 et 1. Le nombre de sites est différent, en effet, l'exécution relative au jeu de données 1 utilise les fonctions `predict_B` et `MB_Recon_B` que l'exécution avec le jeu de donnée 0 n'utilise pas. Chacune de ces fonctions contient deux sites allouant environ 100 objets chacun. Or sur ces sites, le second de la fonction `predict_B` se classe quatrième dans le classement des sites allouant les objets les plus chauds. Ne pas l'inclure dans le profilage à donc de grandes chances de dégrader les performances si la fonction `predict_B` est appelée. Nous avons donc choisi de ne pas pénaliser un site qui ne serait pas présent dans toute les exécutions. Mais il est également important de concevoir une implémentation résiliente à l'appel d'un site d'allocation non couvert par le profilage.

9.2 Méthodologie

Notre plateforme de simulation est capable d'enregistrer, en plus des caractéristiques d'un objet son site d'allocation. La figure 9.2 décrit les étapes nécessaires pour cette stratégie. Nous nous servons des logs d'une exécution de référence pour obtenir la fréquence d'accès par octet de chaque objet de l'exécution. Nous calculons ensuite pour chaque site la valeur moyenne de la fréquence d'accès par octet des objets alloués. Nous pouvons ainsi ordonner les sites d'allocation par score pour obtenir une liste de site d'allocations. La liste est donc triée de celui allouant en moyenne les objets les plus chauds à celui allouant en moyenne les objets les plus froids. A ce stade à chaque site d'allocation correspond une fréquence d'accès par octet moyenne des objets alloués. Il est intéressant de noter que ce calcul est spécifique aux technologies mémoires mises en œuvre. En effet, le calcul de la fréquence d'accès par octet (formule rappelée ci-dessous) dépend des latences des différentes technologies mémoires :

$$freq_{octet}^{acces}(b) = \frac{count_R(b) \times (lat_R(m_s) - lat_R(m_f)) + count_W(b) \times (lat_W(m_s) - lat_W(m_f))}{size(b) \times overlap(b)}$$

Pour tout $b \in B$, l'ensemble d'objets d'une exécution de référence, et avec pour cible les technologies mémoires $M = \{m_f, m_s\}$ avec m_f (resp. m_s) le banc composée de mémoire rapide (resp. lente) :

- $count_R(b)$: nombre d'accès en lecture à l'objet b durant l'exécution
- $count_W(b)$: nombre d'accès en écriture à l'objet b durant l'exécution
- $size(b)$: taille de l'objet b en octets
- $lat_R(m)$: latence en lecture de la mémoire m
- $lat_W(m)$: latence en écriture de la mémoire m

Mais à ce stade la liste de site d'allocation est spécifique à une exécution. Il serait possible de déduire une stratégie sur la base d'une seule exécution de référence. Toutefois, il est dans ce cas possible que les comportements capturés par la stratégie soient des comportements spécifiques à l'exécution particulière plutôt qu'à l'application.

De manière à éviter cette limitation nous profilons plusieurs exécutions de référence sur des jeux de données différents. Nous agrégeons ensuite les listes de sites d'allocation obtenues. Cette agrégation est réalisée par site d'allocation, sur les différents scores des différentes exécutions.

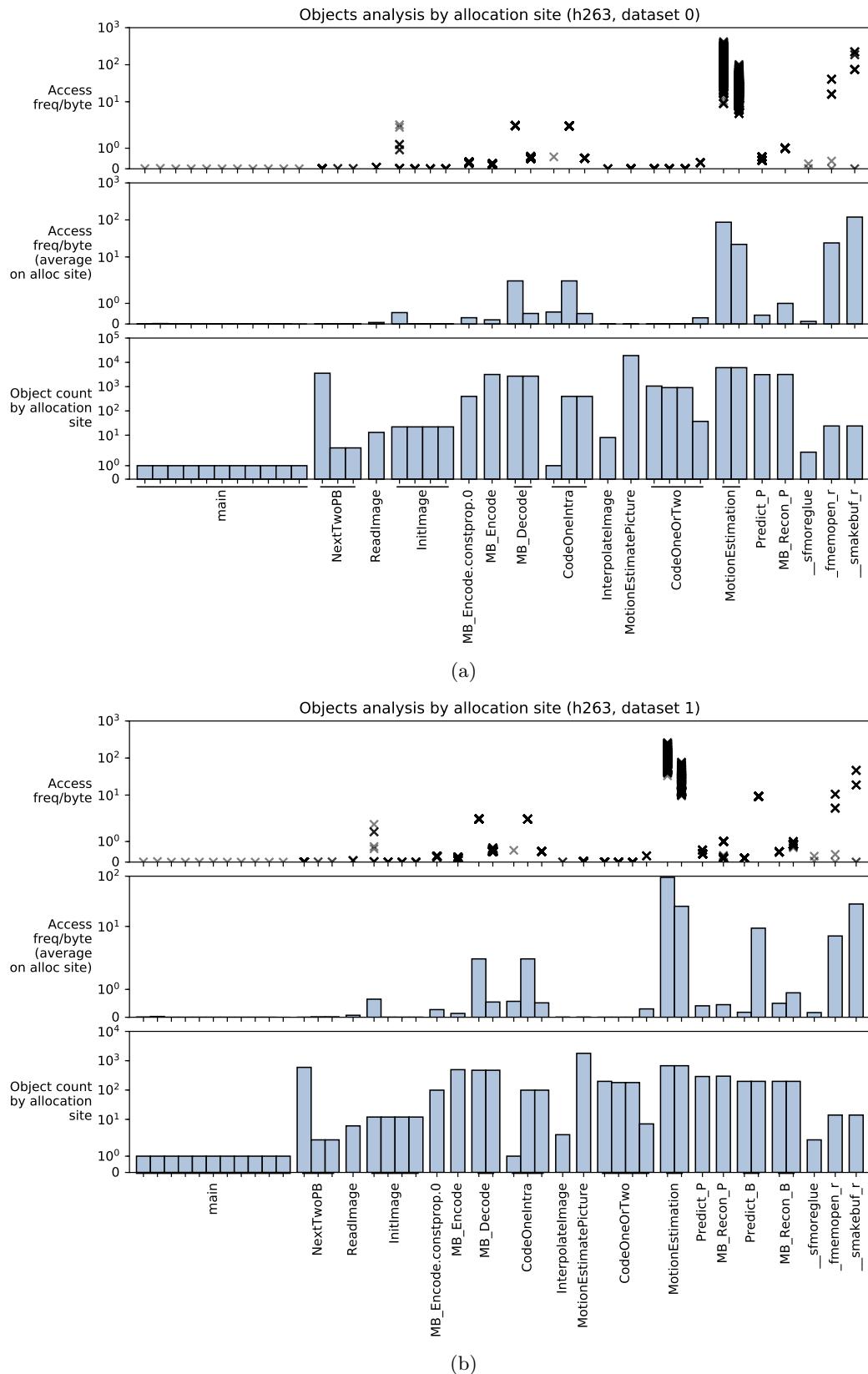


FIGURE 9.1 – H263 : variation de la couverture des sites d’allocations
Exécutions de référence, jeux de données : (a) d0, (b) d1

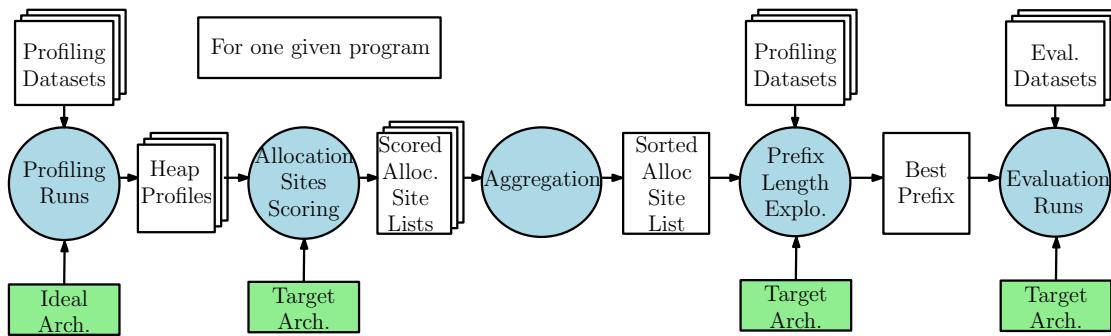


FIGURE 9.2 – Workflow de la stratégie site d’allocation

Nous agrégeons les scores des sites d’allocation en sélectionnant le neuvième décile des scores. Toutefois les listes agrégées ne présentent que des différences à la marge en fonction des opérations d’agrégation que nous avons testées (maximum, moyenne et neuvième décile).

La liste agrégée ainsi obtenue est utilisée pour générer un header, qui intégré dans le code embarqué de l’application permet d’appliquer la stratégie en ligne. Cette stratégie consiste à regarder si l’adresse de retour de la fonction appelant la primitive d’allocation est dans cette liste. Cette liste est néanmoins tronquée : si elle était gardée en entier, cette stratégie serait équivalente à la stratégie de référence Fast First, avec un surcoût supplémentaire. Si l’adresse de retour est dans le préfixe sélectionné de la liste, le dispatcheur transmet la requête à l’allocateur mémoire du tas rapide et dans le cas contraire à celui du tas lent. Dans le cas où un tas n’est pas à même de satisfaire une requête, un mécanisme de fallback la redirige vers l’autre.

Nous allons maintenant discuter plus en détail de la séparation des jeux de données entre profilage et évaluation ainsi que de la sélection des sites d’allocations intéressants.

9.2.1 Séparation des jeux de données

Comme nous l’avons dit précédemment nous avons séparé nos jeux de données en deux ensembles, un pour le profilage et l’autre pour son évaluation. Mais n’ayant que huit jeux de données par application, il est possible que le découpage en lui même présente une influence sur les résultats. Nous avons donc étudié l’influence de ce découpage sur les résultats de la stratégie pour l’application jpg2000.

Nous avons réalisé la liste des sites classés par fréquence d’accès moyenne suivant plusieurs découpage des jeux de données entre évaluation et profilage. Les listes obtenues présentent quelques permutations deux à deux de sites d’allocations mais aucune qui aie un impact. En effet comme nous le décrivons ci dessous nous explorons le préfixe de cette liste à prendre en compte et ainsi les permutations à l’intérieur de ce préfixe ou au delà n’ont pas d’importance.

9.2.2 Sélection des sites d’allocation

Nous avons construit une liste des sites d’allocation d’une application. Ces sites d’allocation sont ordonnés de celui allouant les objets les plus chauds à celui allouant les objets les plus froids. Mais il faut encore décider quel préfixe prendre en compte, c’est à dire quel préfixe de la liste sera effectivement alloué en mémoire rapide à l’exécution.

De manière à répondre à cette question nous avons réalisé l’exploration de la longueur de préfixe par application. Pour ce faire nous devons réaliser une exécution par longueur de préfixe considéré. Toutefois, cette valeur est également relative à une architecture mémoire du tas. En effet si le tas rapide n’a pas la même taille, le préfixe donnant les meilleures performances sera différent.

Nous commençons donc notre évaluation par l’étude de l’influence de la longueur du préfixe sur les performances, par application et par architecture

9.3 Évaluation

Nous avons donc appliqué cette méthodologie aux applications de notre étude. Dans un premier temps nous présentons l'exploration de la longueur de préfixe optimale par application. Il est à noter que l'appellation longueur "optimale" de préfixe est utilisée ici pour désigner la longueur de préfixe donnant les meilleurs résultats en terme d'amélioration de performances. Nous discutons également de l'influence de différents paramètres cette exploration : technologies mémoires, architecture mémoire et jeu de données.

Une fois la longueur optimale de préfixe déterminée nous pouvons utiliser ce préfixe et comparer les performances de la stratégie de site d'allocation aux stratégies hors ligne évaluées précédemment. Ces résultats varient entre des améliorations de performances proche de l'optimal à des améliorations peu significative suivant l'application.

9.3.1 Exploration de la longueur de préfixe

Nous devons donc choisir la longueur du préfixe de la liste de site d'allocation qui seront effectivement alloué en mémoire rapide. Or la quantité de mémoire rapide est déterminée par l'architecture mémoire. Dans notre cas, cette architecture est déterminée par un pourcentage de mémoire rapide, paramétré par l'empreinte mémoire de l'application.

La figure 9.4 présente l'exploration de la longueur du préfixe pris en compte durant l'exécution. Les performances de l'application appliquant la stratégie de "site d'allocation" sont présentées sur l'axe des ordonnées, et l'axe des abscisses correspond à la longueur de préfixe qui a été prise en compte. Ainsi chaque point des courbes présentées dans les figures 9.4, 9.5 et 9.8 correspondent l'exécution d'une application. Durant cette exécution, le dispatcheur applique la stratégie de site d'allocation. Ainsi si le site émettant la requête est dans le préfixe considéré le dispatcheur l'alloue dans le tas rapide, dans le tas lent sinon. Les différentes exécutions d'une application utilisent la même liste de sites, mais la longueur du préfixe pris en compte varie. Cette méthodologie d'exploration est lourde à mettre en place, impliquant plus de 1000 simulations pour l'étude de l'application jpeg2000 dans nos conditions par exemple. Nous avons donc fait varier la granularité d'exploration pour limiter le temps de simulation, permettant de réduire ce nombre de moitié. Nous avons exploré plus finement la longueur de la liste pour les premiers sites d'allocation. Pour les longueurs de listes au delà d'une vingtaine de sites d'allocation, nous utilisons un δ par application, conditionné par le nombre de site à explorer. Ainsi nous utiliserons dans ce chapitre les notations suivantes :

- L : Longueur du préfixe pris en compte
- L_{opt} : Longueur de préfixe donnant le meilleur résultat
- δ : incrément dans l'exploration de la longueur de préfixe

En première approche, nous notons que toutes les courbes présentent un maximum de performances suivi d'une baisse de performances si le préfixe grandit. Les performances s'améliorent en augmentant la longueur du préfixe jusqu'à atteindre leur maximum pour $L = L_{opt}$. Ce comportement est provoqué par l'allocation de plus d'objets chauds dans le tas rapide quand la longueur du préfixe augmente.

Nous observons une baisse de performances pour les longueurs de préfixe dépassant l'optimal ($L > L_{opt}$) Cette baisse de performance est provoquée par le mécanisme de fallback. Ce mécanisme a un impact négatif sur les performances, même si comme nous l'avons montré au chapitre 8 notre implémentation limite cet impact négatif. Mais ce mécanisme a un autre impact, plus visible sur les performances de l'application. En effet, il ne choisit pas quel objet est redirigé vers la mémoire lente dans le cas où le dispatcheur essaye d'allouer dans un tas saturé. Ainsi les objets redirigés vers le tas lent ont de grande chance d'être plus chauds que les objets des sites pris en compte en fin de préfixe. Ainsi, l'ajout d'un ou plusieurs sites dans le préfixe entraînant le déclenchement d'un fallback important dégrade significativement les performances.

Nous avons observé que pour chaque application que nous avons étudié il existe une longueur optimale pour le préfixe. Elle correspond au maximum de sites d'allocation que le dispatcheur peut allouer en mémoire rapide sans provoquer de fallback important, pris en compte dans l'ordre calculé précédemment.

App.	L_{opt}	Longueurs de préfixe considérées		
		$L = L_{opt} - \delta$ (% fb)	$L = L_{opt}$ (% fb)	$L = L_{opt} + \delta$ (% fb)
H263	29	4,68 (0,05)	4,69 (0,08)	0,63 (85,1)
Jpg2000	7	10,99 (0)	10,99 (0)	7,73 (6,63)
Ecdsa	1	Ø	0,21 (88,6)	-0,48 (88,7)
Json	5	-2,21 (0,17)	3,58 (0,15)	-0,12 (17,41)
Dijkstra	2	-0,67 (0)	0,16 (0)	-0,61 (94,8)

FIGURE 9.3 – Performances autour de la longueur optimale de préfixe (L_{opt})

Pour chaque application, la première colonne indique la longueur de préfixe optimale. Les trois dernières colonnes présentent le pourcentage de réduction du temps d'exécution et le pourcentage d'objets alloués par le mécanisme de fallback (%fb). δ correspondant à la granularité d'exploration de longueur de préfixe (entre 1 et 4).

Moyenne sur les jeux de données d'évaluation, architecture 5% de mémoire rapide, scénario technologique "MRAM".

Il est également intéressant de noter que les différences de comportement en fonction du jeu de donnée sont faibles. Ainsi notre stratégie est résiliente au dynamisme dépendant du jeu de données.

Le tableau 9.3 illustre ce point. En observant les gains en performances de la stratégie pour les valeurs de L autour de L_{opt} nous pouvons affirmer que les performances sont dégradées par une stratégie tentant d'allouer trop dans le tas rapide.

Le fallback est donc un mécanisme intéressant pour éviter l'échec de l'allocation mais il faut le limiter au maximum.

Pour mieux visualiser l'intérêt de ce résultat nous proposons les figures 9.6 et 9.7 contenant le tracé de l'occupation du tas pour la longueur de préfixe optimale ($L = L_{opt}$) et le point d'exploration suivant ($L = L_{opt} + \delta$) de l'exploration de préfixe des applications H263 et Json.

Sur ces graphiques, l'axe des abscisses représente le temps d'exécution en cycle processeur et l'ordonnée représente les adresses du tas. À droite de ces graphiques l'architecture mémoire considérée est représentée (nom et latences d'accès pour chaque tas). Chaque rectangle correspond à un objet, la longueur correspondant à sa durée de vie et sa hauteur à sa taille en mémoire. La fréquence d'accès par octet est représentée par l'intensité de la couleur de chaque rectangle : peu saturé la fréquence d'accès de l'objet est faible, au contraire une couleur vive indique un objet très accédé. Dans le cas où la couleur de remplissage du rectangle est bleue, l'allocation s'est déroulée dans le tas choisi par le dispatcheur, si l'objet est rouge, il a été alloué par le mécanisme de fallback.

Pour le cas de H263 (figure 9.6) on constate que le tas rapide est largement sous rempli pour une longueur de préfixe optimale ($L_{opt} = 29$). Il est toutefois complètement plein pour une longueur de préfixe supérieure ($L_{opt} + \delta = 32$). De plus, beaucoup plus de requêtes font l'objet d'un fallback vers le tas lent dans le second cas. Cet exemple illustre qu'un tas sous-rempli n'est pas forcément incompatible avec de bons résultats. En effet dans ce cas les profils d'allocation et d'accès au tas présentent suffisamment d'hétérogénéité pour que le fait de mettre une très faible partie des objets en tas rapide permette de gagner la majorité des gains atteignables.

La figure 9.7 présente la même analyse du tas pour l'application Json, pour des longueurs de préfixe de 5 et 6. Nous y observons le même phénomène, c'est à dire que pour une longueur de préfixe de L_{opt} (5 sites), le tas rapide est très loin d'être rempli mais les performances de l'application sont meilleures (réduction du temps d'exécution de 3,58%) que pour une longueur de 6 (-0,12%). De la même manière l'ajout d'un site d'allocation permet de maintenir le tas rapide saturé mais en contrepartie certains des objets des 5 premiers sites d'allocation de la liste se retrouvent alloués en mémoire lente, diminuant significativement les performances de l'application. Dépendant de l'application la saturation du tas rapide peut aussi avoir un impact visible sur les performances. Dans le cas de l'application json, le remplissage plus important du tas provoque un ralentissement de l'allocateur. En effet le temps d'exécution total augmente de 6.00×10^6 à 6.42×10^6 cycles, et cette augmentation est due pour 12% au temps passé dans l'allocateur (5.4×10^5 à 6.0×10^5 cycles). Au contraire, l'application H263 présente des calculs plus intenses et chaque objet est donc plus accédé. Ainsi la dégradation des performances de l'allocateur reste

négligeable sur le temps d'exécution total (moins de 1% de la variation du temps d'exécution est du à l'allocateur).

Influence de l'architecture mémoire

La figure 9.5 présente l'exploration de la longueur de préfixe pour l'application h263 pour les architectures à 5%, 25% et 75% de mémoire rapide. Cette application a la particularité de présenter quelques sites extrêmement contributifs qui peuvent tous tenir en mémoire rapide avec une architecture n'en contenant que 5%. Nous entendons par là que l'allocation de ces sites en mémoire rapide dégage la majorité de l'amélioration des performances possibles pour l'application. Ainsi la majorité des accès mémoires de l'application vers le tas concernent les objets alloués par ces sites. Toutefois, nous pouvons noter qu'une longueur de préfixe au delà du trentième site dégrade très fortement les performances pour l'architecture mémoire 5%. Une architecture composée de 25% de mémoire rapide est moins impactée par cette augmentation de la longueur de profil et sur l'architecture composée de 75% la baisse de performance n'est pas significative. Pour cet exemple, ajouter de la mémoire rapide n'améliore pas les performances significativement, mais réduit l'impact d'un profil trop long. En effet l'augmentation de la taille de profil ajoute des objets dont la fréquence d'accès n'apporte que peu d'amélioration de performances, mais comme la quantité de mémoire rapide est limitée, ces objets prennent la place d'objets beaucoup plus chauds, dégradant les performances.

Influence des technologies mémoire

Nous avons mené cette étude sur les deux scénarios technologiques présentés au chapitre 3 : un scénario basé sur deux implantations de MRAM et un scénario mettant côte à côte une technologie ReRAM et une technologie SRAM. bien que ces scénarios présentent des différences fortes en terme de latences un premier constat est que les courbes d'exploration de longueur de préfixe sont semblables. Ainsi, la longueur optimale de préfixe est la même pour les deux scénarios. De plus, les observations faites précédemment sur le scénario "*MRAM*" sont visibles sur les courbes impliquant le scénario "*ReRAM*". Nous concluons que notre approche est valide pour ces deux scénarios technologiques. Ainsi, nous déduisons que les comportements observés permettant l'utilisation d'une telle stratégie sont peu influencés par les technologies mémoires mises en oeuvre. Celle ci influencent par contre l'intervalle d'amélioration possible des performances comme présenté au chapitre 8.

Conclusion

L'ensemble de ces expériences nous permet de proposer et d'évaluer la stratégie pour la longueur de préfixe optimale par application et par architecture. Nous pouvons de plus tirer plusieurs conclusions de cette exploration. Tout d'abord il est possible de trouver un préfixe améliorant les performances de l'application. De plus, comme nous l'avons vu avec l'application h263, une application peut avoir besoin de plus ou moins de mémoire rapide pour dégager des résultats substantiels. Une fois cette quantité de mémoire atteinte, augmenter la quantité de mémoire rapide n'améliore pas significativement les performances.

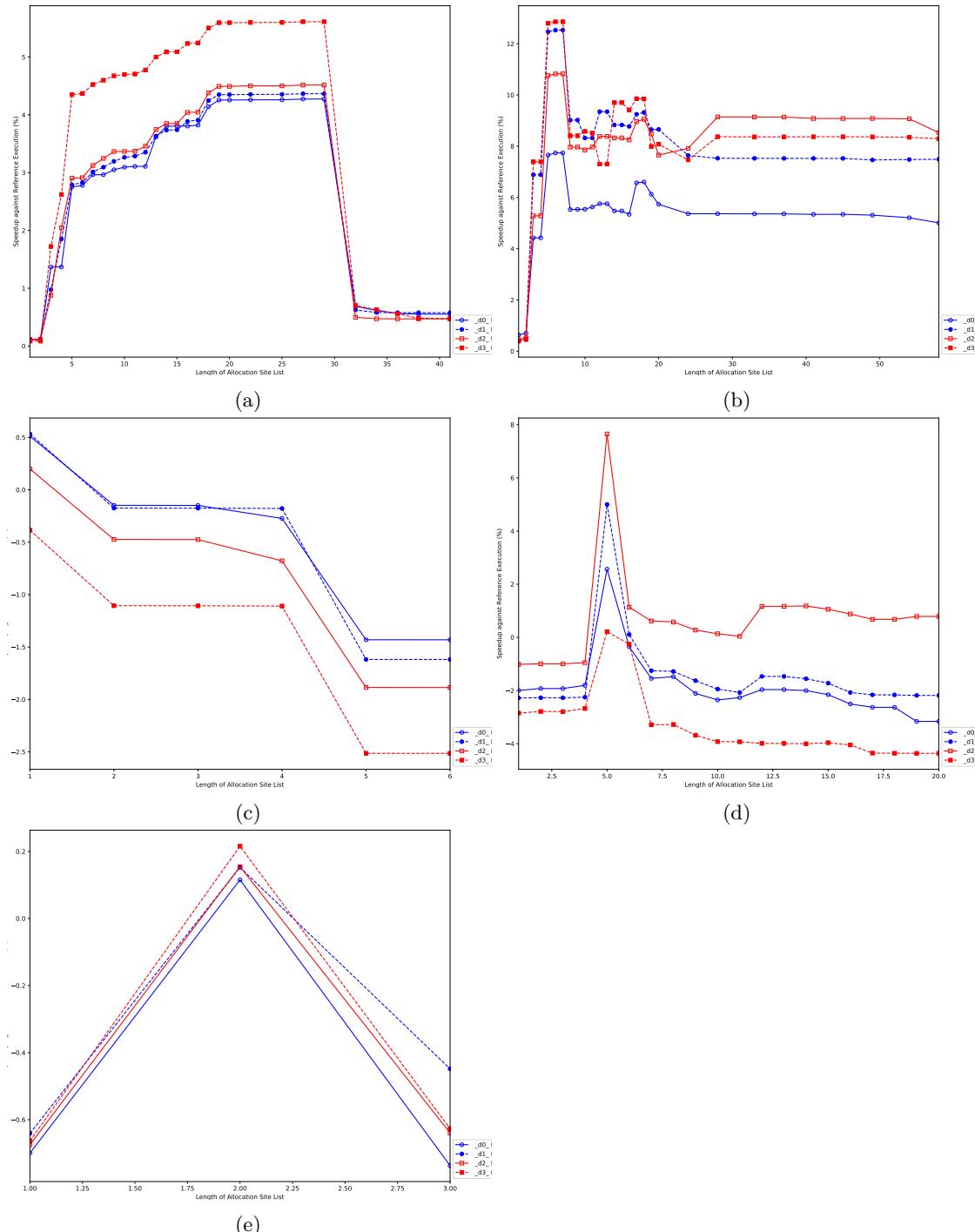


FIGURE 9.4 – Exploration de la longueur du préfixe – **Comportement des applications cibles.**

Applications : (a) H263, (b) jpg 2000, (c) EcDSA, (d) Json Parser et (e) Dijkstra.
Architecture 5% de mémoire rapide, jeux de données de 0 à 3.

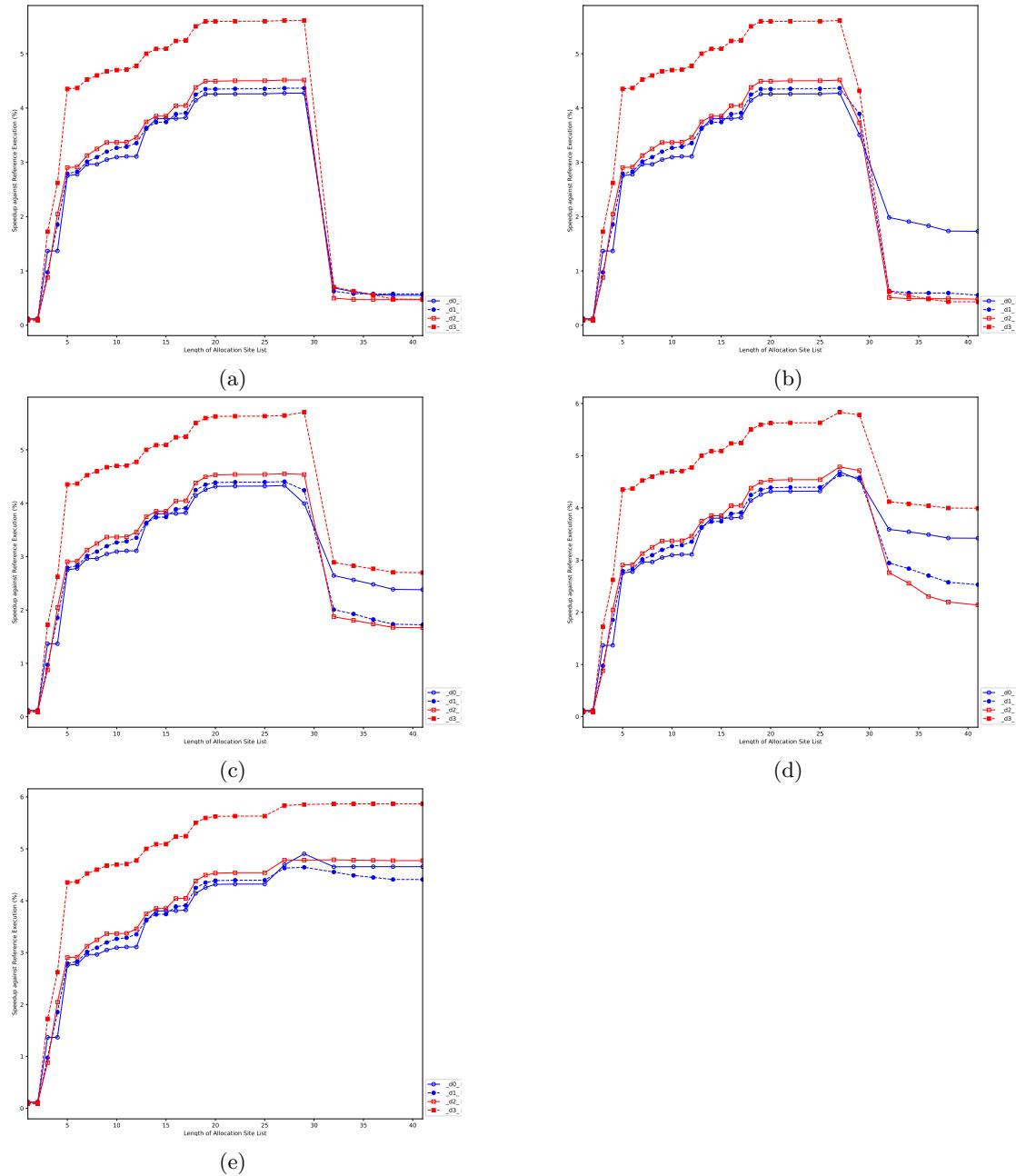


FIGURE 9.5 – Exploration de la longueur du préfixe – **influence de l'architecture cible.**
 Architectures : (a) 5%, (b) 10%, (c) 25%, (d) 50% et (e) 75% (% de mémoire du tas rapide)
 Application H263, jeux de données de 0 à 3.

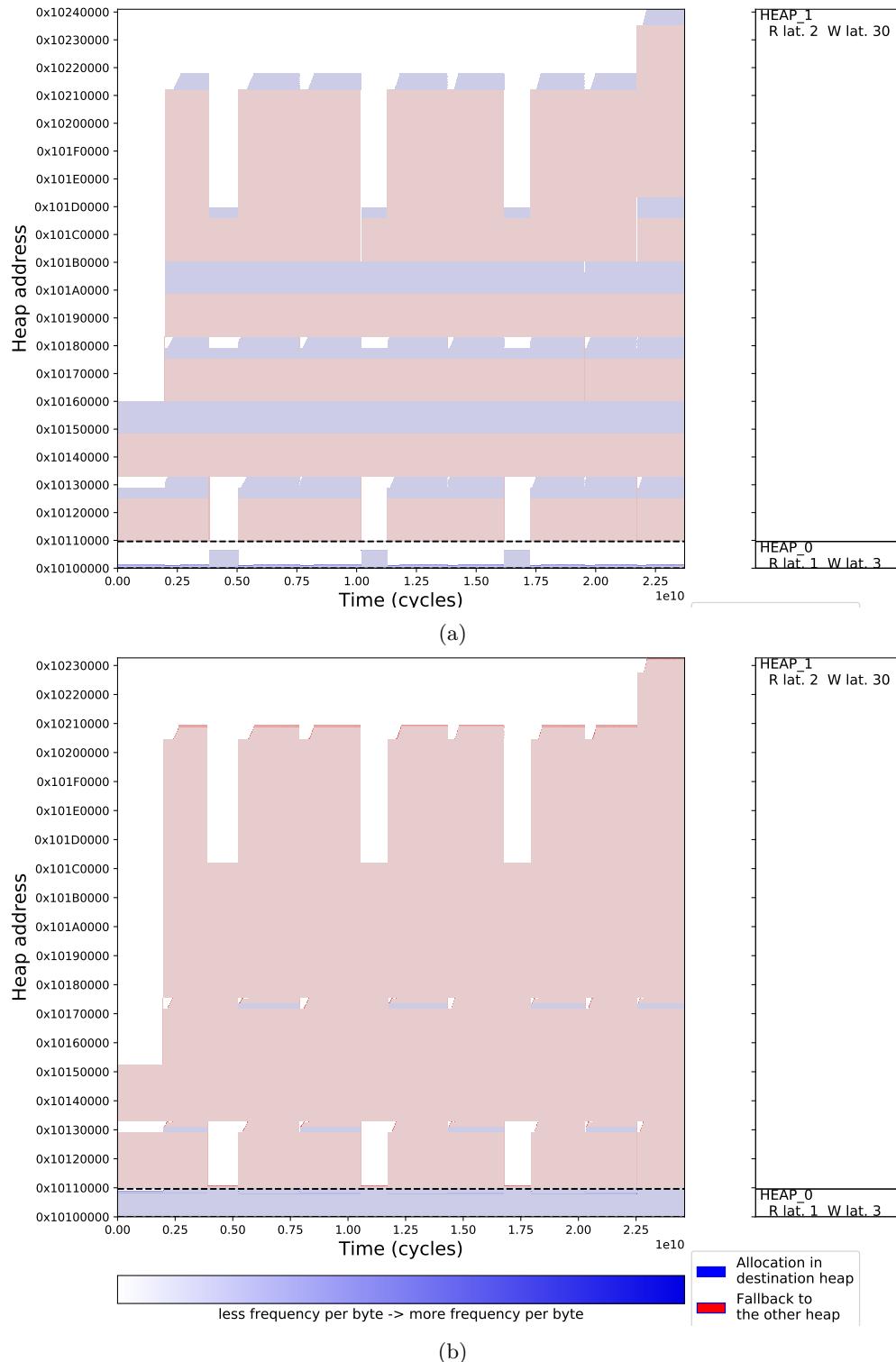


FIGURE 9.6 – Occupation du tas et fallback : H263 – Stratégie de sites d’allocation.
 (a) jeu de données 0, longueur de profil 29 ($L = L_{opt}$)
 (b) jeu de données 0, longueur de profil 32 ($L = L_{opt} + \delta$)

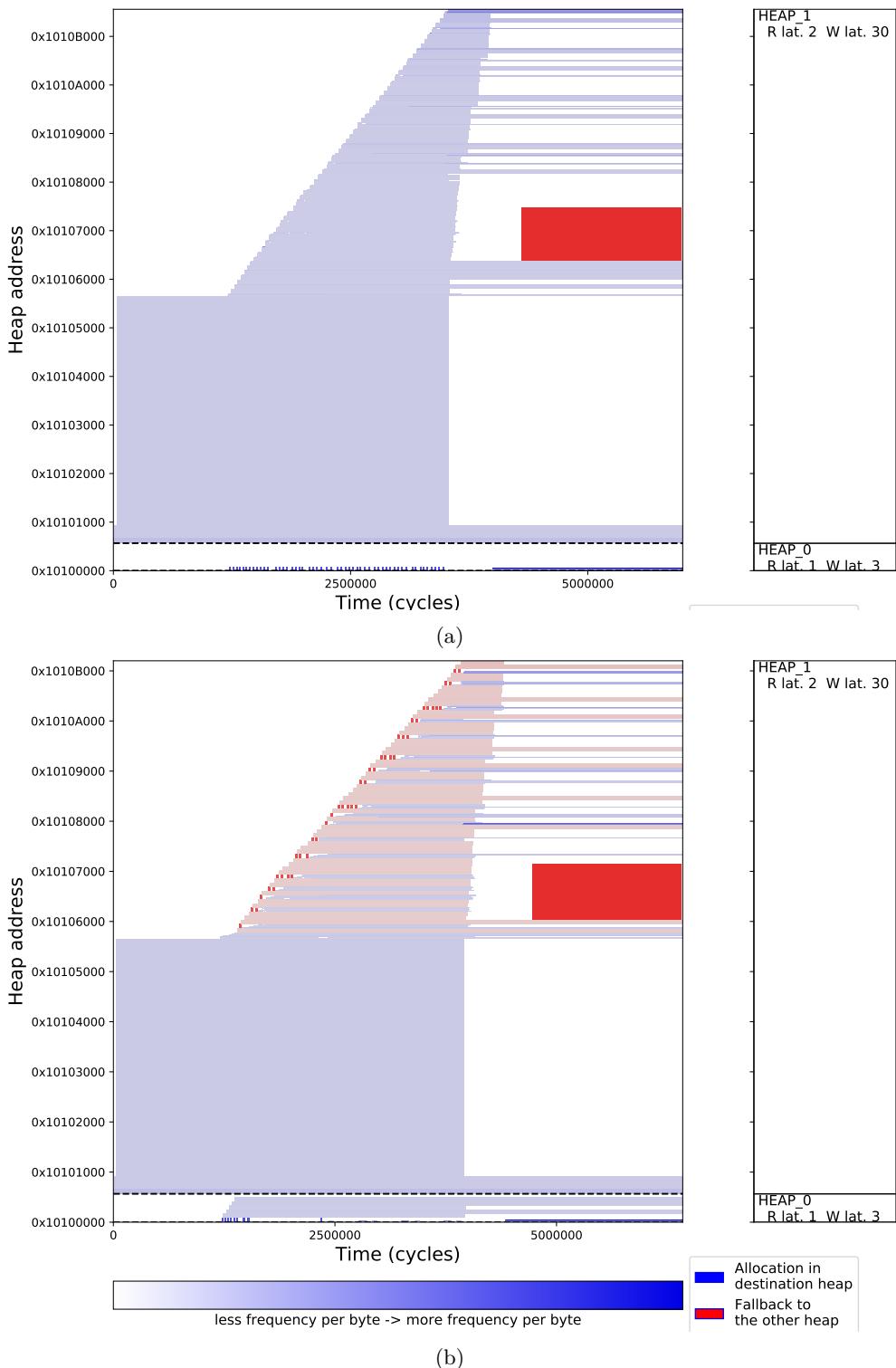


FIGURE 9.7 – Occupation du tas et fallback : Json – stratégie de sites d’allocation.
 (a) jeu de données 2, longueur de profil 5 ($L = L_{opt}$)
 (b) jeu de données 2, longueur de profil 6 ($L = L_{opt} + \delta$)

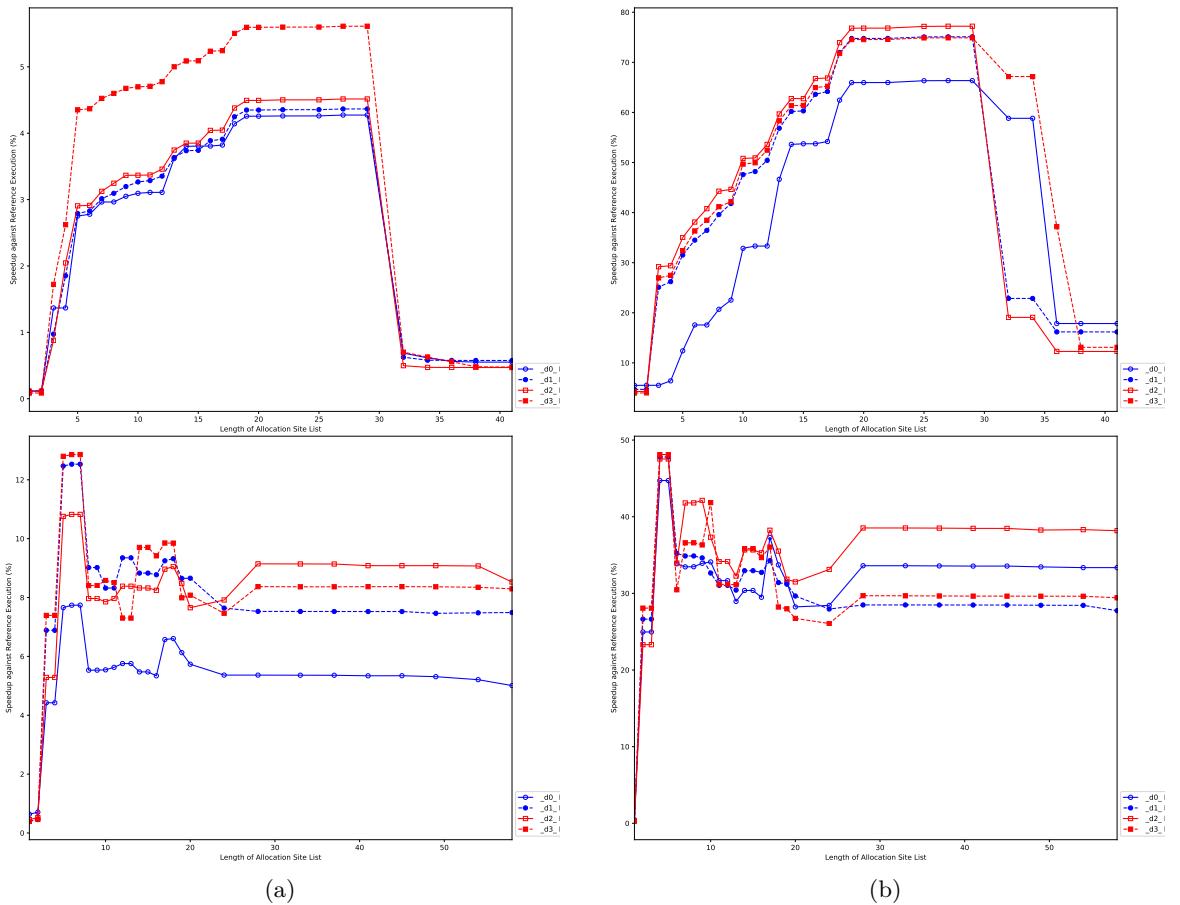


FIGURE 9.8 – Exploration de la longueur du préfixe – **Influence des technologies cibles**. Technologies mémoire : (a) scénario "*MRAM*", (b) scénario "*ReRAM*". Applications (de haut en bas) H263, jpg2000. Jeux de données de 0 à 3, architecture 5% de mémoire rapide.

9.3.2 Résultats de la stratégie

Nous pouvons maintenant présenter les résultats de cette stratégie de la même manière que pour les autres stratégies présentées dans nos travaux. Cette stratégie est évaluée sur des jeux de données différents, présentant des variations de profil d'allocation et de profil d'accès. La figure 9.9 présente les résultats de cette stratégie en termes de réduction du temps d'exécution, en fonction de l'architecture mémoire. Les résultats de la stratégie (courbe orange) correspondent au maxima de performances obtenu par l'exploration de la longueur de préfixe pour chaque couple application/architecture.

Notre stratégie obtient des résultats meilleurs ou identiques à la stratégie "Fast First" dans tout les cas sauf deux : l'application Dijkstra sur 10% de mémoire rapide et l'application json sur 75% de mémoire rapide. Pour ce qui est de json, dans le cadre de notre étude la zone d'intérêt de ces expériences concerne surtout les résultats pour une architecture mémoire contenant moins de 50% de mémoire rapide. Au contraire l'application Dijkstra présente des résultats non satisfaisants pour des architectures mémoires comprenant peu de mémoire rapide. En effet elle exhibe un profil d'allocation très régulier, allouant uniquement des objets de 16 octets, avec peu de variations dans le profil d'accès, les objets étant accédés entre 1 et 300 fois environ. De plus, l'application ne présente que cinq sites d'allocation dont trois dans la librairie standard, générant peu d'allocations. cette stratégie n'est donc pas adaptée à ces conditions.

Applications Json et Jpeg2000

Pour les applications json et Jpeg2000 la stratégie de site d'allocation fait mieux que la baseline, sans dégager des gains proche de l'optimal. Toutefois les résultats dégagés sur les architectures à 5% de mémoire rapide sont significativement meilleurs que la stratégie "Fast First". Ces deux applications présentent des profils d'allocation en rampe. Pour l'application Jpeg2000, des objets chauds sont alloués vers la fin de l'exécution, par les mêmes sites que ceux allouant des objets un peu moins chaud durant toute l'exécution. Or à ce moment là ces objets n'ont plus de place dans le tas rapide. C'est à cause de l'allocation de ces objets dans le tas lent que cette application ne présente pas des résultats proches de l'optimal. L'application Json quand a elle alloue trop d'objets n'étant pas complètement froids, et n'a pas la place de les allouer dans le tas rapide. Dans ce cas là, l'ILP peut en allouer une partie dans le tas rapide, mais la stratégie de sites d'allocation ne peut pas se permettre d'inclure ce site sous peine de ne plus avoir de place pour les objets les plus chauds de l'application.

Application H263

La stratégie de sites d'allocation pour l'application H263 dégage des gains près de l'optimal. En effet, sur une architecture comprenant 5% de mémoire rapide, cette stratégie réduit le temps d'exécution de 4,69% quand avec 100% de mémoire rapide la réduction du temps d'exécution est de 5,27% pour le scénario "*MRAM*". Pour l'autre scénario technologique la réduction du temps d'exécution est de 73,4% (84,5% avec 100% de mémoire rapide).

Application Ecdsa

L'application Ecdsa présente un profil d'allocation et un profil d'accès varié, permettant de réduire significativement le temps d'exécution. En effet, passer de 0% à 100% de mémoire rapide réduit le temps d'exécution de 25,3% en moyenne sur les jeu de données d'évaluation pour le scénario "*MRAM*", et 98% pour le scénario "*ReRAM*".

La stratégie de site d'allocation n'arrive toutefois pas à dégager de bons résultats sur une architecture 5%. En effet sur les huit sites, un seul alloue des objets ayant une fréquence d'accès par octet élevée. Mais ce site alloue des objets très variés dont certains sont très peu accédés. Ainsi la mémoire rapide est vite saturée par les objets de ce site d'allocation, dont rien ne garantit qu'il soient les plus chauds, expliquant que sur cette architecture mémoire la stratégie de site d'allocation se comporte sensiblement comme la stratégie "Fast First".

Dès que la taille du tas augmente en revanche, les gains de la stratégie s'améliorent grandement pour atteindre des résultats proches de l'optimal dès 25% de mémoire rapide. En effet, dès que le tas rapide peut contenir une plus grande partie des objets de ce site d'allocation, plus d'objets

chauds y sont alloués. Nous notons également que cette augmentation de la taille du tas profite beaucoup plus à la stratégie de site d'allocation. En effet, même si elle n'alloue pas tout les objets du seul site allouant des objets chauds, et contrairement à la stratégie "Fast First" elle n'y alloue aucun objet froid des autres sites d'allocation.

Conclusion

Dans son ensemble cette stratégie de placement en ligne permet d'améliorer les performances, pour certaines jusqu'à des performances proches de l'optimal. Néanmoins, pour d'autres applications, elle se heurte a plusieurs limitations dont nous allons maintenant discuter. Les principales pistes d'amélioration découlent des exemples applicatifs ci dessus :

- Mieux exploiter un profil d'allocation alloué majoritairement par un seul site d'allocation.
- Rendre la stratégie plus efficace face à un profil d'allocation en rampe (Json, Jpeg 2000)

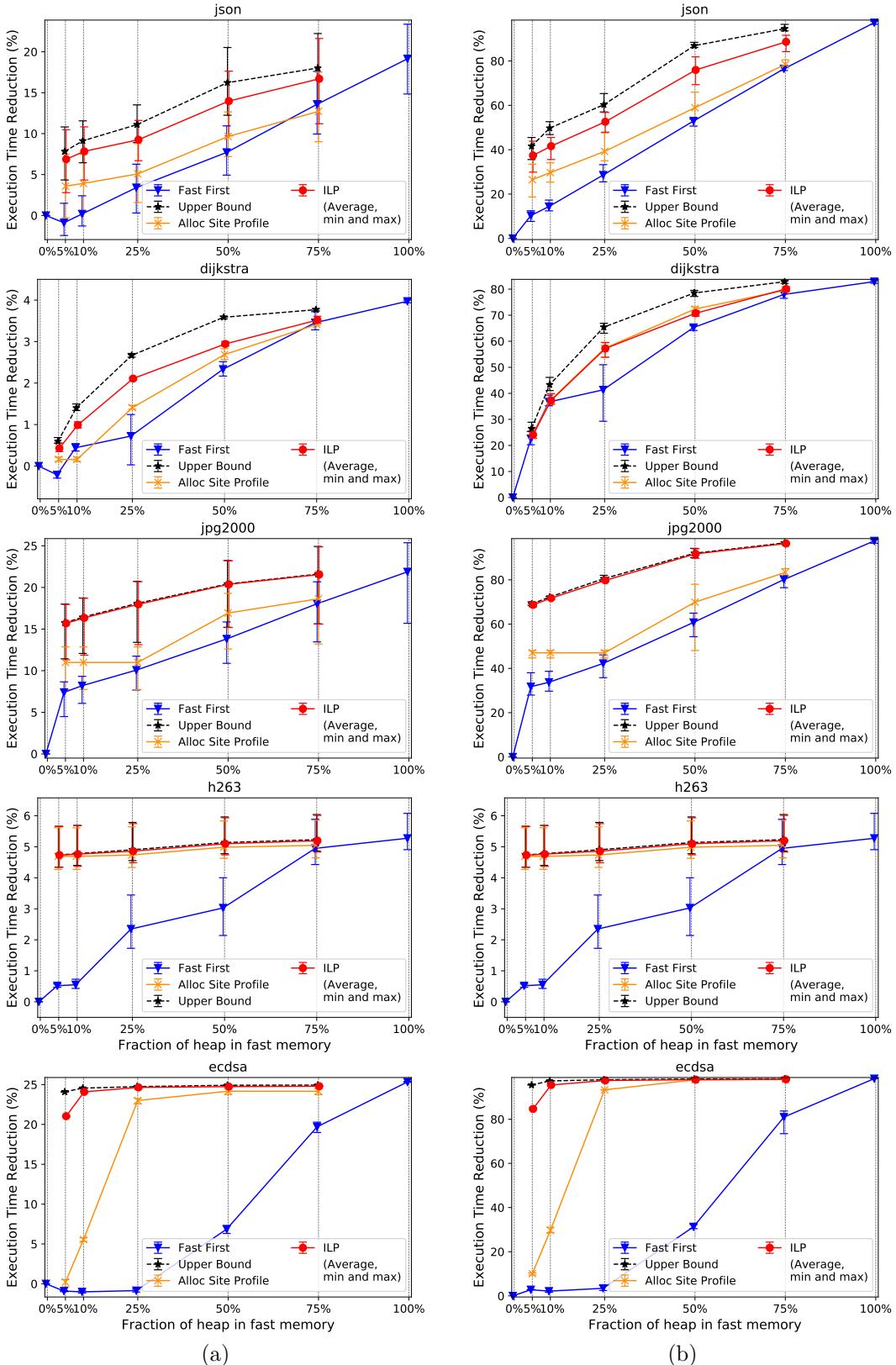


FIGURE 9.9 – Évaluation des performances – stratégie en ligne "profil de sites d'allocation"
 (a) Technologies mémoires scénario "*MRAM*" (b) Technologies mémoires scénario "*ReRAM*"
 Jeux de données 0 à 3.

9.4 Limitations de la stratégie proposée et Perspectives

Des différentes expériences menées sur cette stratégies plusieurs points ressortent. Dans un premier temps la méthodologie d'exploration de la meilleure longueur pour la stratégie est trop lourde. En effet, et même si nous avons utilisé des technologies de simulation plus rapides qu'une simulation VHDL comme discuté au chapitre 7, le fait de devoir explorer chaque couple longueur de préfixe / architecture pour une application est trop consommateur en temps pour être raisonnable. L'exploration de la longueur de préfixe de l'application h263 dépassant la semaine de calcul sur un serveur possédant 16 coeurs de calculs

Toutefois, nous avons déterminé qu'avec notre approche, la longueur optimale est proche de l'apparition du fallback. Ainsi pour une liste de sites correctement ordonnée il suffit de déterminer combien de sites ont la place d'être alloués dans le tas rapide. Nous pourrions donc utiliser cette information pour réduire significativement l'exploration. Par exemple une recherche dichotomique serait bien plus efficace et permettrait de nécessiter un nombre de simulations logarithmique en fonction du nombre de site d'allocation du programme. A noter que cette recherche doit être initialisée à l'exécution de la stratégie comprenant un seul site d'allocation. Ainsi, le cas où l'application alloue trop d'objets depuis son site le plus chaud pour qu'ils tiennent tous dans le tas rapide peut être détecté immédiatement. L'application Ecdsa est un bon exemple de ce type de comportement.

Deuxièmement, les applications qui présentent un profil d'allocation en rampe ne permettent pas d'atteindre des résultats proches de l'optimal. En effet ces applications allouent des objets ayant une fréquence d'accès par octet élevée au sommet de la rampe, c'est à dire, tard dans leur exécution. À ce moment là, et pour une application allouant en rampe, le tas rapide sera potentiellement saturé, sans avoir réussi à préserver d'espace pour ces objets chauds. Ainsi, les meilleures performances sont obtenues pour des longueurs de préfixe plus faibles sous utilisant le tas rapide. Contrairement à ce comportement une application allouant et désallouant régulièrement des objets (comportement en pics ou en plateaux) libérera de l'espace tout au long de son exécution et à donc une chance d'encore disposer d'assez de place pour allouer un objet chaud dans le tas rapide même en fin d'exécution. Nous développerons plus bas de quelle manière nous avons essayé de rendre la stratégie plus conservative dans ces cas de figure pour augmenter la longueur de préfixe.

Enfin, l'application Ecdsa présente un potentiel de gain fort, mais un seul site d'allocation contributif, c'est à dire allouant des objets très accédés en nombre important. En effet la stratégie de résolution hors ligne ILP dégage plus de 20% de réduction du temps d'exécution sur 5% du tas en mémoire rapide pour le scénario "*MRAM*". Toutefois la stratégie de site d'allocation présente des résultats équivalents à la stratégie baseline sur 5% de mémoire rapide car comme nous l'avons expliqué précédemment le seul site d'allocation contributif alloue beaucoup d'objets, dont certains froids. Ainsi, le dispatcheur se retrouve à allouer la majorité des objets chauds de l'application en mémoire lente faute de place en mémoire rapide. Cette stratégie pourrait cependant être améliorée pour ce cas de figure. Il est généralement possible dans notre contexte de réaliser une introspection de pile pour déterminer le site d'appel de la fonction allouant l'objet. Ainsi, en remontant la pile d'appel il semble possible de dégager des différences dans la pile d'appel aboutissant à l'allocation de chaque objet. La liste construite ne serait donc plus par site d'allocation mais par site d'appel d'un site d'allocation. Nous n'avons toutefois pas eu le temps d'explorer cette piste pour l'application Ecdsa.

Nous allons maintenant aborder les pistes que nous avons suivies pour tenter d'améliorer les performances de cette stratégie.

9.4.1 Longueur de profil dynamique

Comme nous l'avons vu les applications présentant un profil d'allocation en rampe ne présentent pas de bons résultats pour la stratégie de site d'allocation. Nous avons proposé ci dessus l'explication que contrairement à un profil d'allocation en pics ou en plateau une telle application ne libère pas de mémoire et a ainsi de grande chances de ne plus avoir de mémoire rapide en fin d'exécution pour des objets intéressants.

Nous avons donc évalué une approche plus dynamique, ayant pour but d'allouer plus de sites dans le tas rapide tout en conservant la place nécessaire à l'allocation des objets chauds de la fin

de l'exécution dans le tas rapide. Cette stratégie se base sur la même liste de site d'allocation que celle présentée plus haut dans le chapitre mais propose de faire varier le préfixe en fonction du remplissage du tas. L'intuition derrière cette stratégie étant que plus le tas rapide est rempli, plus la stratégie doit être sélective vis à vis des objets qu'elle alloue dedans. Nous avons donc implémenté dans le dispatcheur un mécanisme pour surveiller le remplissage du tas, de manière à réduire la longueur du préfixe dynamiquement en fonction du remplissage du tas.

La stratégie que nous avons évaluée est donc, partant d'une valeur initiale pour la longueur du préfixe de réduire cette longueur en fonction du remplissage du tas linéairement de manière à ce qu'une fois le tas rempli la stratégie ne considère plus qu'un seul site. Mais cette approche présente néanmoins plusieurs problèmes. L'exploration de la longueur de préfixe initiale ne permet pas d'éviter la phase d'exploration de la stratégie statique. De plus le calcul précis de l'occupation du tas nécessiterait de suivre en détail la fragmentation du tas. Ce calcul représenterait un surcoût prohibitif, nous avons donc choisi de ne calculer qu'une approximation du remplissage du tas en ligne, ignorant la fragmentation. Toutefois, cette approche ne nous a pas permis d'améliorer les résultats par rapport à la stratégie de site d'allocation, présentant des résultats similaires à celle-ci. Dans ces conditions, le surcoût de calculer le remplissage du tas, même en négligeant la fragmentation, dégrade les performances.

9.4.2 Informations de placement de la solution hors ligne ILP

L'autre piste que nous avons suivie est celle d'utiliser les résultats de la stratégie ILP pour améliorer la liste de site d'allocation. En effet une des limitations principales de la stratégie de site d'allocation est qu'elle n'est pas capable de préserver de l'espace dans le tas pour des allocations futures. C'est ce qui nous a conduit à essayer de rendre la longueur du préfixe dynamique, sans succès.

Une autre possibilité pour parvenir à ce but est de prendre en compte les décisions de placement du solveur ILP pour améliorer la stratégie de site d'allocation. Ainsi, il serait possible de tirer parti du fait que le solveur place les objets hors ligne pour les jeux de données de profilage au niveau du site d'allocation. Par exemple, Le solveur va placer les objets les plus chauds d'une exécution en mémoire rapide, Or nous avons conclu au chapitre 7 que les sites d'allocation allouent des objets qui ont souvent des fréquences d'accès par octet proches. Le solveur peut alors être amené à placer les objets du deuxième site le plus chaud en mémoire lente pour préserver l'espace du tas nécessaire à l'allocation des premiers. Ce comportement requiert la connaissance du profil d'allocation et d'accès à l'avance. Toutefois ces comportements temporels peuvent être communs à toutes les exécutions d'une application, même si le profil d'allocation et d'accès varie selon le jeu de données. Notre objectif est donc de prendre en compte ces comportements temporels dans la stratégie de site d'allocation.

Nous avons construit une liste de site d'allocation basé sur les décisions de placement de l'ILP, indiquant par site d'allocation la décision de placement retenue par le solveur. Nous désignons cette liste par site d'allocation basé sur la résolution du problème hors ligne par le solveur ILP "liste ILP" dans la suite. Un exemple de ces informations est donné en figure 9.10 pour l'application H263 sur l'architecture 5% de mémoire rapide et pour le jeu de données 0. La solution ILP considérée parmi les différentes solutions ayant des compensations de la fragmentation est celle présentant les meilleures performances (voir chapitre 8). Les informations de placement de l'ILP sont retenues sans tenir compte du mécanisme de fallback. Ainsi nous obtenons pour chaque site d'allocation le pourcentage d'objets alloués par ce site placé en mémoire rapide par le solveur ILP.

On peut y voir que les 13 objets alloués par la fonction `ReadImage()` pourtant classé 16ème ($L_{opt} = 29$) site d'allocation le plus chaud est alloué par le solveur ILP dans le tas lent. Ainsi le profil d'allocation semble donner lieu à plus de gains potentiels si ces objets sont alloués en mémoire lente pour préserver le tas rapide. De la même manière que précédemment nous agrégeons ces informations pour les différents jeux de données de profilage pour construire la liste ILP.

Partant de cette liste, nous avons évalué plusieurs stratégies :

- Utiliser la liste ILP directement pour la stratégie de site d'allocation
- Construire une liste conservatrice pour le tas rapide basé sur la liste ILP
- Améliorer l'ordre de la liste de site d'allocation avec la liste ILP

Site d'allocation	Nb objects	$freq^{acces}_{octet}$ moyenne	tas rapide ILP
__smakebuf_r	24	2091.4	95%
_fmemopen_r	24	195.10	95%
MotionEstimation	6048	99.746	100%
MB_Decode	2706	41.355	100%
CodeOneIntra	396	41.355	100%
MotionEstimation	6048	28.960	100%
MB_Recon_P	3168	14.009	100%
CodeOneIntra	1	9.2035	100%
MB_Decode	2706	7.2785	100%
CodeOneIntra	396	7.1889	100%
Predict_P	3131	5.8266	100%
CodeOneOrTwo	37	4.2070	100%
MB_Encode.constprop.0	396	2.4815	100%
MB_Encode	3168	2.0633	100%
__sfmoreglue	2	1.7549	100%
ReadImage	13	1.0001	0%
InitImage	22	0.55367	95%
NextTwoPB	3	0.37480	100%
NextTwoPB	3	0.37461	100%
main	1	0.28668	100%
main	1	0.049551	100%
InterpolateImage	8	0.0055128	0%
NextTwoPB	3564	0.0043384	81%
main	1	0.0025987	0%
InitImage	22	0.0024862	0%
InitImage	22	0.0020807	23%
InitImage	22	0.0020804	36%
MotionEstimatePicture	19008	0.0020156	10%
CodeOneOrTwo	912	0.0012886	0%
main	1	0.0011707	0%
main	1	0.0011705	0%
CodeOneOrTwo	912	0.00092084	0%
CodeOneOrTwo	1056	0.00090616	0%
main	1	0.00012607	0%
main	1	0.00012541	0%
main	1	9.0302e-05	0%
main	1	7.5253e-05	0%
main	1	5.8156e-05	0%
main	1	0.0	0%

FIGURE 9.10 – Sites d'allocation et placement ILP, application h263, jeu de données 0, arch. 5%
Pour chaque site d'allocation, en ligne, nous présentons la fonction contenant le site, le nombre d'objets alloués, la fréquence d'accès par octet moyenne du site et le pourcentage des objets du site alloué par l'ILP dans le tas rapide.

Liste ILP

Nous avons évalué la construction d'une liste de site d'allocation pour la stratégie en ligne de site d'allocation basé uniquement sur les pourcentages de placement du solveur ILP. L'idée étant de prendre en compte au plus finement possible les comportements temporels que l'ILP prend en compte dans sa solution. Mais l'application de cette liste à la stratégie de site d'allocation présente de mauvais résultats. En effet, la stratégie en ligne se basant sur les informations de site d'allocation n'est pas capable de garder de la place pour des allocations futures. Ainsi, un site n'étant pas alloué en mémoire rapide pour garder de la place dans le tas rapide ne permettra pas forcément d'en garder. En effet la stratégie est appliquée à la granularité du site d'allocation. Ainsi, garder de la place pour les objets d'un site d'allocation alloués vers la fin de l'exécution n'est pas possible dans un cas de figure normal. Si l'intégralité des sites placés au dessus du site concerné dans la "liste ILP" sont placés à 100% dans le tas rapide, alors il sera possible de garder de la place en utilisant cette liste. En effet, si un site n'est pas placé à 100% dans le tas rapide, les objets de ce site que l'ILP place en mémoire lente prendront la place que la stratégie essaye de préserver. Le cas favorable à cette stratégie ne se présentant pas dans nos applications, cette piste ne présente pas de bons résultats.

Liste conservative

Partant de cette considération nous avons étudié la possibilité de construire une liste basée sur les décisions du solveur ILP mais n'incluant que les sites placés à 100% par l'ILP dans le tas rapide.

Cette approche vise principalement les applications ayant un comportement d'allocation en rampe, comme Json et Jpeg 2000. Les résultats de cette approche n'améliorent toutefois que très légèrement les performances de l'application Json. Nous n'avons pas observé de différence pour l'application Jpeg 2000. Et pour finir, sur les applications ne présentant pas un profil d'allocation en rampe, cette stratégie présente une réduction du temps d'exécution inférieure à la baseline "Fast First". En effet, si il n'y a pas assez de sites placés à 100% en mémoire rapide par l'ILP cette stratégie ne peut pas fonctionner. Elle exclut également les sites allouant en moyenne des objets avec des fréquences d'accès par octet très élevées mais dont le taux de placement en mémoire rapide n'est pas de 100%.

Liste améliorée

Partant des réflexions précédentes nous présentons une stratégie de site d'allocation améliorée par la "liste ILP". La stratégie que nous avons évaluée dans ce chapitre présente des résultats quasi optimaux dans plusieurs cas. Partant de cette liste de sites nous avons intégré les informations de la résolution ILP que nous pouvons appliquer à la granularité du site d'allocation. En effet nous avons vu ci dessus que les informations de l'ILP ne sont pas directement applicable à une stratégie car elles s'appliquent à la granularité de l'objet. Nous avons donc limité la modification de la liste de sites à la permutation de sites que les solutions ILP calculées sur les jeux de données de profilage ont placées à 100% ou à 0% en mémoire rapide. En effet, un site alloué à 100% par l'ILP devrait se retrouver en haut de la liste de sites d'allocation, alors qu'un site alloué à 0% par l'ILP en mémoire rapide peut être placé en bas de la liste.

Résultats

Cette approche ne garanti pas une amélioration systématique, mais garantit l'application de décisions ILP au niveau de granularité où la liste est utilisée. Nous présentons les résultats de cette stratégie en figure 9.11.

Nous pouvons y comparer les résultats de la stratégie liste améliorée (courbe violette) à ceux de la stratégie de site d'allocation (courbe orange). Pour pouvoir situer ces stratégies dans un intervalle de performances nous incluons les résultats des stratégies "Fast First" et ILP.

Cette approche conserve de bons résultats pour les applications H263 et EcDSA (résultats proches de l'optimal pour la stratégie de sites d'allocation). Nous distinguons également une amélioration des résultats pour les applications présentant un profil d'allocation en rampe (Jpeg 2000 et Json), pour les architectures contenant peu de mémoire rapide. L'application Json par

exemple sur l'architecture 5% de mémoire rapide passe d'une réduction du temps d'exécution de 3,4% à 4,1%. Pour l'architecture composée de 50% de mémoire rapide l'amélioration est plus importante, passant de 9,6% à 12,5%, sachant que l'optimal est de 13,8%. Cette combinaison d'application et d'architecture est toutefois la seule qui atteint une amélioration des performances significative.

9.4.3 Discussion

Les résultats de l'amélioration de la liste de sites d'allocation basée sur les informations de l'ILP ne permet pas de significativement améliorer les performances de nos applications. Dans le cas de l'application Ecdsa, nous pensons que l'amélioration des performances passe par une introspection de la pile de manière à pouvoir séparer les objets du seul site contributif.

Toutefois, pour les applications Json et Jpeg 2000, les performances de notre stratégie de sites d'allocation pourraient encore être largement améliorées. L'introspection de pile pourrait également améliorer les performances de ces applications en fournissant à la stratégie plus de finesse dans la classification des objets par site. Il est également possible qu'il n'y ait pas plus d'informations permettant l'amélioration de la stratégie en fonction de la pile d'appel. C'est le cas pour l'application Dijkstra. En effet, celle-ci possède uniquement deux sites d'allocation dans la fonction principale de calcul, les autres sites d'allocation pour son exécution étant dans la librairie standard.

Pour les applications Jpeg 2000 et json réaliser une analyse des profils d'allocation et d'accès par taille pour chaque site d'allocation est une piste intéressante.

L'application Dijkstra au contraire représente une limite pour notre approche. En effet, au-delà de n'avoir que deux sites d'allocation contributifs, les profils d'allocation et d'accès présentent trop peu d'hétérogénéité à exploiter. Les objets sont de la même taille, avec peu de variation de la durée de vie et accédés sensiblement de la même manière.

9.5 Conclusion

Nous avons présenté dans ce chapitre une stratégie de résolution en ligne du problème de placement des objets du tas en mémoire hétérogène. Ayant préalablement estimé les gains atteignables par cette méthode nous pouvons affirmer que pour une partie de nos applications nous atteignons des résultats proches de l'optimal. Dans la majorité des cas nous améliorons les performances de la stratégie baseline "Fast First" et nous atteignons des résultats plus proches de l'optimal que de la baseline pour les architectures mémoires contenant peu de mémoire rapide (5% à 25%).

Nous avons mis en avant le comportement de l'application Ecdsa, ne présentant pas assez de site d'allocation pour exploiter efficacement l'hétérogénéité du profil d'allocation pour les architectures avec peu de mémoire rapide et avons proposé une piste d'amélioration basée sur l'introspection de pile.

Nous avons également étudié l'amélioration des performances des applications présentant un profil d'allocation en rampe en proposant une stratégie de site d'allocation amélioré en utilisant les solutions hors ligne calculées en utilisant la programmation linéaire en nombres entiers.

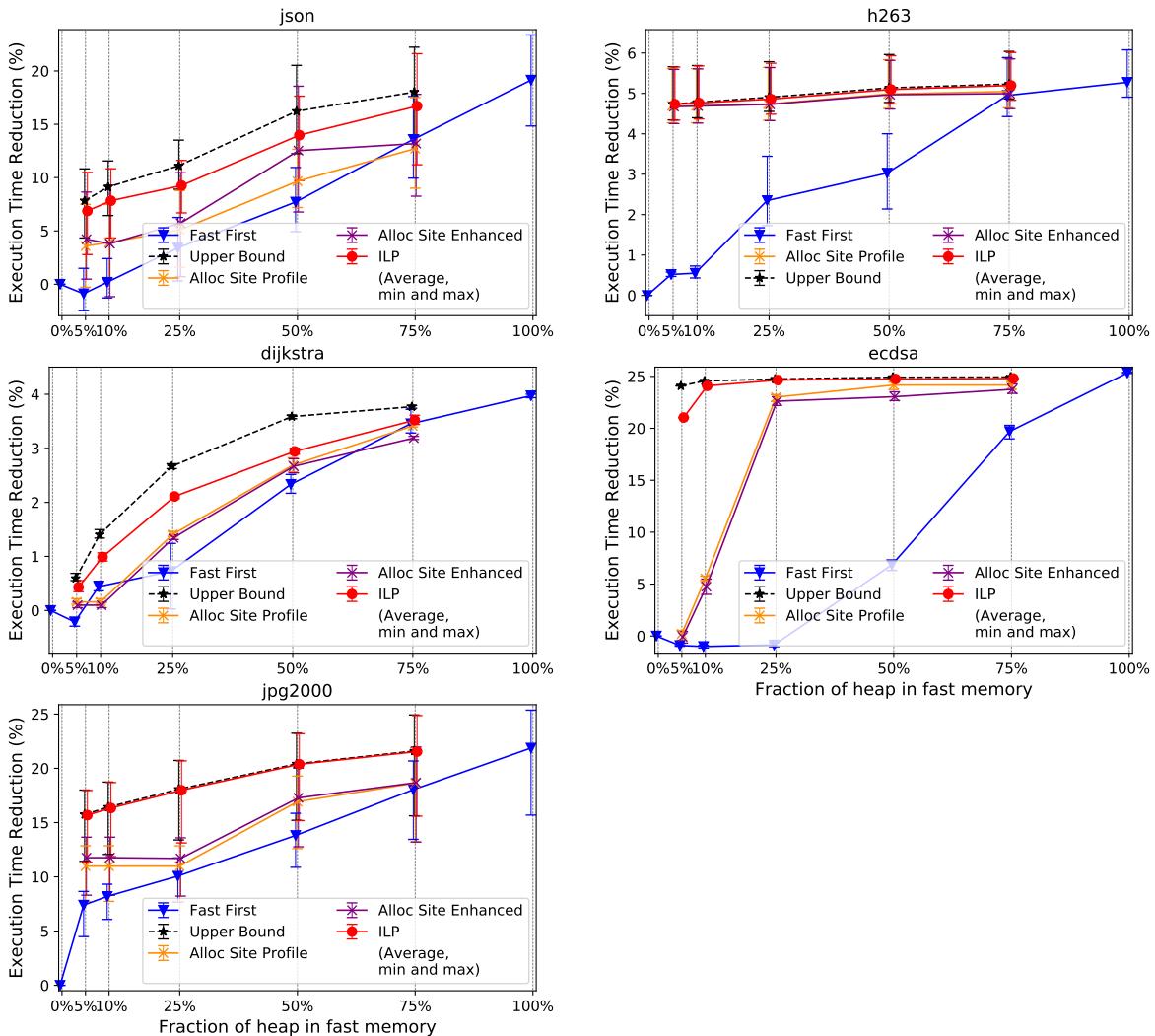


FIGURE 9.11 – Évaluation des performances – stratégie en ligne "sites amélioré par ILP"
Technologies mémoires scénario "MRAM"
Jeux de données 0 à 3.

Chapitre 10

Bilan et perspectives

Tout au long de ce travail, nous avons étudié les performances des applications embarquées sur des plateformes présentant une hétérogénéité dans leur mémoire de travail. Contrairement aux domaines du calcul haute performance et NUMA, les plateformes que nous avons considérés contraignent fortement les mécanismes logiciels utilisables. Ainsi, nous ne pouvons pas nous reposer sur l'usage d'un cache matériel ou d'une MMU permettant l'utilisation de la mémoire virtuelle et ainsi autorisant le déplacement des objets du tas.

10.1 Bilan de la thèse

Pour l'étude du mécanisme d'allocation multi-tas, nous avons mis en œuvre des simulations, qui nous ont permis de prendre en compte plusieurs scénarios technologiques. De plus la simulation nous a permis de réaliser une exploration architecturale permettant d'évaluer l'intérêt de placer le tas d'une application en mémoire hétérogène.

Nous avons ensuite proposé une formalisation du problème de l'allocation des objets du tas en mémoire hétérogène. Nous avons évalué une solution naïve à ce problème, la stratégie "Fast First". Cette étude nous permet d'affirmer que cette stratégie, bien qu'améliorant les performances dans la majorité des cas ne permet jamais d'atteindre des gains importants.

Pour quantifier plus finement les performances des stratégies que nous souhaitions étudier, nous avons donc entrepris de déterminer l'impact maximal sur les performances de la résolution de placement. Cette quantification est paramétrée par l'architecture mémoire, l'application cible et le jeu de données d'entrée. En effet, il est certain que l'application ne pourra pas aller plus vite que si le tas était entièrement localisé en mémoire rapide. Mais l'intérêt est ici de constater quelles performances sont atteignables pour un pourcentage donné du tas de l'application en mémoire rapide.

Nous avons donc modélisé le problème sous forme de programme linéaire en nombres entiers, et exécuté les placements calculés hors ligne pour borner l'optimal. En effet un des aspects les plus contraignants de cette étude est le suivant. Au delà de ne pas connaître la taille, le nombre et l'ordre d'arrivée des requêtes d'allocations et de désallocation, il n'est pas trivial pour autant de caractériser la manière dont chaque tas gère le problème de fragmentation de manière interne. De plus l'étude du placement changeant la répartition des objets entre les tas, la résolution des problèmes de fragmentation de chaque tas ne peut être ni connue à l'avance, ni modélisée à l'intérieur de la solution ILP. Nous avons donc proposé une sur-approximation et une sous-approximation des performances qu'obtiendrait un placement optimal pour une application et un jeu de données fixés s'exécutant sur une architecture particulière. Le majorant et le minorant de l'optimal étant généralement proches, nous avons donc une estimation fine de l'optimal.

Nous avons ainsi pu mettre en évidence le fait que pour une majorité des programmes étudiés il était envisageable d'atteindre au moins la moitié des gains atteignables en disposant uniquement de 5% ou 10% de mémoire rapide. Ce résultat contraste très clairement avec l'amélioration de performances médiocre dégagée par la stratégie "Fast First" et prouve que la mise en place d'une stratégie de placement intelligente est nécessaire pour tirer parti correctement de l'hétérogénéité mémoire.

Le reste de cette étude est donc consacré à la réalisation d'une telle stratégie. Nous avons construit une métrique permettant l'évaluation de l'importance de l'objet dans la résolution du problème de placement, la fréquence d'accès par octet. Mais évaluer les objets à posteriori ne permet de construire une stratégie que si il est possible de corrélérer cette évaluation à une information disponible lors de futures exécutions. Le site d'allocation des objets est l'information qui nous permet d'évaluer les objets en moyenne, de les grouper plutôt que de les observer individuellement. De plus, évaluer les sites d'allocation au travers des objets qu'ils allouent permet de prendre des décisions qui, contrairement aux décisions prises à l'échelle de l'objet, sont applicables à plusieurs exécutions avec des jeux de données différents.

Nous avons donc construit la stratégie de profil de site d'allocation sur l'analyse des traces d'exécution des jeux de données de profilage. Puis nous avons évalué cette stratégie sur l'autre moitié de nos jeux de données.

L'évaluation de cette stratégie montre que dans une partie des cas, elle peut dégager des résultats proches de l'optimal. Dans une autre partie des cas, cette stratégie présente des performances meilleures que la stratégie naïve Fast First sans pour autant se rapprocher de l'optimal d'une manière significative. Enfin, pour l'une de nos applications d'évaluation, la conclusion à tirer est une limite plus dure pour notre approche globale. En effet il est important de noter que le placement automatique des objets en mémoire hétérogène ne présente d'intérêt que si l'application cible présente suffisamment d'hétérogénéité, que ce soit en terme d'objets alloués ou d'accès à ces objets alloués.

Ainsi, des applications complexes, réalisant plus d'un millier d'allocations de tailles et de durées de vies variées durant leur exécution voient leur performances significativement augmentées par un placement intelligent, relativement aux technologies mémoires impliquées.

10.2 Perspectives

Nous allons maintenant discuter les possibles axes de recherches pouvant découler de ces travaux. En effet, partant d'un problème peu étudié, de nombreuses questions restent ouvertes. Nous avons identifié deux axes d'exploration des mécanismes logiciels proposés.

Le premier concerne l'élargissement des conditions considérées dans l'étude (nombre de bancs mémoire, performances évaluées), l'autre concerne les stratégies et notamment l'usage de modules matériels dédiés à la résolution du problème de placement. Nous allons donc aborder dans la suite ces axes et les implications probables qu'ils impliquent.

Dans la réalisation de ce projet nous avons choisi de limiter la complexité des phénomènes étudiés de manière à pouvoir les caractériser plus efficacement. Mais rien dans le domaine de l'embarqué contraint ou dans la recherche actuellement menée sur les mémoires non volatiles émergentes ne justifient cette restriction.

Les restrictions que nous avons considérées portent sur deux aspects : le nombre de technologies mémoires et les performances envisagées, et mesurées. Nous discuterons dans un premier temps de l'augmentation du nombre de mémoires à considérer dans le problème de placement puis nous reviendrons sur l'intégration d'autres critères d'évaluation des performances.

10.2.1 Augmenter l'hétérogénéité

Une des limitations de ce travail est le fait de réduire l'hétérogénéité à deux technologies mémoires accessible directement au processeur. Sans anticiper sur la suite de cette section contenant les performances, considérer 3 mémoires de tailles différentes présentant des latences différentes ouvre de nombreuses possibilités. On peut facilement imaginer un cas favorable où deux mémoires rapides ont des asymétries opposées permettant une ségrégation efficace entre objets chauds, mais surtout accédés en lecture, objets chaud accédés en écriture et objets froids. Mais comment placer les objets dans l'architecture mémoire suivante ? si l'on juxtapose trois bancs mémoires, un ayant pour latences (en lecture / écriture) 1/3 (m_0), la suivante 10/10 (m_1) et la dernière 5/100 (m_2) ?

Il n'y a pas de réponse à priori, ne serait-ce que parce que nous avons démontré que cette réponse dépend de l'application considérée et de la taille de chaque banc mémoire. Il serait de plus nécessaire de repenser les stratégies proposées dans ces travaux. En effet la stratégie "Fast

"First" implique de pouvoir ordonner les mémoires. La stratégie de site d'allocation ne serait pas non plus applicable en l'état. Mais cette hypothèse technologique remet également en cause la manière que nous avons eu de résoudre le problème hors ligne par utilisation d'un solveur ILP.

Plutôt que de considérer une variable du problème par objet b_i à placer il faudrait considérer une variable binaire par objet par banc mémoire tel que, si l'on reprend les notations du chapitre 8 :

$$x_i = \begin{cases} 1 & \iff \text{mem}(b_i) = m_0 \\ 0 & \iff \text{mem}(b_i) \neq m_0 \end{cases} \quad y_i = \begin{cases} 1 & \iff \text{mem}(b_i) = m_1 \\ 0 & \iff \text{mem}(b_i) \neq m_1 \end{cases} \quad z_i = \begin{cases} 1 & \iff \text{mem}(b_i) = m_2 \\ 0 & \iff \text{mem}(b_i) \neq m_2 \end{cases}$$

Sans oublier d'ajouter la contrainte suivante au problème, et ce pour chaque objet :

$$x_i + y_i + z_i = 1$$

On voit ici que la taille du problème a résoudre s'en augmentrait significativement. Or, dans les cas que nous avons étudiés, la limite de traçabilité du problème nous a forcé à utiliser un outil plus optimisant (cplex) au lieu de l'outil open source que nous avions mis en œuvre au départ (Ipsolve). En effet ce dernier n'a pas été capable de résoudre des problèmes d'une taille aussi conséquente que celle que nous considérons dans des temps raisonnables. Par exemple, le nombre d'objets pour la résolution du problème dans cette étude va jusqu'à 70 000 pour l'application H263. Nous soutenons donc que trouver une autre méthode d'estimation des gains optimaux, ou faire évoluer celle utilisée dans ces travaux est un travail nécessaire pour l'augmentation du nombre de mémoires considérées.

Toutefois, l'augmentation du nombre de technologies mémoire à prendre en compte semble des plus intéressantes, surtout si elle est considérée en parallèle de la piste d'amélioration suivante, l'addition de critères d'évaluation des performances supplémentaires.

10.2.2 Optimiser d'autres critères de performance

Un autre axe qui augmenterait les usages et l'intérêt de notre approche est l'intégration d'autres critères d'optimisation que le seul facteur du temps d'exécution considéré dans cette étude.

La modélisation et la mesure de l'énergie semble intéressante dans un premier temps et ce pour plusieurs raisons. La consommation des systèmes embarqués étant une donnée des plus importantes dans leur conception, il est possible de l'optimiser directement par notre approche. Mais du à la non-volatilité des technologies mémoires émergentes et au profil d'activité en pic des systèmes embarqués il est possible de combiner cet aspect avec l'étude du temps d'exécution. En effet, l'approche Normally-Off permet d'économiser la consommation statique du système et exécuter un calcul plus vite peut ainsi économiser de l'énergie. Modéliser l'énergie dans notre approche permettrait donc d'optimiser directement ce critère. Mais il deviendrait également possible d'évaluer les interactions entre optimisation du temps d'exécution et optimisation de la consommation énergétique dans un cadre Normally-Off.

À terme, une modélisation de critères comme l'endurance peut offrir des compromis intéressants pour des systèmes contraints visant à allonger leur durée de vie. Et ce, que ce soit en économisant leur batterie ou en préservant les mémoires à faible endurance. On comprend bien ici l'intérêt de considérer plus de deux mémoires, de manière à avoir à disposition de la stratégie un éventail de mémoires cibles plus grand pour pouvoir optimiser plusieurs critères de performances. En effet augmenter les choix possibles de la stratégie lui permet de mieux prendre en compte à la fois l'hétérogénéité du système mémoire ainsi que l'hétérogénéité des objets de l'application.

10.2.3 Non-volatilité des mémoires et exécutions intermittente

Prendre en compte plus de critères d'amélioration de l'exécution des applications est une piste intéressante. Mais l'interaction entre exécution intermittente et placement en mémoire hétérogène est une piste de recherche en soi. En effet, le compromis entre le placement optimisant le temps d'exécution pour augmenter le temps passé arrêté ou placement optimisant la consommation d'énergie est une piste à explorer. De plus, de tels systèmes ne comportent pas forcément

uniquement des mémoires non-volatiles, tel notre scénario technologique associant ReRAM et SRAM. L'exécution intermittente ou Normally-Off doit alors passer par une phase de checkpointing et de restauration. Cette phase présente un compromis entre d'une part, le coût lié aux copies RAM-NVRAM et d'autre part, le coût de placer directement certains objets dans une mémoire plus lente mais non-volatile. On peut également se demander à quel moment de l'exécution d'un programme il serait le moins coûteux de réaliser une telle sauvegarde des objets du tas. Ce dernier point peut être regardé sous l'angle du problème de placement dans un contexte Normally-Off. Dans ce cas, l'enjeu devient d'adapter la stratégie pour une coupure de courant prévue. On peut penser ici raisonner sur la durée de vie des objets. Il conviendrait alors d'allouer en priorité les objets dont la durée de vie se terminerait avant l'arrêt de l'alimentation dans la mémoire volatile. Ainsi le coût du checkpoint serait limité aux objets encore en vie à ce moment là (et potentiellement très accédés).

10.2.4 Proposer des stratégies plus complexes

Nous évoquerons ici un dernier point dont le développement semble intéressant. En effet la stratégie en ligne que nous avons proposée, basée sur un profil de sites d'allocation, n'atteint des résultats proches de l'optimal que dans un nombre limité de cas. Nous pensons donc qu'il est possible d'améliorer cette stratégie, ou d'en proposer une plus efficace.

Nous avons proposé dans les discussions du chapitre 9 des pistes d'amélioration directes. Dans un premier temps, l'introspection de pile peut permettre de mieux ségréguer les requêtes d'allocations d'un même site d'allocation. Et nous pensons qu'une analyse des objets par taille par site d'allocation pourrait exhiber d'autres régularités à exploiter.

Pour améliorer les stratégies d'allocation mémoire dynamique, il faut trouver des informations sur lesquelles baser leur décisions. Une possibilité pour disposer d'autres information est de réaliser une partie du profilage en ligne. Cho et al. [CPIDAP09] explorent cette piste pour l'allocation des données statiques d'applications multimédia. Ils utilisent des données de profilage réalisées avant l'exécution pour construire plusieurs profils et le choix du profil utilisé est pris durant l'exécution, basé sur un profilage en ligne léger utilisant des compteurs matériels.

Il semble difficile à première vue de baser les décisions d'une stratégie efficace uniquement sur un profilage en ligne. Toutefois, une approche où des informations partielles récoltées durant l'exécution permettent d'orienter une stratégie basée sur des informations récoltées hors ligne semble prometteuse. On peut aussi imaginer utiliser ce profilage en ligne pour surveiller le comportements de sites d'allocation présentant des chaleurs d'objets alloués moyenne très variables entre les exécutions.

Références Bibliographiques

- [ABC03] Federico ANGIOLINI, Luca BENINI et Alberto CAPRARA. « Polynomial-time algorithm for on-chip scratchpad memory partitioning ». In : *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. ACM. 2003, p. 318-326.
- [ABS02] Oren AVISSAR, Rajeev BARUA et Dave STEWART. « An optimal memory allocation scheme for scratch-pad-based embedded systems ». In : *ACM Transactions on Embedded Computing Systems (TECS)* 1.1 (2002), p. 6-26.
- [AH16] Shahid ALAM et Nigel HORSPOOL. « A Survey : Software-Managed On-Chip Memories ». In : *Computing and Informatics* 34.5 (2016), p. 1168-1200.
- [AMS14] Fayçal Ait AOUDA, Kevin MARQUET et Guillaume SALAGNAC. « 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*. IEEE. 2014, p. 1-4.
- [AODSK+13] Dmytro APALKOV, Adrian ONG, Alexander DRISKILL-SMITH, Mohamad KROUNBI et al. « Spin-transfer torque magnetic random access memory (STT-MRAM) ». In : *ACM Journal on Emerging Technologies in Computing Systems* 9.2 (mai 2013), 1-35. URL : <http://dx.doi.org/10.1145/2463585.2463589>.
- [ARM08] ARM. *mbedTLS*. <https://github.com/ARMmbed/mbedtls>. 2008.
- [ASME18] Shoaib AKRAM, Jennifer B SARTOR, Kathryn S MCKINLEY et Lieven EECKHOUT. « Write-rationing garbage collection for hybrid memories ». In : *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. 2018, p. 62-77.
- [ASME19] Shoaib AKRAM, Jennifer SARTOR, Kathryn MCKINLEY et Lieven EECKHOUT. « Crystal Gazer : Profile-driven write-rationing garbage collection for hybrid memories ». In : *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3.1 (2019), p. 9.
- [AWDSCDN10] Dmytro APALKOV, Steven WATTS, Alexander DRISKILL-SMITH, Eugene CHEN, Zhitao DIAO et Vladimir NIKITIN. « Comparison of Scaling of In-Plane and Perpendicular Spin Transfer Switching Technologies by Micromagnetic Simulation ». In : *IEEE Transactions on Magnetics* 46.6 (juin 2010), 2240–2243. URL : <http://dx.doi.org/10.1109/TMAG.2010.2041330>.
- [BBBR+11] Nathan BINKERT, Bradford BECKMANN, Gabriel BLACK, Steven K REINHARDT, Ali SAIDI, Arkaprava BASU, Joel HESTNESS, Derek R HOWER, Tushar KRISHNA, Somayeh SARDASHTI et al. « The gem5 simulator ». In : *ACM SIGARCH Computer Architecture News* 39.2 (2011), p. 1-7.
- [BDFK+14] A. BELMONTE, R. DEGRAEVE, A. FANTINI, W. KIM, M. HOUSSA, M. JURCZAK et L. GOUX. « Origin of the deep reset and low variability of pulse-programmed WAl2O3TiWCu CBRAM device ». In : *2014 IEEE 6th International Memory Workshop (IMW)*. Mai 2014, p. 1-4.
- [BDMRS17] G. BERTHOU, T. DELIZY, K. MARQUET, T. RISSET et G. SALAGNAC. « Peripheral state persistence for transiently-powered systems ». In : *2017 Global Internet of Things Summit (GIoTS)*. Juin 2017.

- [BDMRS18] Gautier BERTHOU, Tristan DELIZY, Kevin MARQUET, Tanguy RISSET et Guillaume SALAGNAC. « Sytare : a Lightweight Kernel for NVRAM-Based Transiently-Powered Systems ». In : *IEEE Transactions on Computers* (2018), p. 1-14. URL : <https://hal.archives-ouvertes.fr/hal-01954979>.
- [BPSY+08] Michael BUETTNER, Richa PRASAD, Alanson SAMPLE, Daniel YEAGER, Ben GREENSTEIN, Joshua R SMITH et David WETHERALL. « RFID sensor networks with the Intel WISP ». In : *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM. 2008, p. 393-394.
- [BS10] K. BAI et A. SHRIVASTAVA. « Heap data management for limited local memory (LLM) multi-core processors ». In : *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Oct. 2010, p. 317-325.
- [BSLBM02] Rajeshwari BANAKAR, Stefan STEINKE, Bo-Sik LEE, Mahesh BALAKRISHNAN et Peter MARWEDEL. « Scratchpad memory : A design alternative for cache on-chip memory in embedded systems ». In : *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627)*. IEEE. 2002, p. 73-78.
- [BWMAHBB15] Domenico BALSAMO, Alex S. WEDDELL, Geoff V. MERRETT, Bashir M. AL-HASHIMI, Davide BRUNELLI et Luca BENINI. « Hibernus : Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems ». In : *IEEE Embedded Systems Letters* 7.1 (mar. 2015), 15–18. URL : <http://dx.doi.org/10.1109/LES.2014.2371494>.
- [Chu11] Leon CHUA. « Resistance switching memories are memristors ». In : *Applied Physics A* 102.4 (jan. 2011), 765–783. URL : <http://dx.doi.org/10.1007/s00339-011-6264-9>.
- [Chu71] L. CHUA. « Memristor-The missing circuit element ». In : *IEEE Transactions on Circuit Theory* 18.5 (sept. 1971), p. 507-519.
- [CIDYP07] Doosan CHO, Ilya ISSENIN, Nikil DUTT, Jonghee W YOON et Yunheung PAEK. « Software controlled memory layout reorganization for irregular array access patterns ». In : *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2007, p. 179-188.
- [Con+] SOCLIB CONSORTIUM et al. *Projet SOCLIB : Plate-forme de modélisation et de simulation de systèmes intégrés sur puce (SOCLIB project : An integrated system-on-chip modelling and simulation platform)* Technical report, CNRS, 2003.
- [CPIDAP09] Doosan CHO, Sudeep PASRICHA, Ilya ISSENIN, Nikil D DUTT, Minwook AHN et Yunheung PAEK. « Adaptive scratch pad memory management for dynamic behavior of multimedia applications ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.4 (2009), p. 554-567.
- [DCANK17] Matthias DIENER, Eduardo HM CRUZ, Marco AZ ALVES, Philippe OA NAVAUX et Israel KOREN. « Affinity-based thread and data mapping in shared memory systems ». In : *ACM Computing Surveys (CSUR)* 49.4 (2017), p. 64.
- [DNB07] Angel DOMINGUEZ, Nghi NGUYEN et Rajeev K BARUA. « Recursive function data allocation to scratch-pad memory ». In : *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2007, p. 65-74.
- [DSCM+15] Atienza DAVID, Mamagkakis STYLIANOS, Poucet CHRISTOPHE, Peón-Quirós MIGUEL, Bartzas ALEXANDROS, Catthoor FRANCKY et Soudris DIMITRIOS. *Dynamic Memory Management for Embedded Systems*. Springer International Publishing, 2015.
- [DUB05] Angel DOMINGUEZ, Sumesh UDAYAKUMARAN et Rajeev BARUA. « Heap data allocation to scratch-pad memory in embedded systems ». In : *Journal of Embedded Computing* 1.4 (2005), p. 521-540.

- [DXXJ12] Xiangyu DONG, Cong XU, Yuan XIE et N. P. JOUPPI. « NVSim : A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory ». In : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.7 (juil. 2012), 994–1007. URL : <http://dx.doi.org/10.1109/TCAD.2012.2185930>.
- [EKJLMS10] Bernhard EGGER, Seungkyun KIM, Choonki JANG, Jaejin LEE, Sang Lyul MIN et Heonshik SHIN. « Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU ». In : *IEEE Transactions on Computers* 59.8 (août 2010), 1047–1062. URL : <http://dx.doi.org/10.1109/TC.2009.188>.
- [EKJNLM06] Bernhard EGGER, Chihun KIM, Choonki JANG, Yoonsung NAM, Jaejin LEE et Sang Lyul MIN. « A dynamic code placement technique for scratchpad memory using postpass optimization ». In : *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM. 2006, p. 223-233.
- [ELS08] Bernhard EGGER, Jaejin LEE et Heonshik SHIN. « Dynamic scratchpad memory management for code in portable systems with an MMU ». In : *ACM Transactions on Embedded Computing Systems* 7.2 (fév. 2008), 1–38. URL : <http://dx.doi.org/10.1145/1331331.1331335>.
- [FCW16] Zhen FAN, Jingsheng CHEN et John WANG. « Ferroelectric HfO₂-based materials for next-generation ferroelectric memories ». In : *Journal of Advanced Dielectrics* 06.02 (juin 2016), p. 1630003. URL : <http://dx.doi.org/10.1142/S2010135X16300036>.
- [FMABCM04] Poletti FRANCESCO, Paul MARCHAL, David ATIENZA, Luca BENINI, Francky CATTHOOR et Jose M MEDIAS. « An integrated hardware/software approach for run-time scratchpad management ». In : *Proceedings of the 41st annual Design Automation Conference*. ACM. 2004, p. 238-243.
- [Fou18] Global FOUNDRIES. *Embedded Memory : eMRAM, eFlash, SIP*. <https://www.globalfoundries.com/sites/default/files/product-briefs/pb-emem.pdf>. 2018.
- [FSTW09] Jason E FRITTS, Frederick W STEILING, Joseph A TUCEK et Wayne WOLF. « MediaBench II video : Expediting the next generation of video systems research ». In : *Microprocessors and Microsystems* 33.4 (2009), p. 301-318.
- [Gab12] Krzysztof GABIS. *Parson : Lightweight JSON library written in C*. <https://github.com/kgabis/parson>. 2012.
- [Gel10] Rich GELDREICH. *picojpeg*. <https://code.google.com/archive/p/picojpeg>. 2010.
- [Gli] *The GNU C Library*. <https://www.gnu.org/software/libc/>.
- [GNWS+19] Abdoulaye GAMATIÉ, Alejandro NOCUA, Joel WELOLI, Gilles SASSATELLI, Lionel TORRES, David NOVO et Michel ROBERT. « Emerging NVM Technologies in Main Memory for Energy-Efficient HPC : an Empirical Study ». In : (2019).
- [GREAMB01] M. R. GUTHAUS, J. S. RINGENBERG, D. ERNST, T. M. AUSTIN, T. MUDGE et R. B. BROWN. « 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.
- [GYW12] Ximeng GUAN, Shimeng YU et H.-S. Philip WONG. « On the Switching Parameter Variation of Metal-Oxide RRAM ;Part I : Physical Modeling and Simulation Methodology ». In : *IEEE Transactions on Electron Devices* 59.4 (avr. 2012), 1172–1182. URL : <http://dx.doi.org/10.1109/TED.2012.2184545>.

- [HXZTS11] Jingtong HU, Chun Jason XUE, Qingfeng ZHUGE, Wei-Che TSENG et Edwin H-M SHA. « Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory ». In : *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE. 2011, p. 1-6.
- [HXZTS13] Jingtong HU, Chun Jason XUE, Qingfeng ZHUGE, Wei-Che TSENG et Edwin H.-M. SHA. « Data Allocation Optimization for Hybrid Scratch Pad Memory With SRAM and Nonvolatile Memory ». In : *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.6 (juin 2013), 1094–1102. URL : <http://dx.doi.org/10.1109/TVLSI.2012.2202700>.
- [HZXTS14] Jingtong HU, Qingfeng ZHUGE, Chun Jason XUE, Wei-Che TSENG et Edwin H.-M. SHA. « Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors ». In : *ACM Transactions on Embedded Computing Systems* 13.4 (mar. 2014), 1–25. URL : <http://dx.doi.org/10.1145/2560019>.
- [Ins14] Texas INSTRUMENTS. « MSP430 FRAM Technology-How To and Best Practices ». In : *Application Note, SLAA628* (2014).
- [JASL+19] Pulkit JAIN, Umut ARSLAN, Meenakshi SEKHAR, Blake C LIN, Liqiong WEI, Tanaya SAHU, Juan ALZATE-VINASCO, Ajay VANGAPATY, Mesut METEREL-LIYOZ, Nathan STRUTT et al. « 13.2 A 3.6 Mb 10.1 Mb/mm² Embedded Non-Volatile ReRAM Macro in 22nm FinFET Technology with Adaptive Forming/Set/Reset Schemes Yielding Down to 0.5 V with Sensing Time of 5ns at 0.7 V ». In : *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE. 2019, p. 212-214.
- [JLLRKR14] H. JAYAKUMAR, K. LEE, W. S. LEE, A. RAHA, Y. KIM et V. RAGHUNATHAN. « Powering the Internet of Things ». In : *Low Power Electronics and Design (ISLPED), 2014 IEEE/ACM International Symposium on*. Août 2014, p. 375-380.
- [KGGS18] Sudarsun KANNAN, Ada GAVRILOVSKA, Vishal GUPTA et Karsten SCHWAN. « HeteroOS : OS Design for Heterogeneous Memory Management in Data-centers ». In : *SIGOPS Oper. Syst. Rev.* 52.1 (août 2018), p. 13-26. URL : <http://doi.acm.org/10.1145/3273982.3273985>.
- [KICK04] Mahmut KANDEMIR, Mary Jane IRWIN, Guilin CHEN et Ibrahim KOLCU. « Banked scratch-pad memory management for reducing leakage energy consumption ». In : *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004*. IEEE. 2004, p. 120-124.
- [KKS17] Constantinos KOLIAS, Georgios KAMBOURAKIS, Angelos STAVROU et Jeffrey VOAS. « DDoS in the IoT : Mirai and other botnets ». In : *Computer* 50.7 (2017), p. 80-84.
- [KRIVKP01] Mahmut KANDEMIR, J RAMANUJAM, Mary Jane IRWIN, Narayanan VIJAY-KRISHNAN, Ismail KADAYIF et Amisha PARIKH. « Dynamic management of scratch-pad memory space ». In : *Design Automation Conference, 2001. Proceedings*. IEEE. 2001, p. 690-695.
- [LG96] Doug LEA et Wolfram GLOGER. *A memory allocator*. 1996.
- [LIMB09] Benjamin C. LEE, Engin IPEK, Onur MUTLU et Doug BURGER. « Architecting Phase Change Memory As a Scalable Dram Alternative ». In : *SIGARCH Comput. Archit. News* 37.3 (juin 2009), p. 2-13. URL : <http://doi.acm.org/10.1145/1555815.1555758>.
- [LJBGD+16] Christophe LAYER, Virgile JAVERLIAC, Fabrice BERNARD-GRANGER, Loic DECLOEDT et al. « Reducing System Power Consumption Using Check-Pointing on Nonvolatile Embedded Magnetic Random Access Memories ». In : *ACM Journal on Emerging Technologies in Computing Systems* 12.4 (mai 2016), 1-24. URL : <http://dx.doi.org/10.1145/2876507>.

- [LLCS19] J. LIN, J. LU, J. CAI et A. SHRIVASTAVA. « Efficient Heap Data Management on Software Managed Manycore Architectures ». In : *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*. Jan. 2019, p. 269-274.
- [LZH+12] Qingan LI, Yingchao ZHAO, Jingtong HU, Chun Jason XUE, Edwin SHA, Yanxiang HE, IDepartment of COMPUTER SCIENCE, City University of HONG KONG et Hong KONG. « MGC : Multiple Graph-Coloring for Non-Volatile Memory Based Hybrid Scratchpad Memory ». In : *Proceedings of the 2012 16th Workshop on • IEEE Interaction between Compilers and Computer Architectures (ijcomputer)*. IEEE, 2012.
- [MDS08] Ross McILROY, Peter DICKMAN et Joe SVENTEK. « Efficient dynamic heap allocation of scratch-pad memory ». In : *Proceedings of the 7th international symposium on Memory management*. ACM. 2008, p. 31-40.
- [MF11] Tiago Rogério MÜCK et Antônio Augusto FRÖHLICH. « Run-time scratch-pad memory management for embedded systems ». In : *IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society*. IEEE. 2011, p. 2833-2838.
- [MSCT14] Jagan Singh MEENA, Simon Min SZE, Umesh CHAND et Tseung-Yuen TSENG. « Overview of emerging nonvolatile memory technologies ». In : *Nanoscale research letters* 9.1 (2014), p. 526.
- [New] *Newlib C Library*. <https://www.gnu.org/software/libc/>.
- [NZANC18] A. NARAYAN, T. ZHANG, S. AGA, S. NARAYANASAMY et A. COSKUN. « MOCA : Memory Object Classification and Allocation in Heterogeneous Memory Systems ». In : *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Mai 2018, p. 326-335.
- [PJVP+16] Guillaume PRENAT, Kotb JABEUR, Pierre VANHAUWAERT, Gregory Di PENDINA et al. « Ultra-Fast and High-Reliability SOT-MRAM : From Cache Replacement to Normally-Off Computing ». In : *IEEE Transactions on Multi-Scale Computing Systems* 2.1 (jan. 2016), 49–60. URL : <http://dx.doi.org/10.1109/TMSCS.2015.2509963>.
- [RKMH+10] ROSENDALE, KIANIAN, MANNING, HAMILTON, HUANG, ROBINSON, KIM et T. RUECKES. « A 4 Megabit Carbon Nanotube-based nonvolatile memory (NRAM) ». In : *2010 Proceedings of ESSCIRC*. Sept. 2010, p. 478-481.
- [RSF11] Benjamin RANSFORD, Jacob SORBER et Kevin FU. « Mementos : System Support for Long-running Computation on RFID-scale Devices ». In : *SIGARCH Comput. Archit. News* 39.1 (mar. 2011), p. 159-170. URL : <http://doi.acm.org/10.1145/1961295.1950386>.
- [RTK14] Gabriel RODRÍGUEZ, Juan TOURIÑO et Mahmut T. KANDEMIR. « Volatile STT-RAM Scratchpad Design and Data Allocation for Low Energy ». In : *ACM Transactions on Architecture and Code Optimization* 11.4 (déc. 2014), 1–26. URL : <http://dx.doi.org/10.1145/2669556>.
- [SKL09] Aviral SHRIVASTAVA, Arun KANNAN et Jongeun LEE. « A software-only solution to use scratch pads for stack data ». In : *IEEE Transactions on computer-aided design of integrated circuits and systems* 28.11 (2009), p. 1719-1727.
- [SKR16] Yeongkyo SEO, Kon-Woo KWON et Kaushik ROY. « Area-Efficient SOT-MRAM With a Schottky Diode ». In : *IEEE Electron Device Letters* 37.8 (août 2016), 982–985. URL : <http://dx.doi.org/10.1109/LED.2016.2578959>.
- [STSG16] Sophiane SENNI, Lionel TORRES, Gilles SASSATELLI et Abdoulaye GAMATIE. « Non-Volatile Processor Based on MRAM for Ultra-Low-Power IoT Devices ». In : *J. Emerg. Technol. Comput. Syst.* 13.2 (déc. 2016), 17 :1-17 :23. URL : <http://doi.acm.org/10.1145/3001936>.

- [SWVC+15] Dipanjan SENGUPTA, Qi WANG, Haris VOLOS, Ludmila CHERKASOVA, Jun LI, Guilherme MAGALHAES et Karsten SCHWAN. « A framework for emulating non-volatile memory systems with different performance characteristics ». In : *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM. 2015, p. 317-320.
- [Sys] *SystemC*. <https://www.accellera.org/downloads/standards/systemc/>.
- [TMAJ08] Shyamkumar THOZIYOOR, Naveen MURALIMOHAR, Jung Ho AHN et Norman P JOUPPI. *CACTI 5.1*. Rapp. tech. Technical Report HPL-2008-20, HP Labs, 2008.
- [TOTK+14] N TAKAURA, T OHYANAGI, M TAI, M KINOSHITA, K AKITA, T MORIKAWA, H SHIRAKAWA, M ARAIDAI, K SHIRAISHI, Y SAITO et al. « 55- μ A Ge x Te 1-x/Sb 2 Te 3 superlattice topological-switching random access memory (TRAM) and study of atomic arrangement in Ge-Te and Sb-Te structures ». In : *2014 IEEE International Electron Devices Meeting*. IEEE. 2014, p. 29-2.
- [TOTMKA14] N. TAKAURA, T. OHYANAGI, M. TAI, T. MORIKAWA, M. KINOSHITA et K. AKITA. « Fabrication of topological-switching RAM (TRAM) ». In : *2014 14th Annual Non-Volatile Memory Technology Symposium (NVMTS)*. Oct. 2014, p. 1-4.
- [VEPAG12] Brian VAN ESSEN, Roger PEARCE, Sasha AMES et Maya GOKHALE. « On the Role of NVRAM in Data-intensive Architectures : An Evaluation ». In : *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (mai 2012). URL : <http://dx.doi.org/10.1109/IPDPS.2012.69>.
- [VWM04] Manish VERMA, Lars WEHMEYER et Peter MARWEDEL. « Dynamic overlay of scratchpad memory for energy minimization ». In : *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM. 2004, p. 104-109.
- [WCLL18] Shasha WEN, Lucy CHERKASOVA, Felix Xiaozhu LIN et Xu LIU. « Profdp : A lightweight profiler to guide data placement in heterogeneous memory systems ». In : *Proceedings of the 2018 International Conference on Supercomputing*. ACM. 2018, p. 263-273.
- [Wil08] R. Stanley WILLIAMS. *How we found the missing memristor*. Rapp. tech. HP Labs, 2008.
- [WIL17] Sean WILLIAMS, Latchesar IONKOV et Michael LANG. « NUMA Distance for Heterogeneous Memory ». In : *Proceedings of the Workshop on Memory Centric Programming for HPC*. ACM. 2017, p. 30-34.
- [WILL18] Sean WILLIAMS, Latchesar IONKOV, Michael LANG et Jason LEE. « Heterogeneous Memory and Arena-Based Heap Allocation ». In : *Proceedings of the Workshop on Memory Centric High Performance Computing*. ACM. 2018, p. 67-71.
- [WJNB95] Paul R WILSON, Mark S JOHNSTONE, Michael NEELY et David BOLES. « Dynamic storage allocation : A survey and critical review ». In : *Memory Management*. Springer, 1995, p. 1-116.
- [WM95] Wm A WULF et Sally A MCKEE. « Hitting the memory wall : implications of the obvious ». In : *ACM SIGARCH computer architecture news* 23.1 (1995), p. 20-24.
- [YSS12] J. Joshua YANG, Dmitri B. STRUKOV et Duncan R. STEWART. « Memristive devices for computing ». In : *Nature Nanotechnology* 8.1 (déc. 2012), 13–24. URL : <http://dx.doi.org/10.1038/NNANO.2012.240>.
- [Zen06] Olivier ZENDRA. « Memory and compiler optimizations for low-power and-energy ». In : *arXiv preprint cs/0610028* (2006).

- [ZQCG18] Hui ZHAO, Meikang QIU, Min CHEN et Keke GAI. « Cost-aware optimal data allocations for multiple dimensional heterogeneous memories using dynamic programming in big data ». In : *Journal of computational science* 26 (2018), p. 402-408.

Troisième partie

Annexes

Annexe A

Description des jeux de données

Les tableaux ci-après contiennent les information d'exécution par application. Pour chaque jeu de données le tableau contient :

- "N" : numéro de désignation du jeu de donnée dans le document
- "File name" : nom du fichier contenant le jeu de données dans le simulateur
- "Size" : taille du jeu de données en octet
- "Footprint" : taille maximale occupée par le tas durant l'exécution de référence (à un seul tas)
- "Objs" : nombre d'objets alloués durant l'exécution de référence pour ce jeu de données
- "R/W" : ratio lecture sur écriture moyen pour les objet d'une exécution
- "Reuse" : ratio de réutilisation de la mémoire (somme de la taille des allocations / empreinte mémoire)
- "Ref time" : temps de l'exécution de référence, avec un seul tas composé de mémoire lente
- "Fast time" : temps de l'exécution idéale, avec un seul tas composé de mémoire rapide

Json parser

N	File name	Size	Footprint	Objs	R/W	Reuse	Ref time	Fast time
0	0_walking_dead_short.json	13041	38864	1637	2.0	1.4	8161492	6878609
1	1_increased_depth.json	18050	44688	1239	1.5	1.5	6163214	4767869
2	2_lorem_ipsum.json	19356	44952	634	1.6	1.5	6486132	4970090
3	3_minimal_episodes.json	2714	11744	634	2.1	1.3	2328975	1983302
4	4_more_objects.json	11183	41616	2221	2.3	1.3	9790569	8479883
5	5_more_things_in_ep.json	22204	52080	949	3.1	1.6	12274625	10362110
6	6_less_strings.json	45171	120232	6570	1.7	1.3	25275399	21588059
7	7_games_of_thrones_short.json	15544	45544	1870	2.0	1.4	9562264	8040647

Dijkstra

N	File name	Size	Footprint	Objs	R/W	Reuse	Ref time	Fast time
0	input_1.dat	7286	10072	14979	8.9	24.0	211346095	202959989
1	Input_2.dat	6478	13216	12492	11.5	15.3	192313868	184743782
2	Input_3.dat	7211	10744	13929	9.8	21.0	203650122	195501503
3	input_4.dat	7304	9904	13095	9.6	21.4	194820739	187052600
4	input_5.dat	7306	10432	14059	9.4	21.8	203744093	195616685
5	input_6.dat	7303	10072	12869	9.4	20.7	191727393	184114856
6	input_7.dat	10025	4888	2004	3.6	7.1	79721755	76360562
7	input_8.dat	5259	8824	6887	9.4	12.8	127948719	123591593

Ecdsa

N	Input name	Size	Footprint	Objs	R/W	Reuse	Ref time	Fast time
0	-	46	7280	14505	1.4	59.1	137677408	102850898
1	-	119	7480	14583	1.4	57.8	137880202	102974018
2	-	613	7328	14543	1.4	58.9	137923586	103073879
3	-	445	7544	14724	1.4	57.7	138885665	103699598
4	-	2691	7496	14673	1.4	57.9	138909873	103831112
5	-	17	7280	14810	1.4	60.1	139027298	103740926
6	-	178	7280	14746	1.4	59.9	138853056	103648619
7	-	260	7376	14743	1.4	59.1	138836265	103663913

NB : les jeux de données sont contenus dans le header de l'application.

Jpg 2000

N	File name	Size	Footprint	Objs	R/W	Reuse	Ref time	Fast time
0	input_0.pnm	76086	1704760	10256	3.0	1.2	1184926626	998994503
1	input_1.pnm	206310	3294536	10617	2.3	1.1	1666432432	1417486178
2	input_2.pnm	187560	2780424	10274	2.6	1.2	1887847298	1307888462
3	input_3.pnm	196668	2932040	10293	2.2	1.2	1890956966	1411337927
4	gradient.pnm	196668	2776480	10257	2.6	1.1	1484434466	1160536769
5	lena.pnm	67560	1627760	10254	3.1	1.2	1158908584	983097566
6	max_color.pnm	64860	1425784	10042	3.0	1.3	1180174345	996722615
7	vitrail.pnm	70860	1496256	10056	2.7	1.3	1294615467	1063990715

H263

N	File name	Size	Footprint	Objs	R/W	Reuse	Ref time	Fast time
0	input_0_i420_352x288.yuv	342144	1260744	53829	20.4	34.8	24780282313	23530385484
1	input_1_i420_176x144.yuv	57024	366200	7511	12.9	18.2	4354747092	4141059981
2	input_2_i420_176x144.yuv	294624	366224	43369	10.4	105.9	23309025053	22128066651
3	input_3_i420_352x288.yuv	456192	1417224	62904	27.3	42.1	42188028838	39624220506
4	input_4_i420_352x288.yuv	380160	1417208	51543	17.2	34.4	30032636602	28334126037
5	input_5_i420_176x144.yuv	475200	368040	70553	8.6	172.5	37151671026	35297586702
6	input_6_i420_176x144.yuv	380160	366224	57773	14.0	138.6	33788854284	32090500812
7	input_7_i420_352x288.yuv	456192	1417224	62744	24.7	42.0	40759078077	38314371270

Annexe B

Analyse des objets par exécution

Json parser

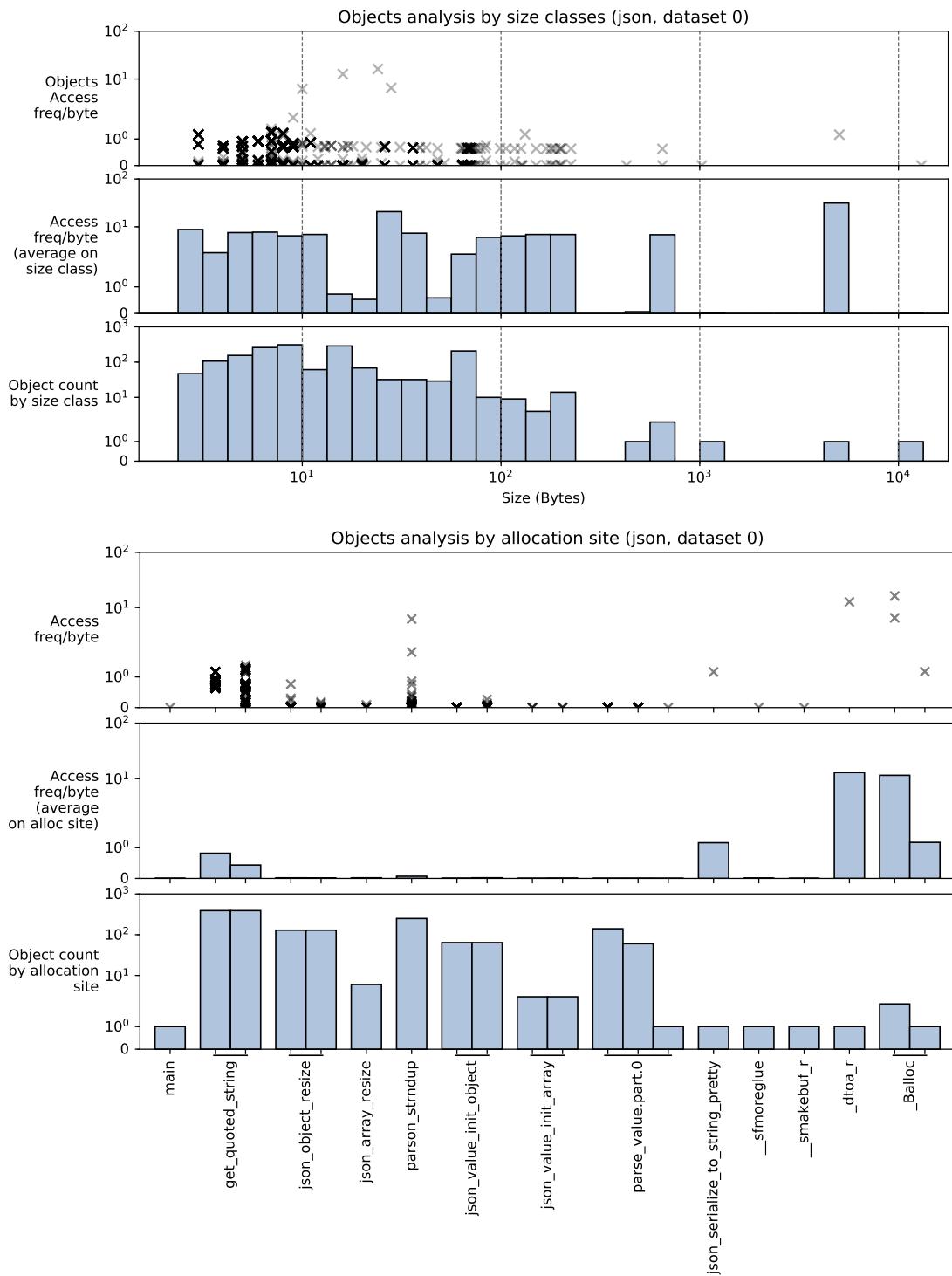


FIGURE B.1 – Json parser Jeu de données 0 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

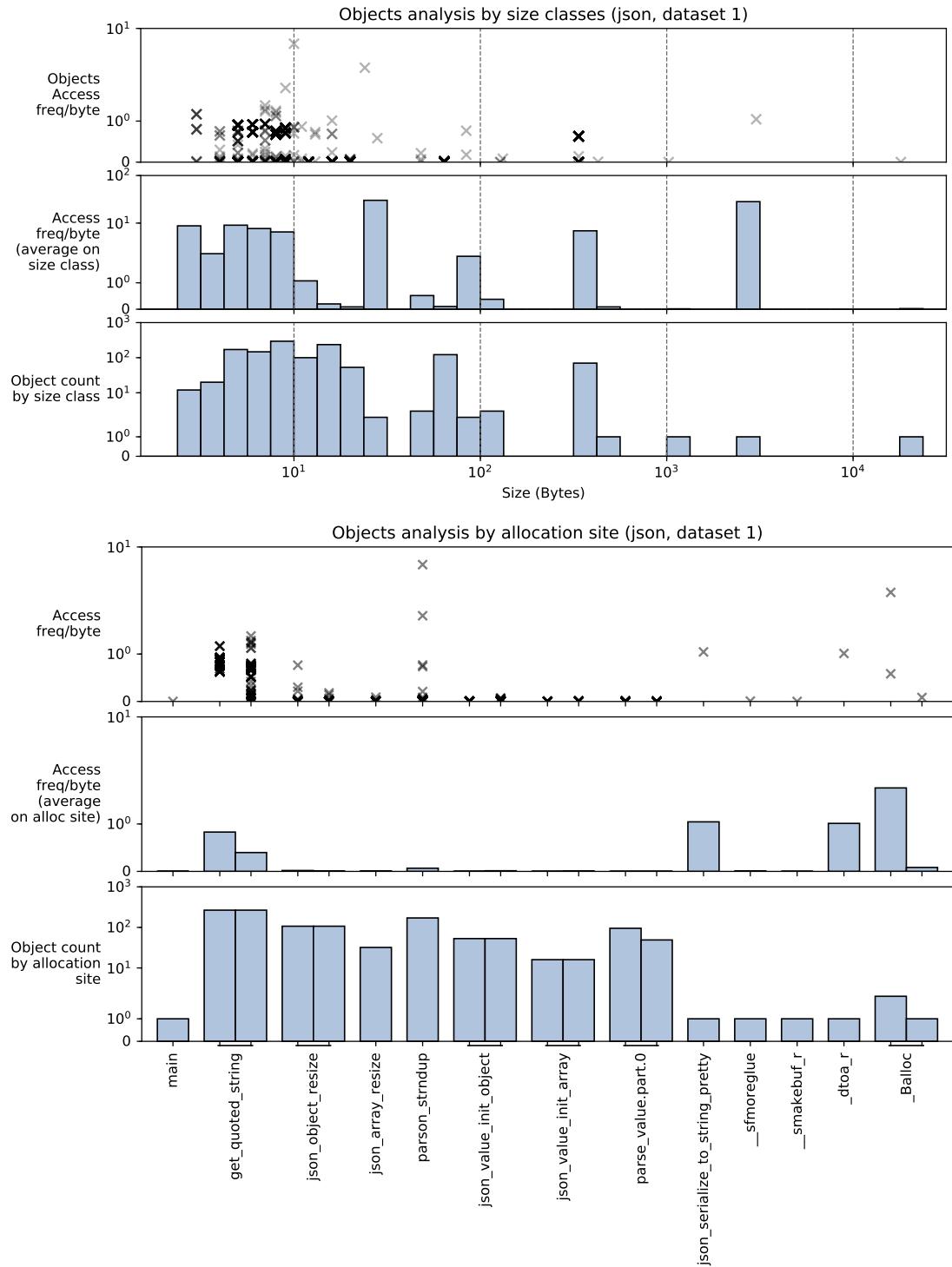


FIGURE B.2 – Json parser Jeu de données 1 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

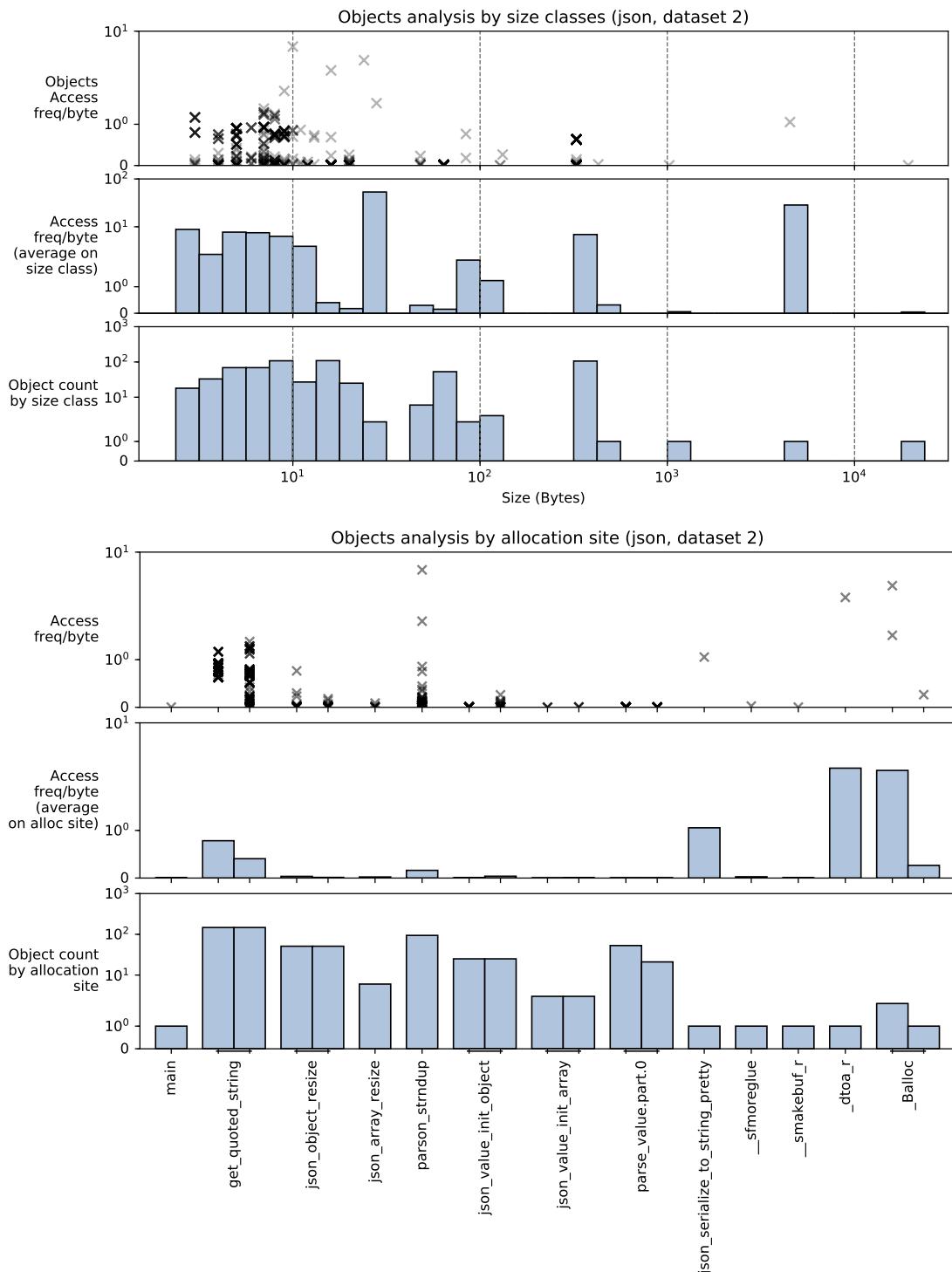


FIGURE B.3 – Json parser Jeu de données 2 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

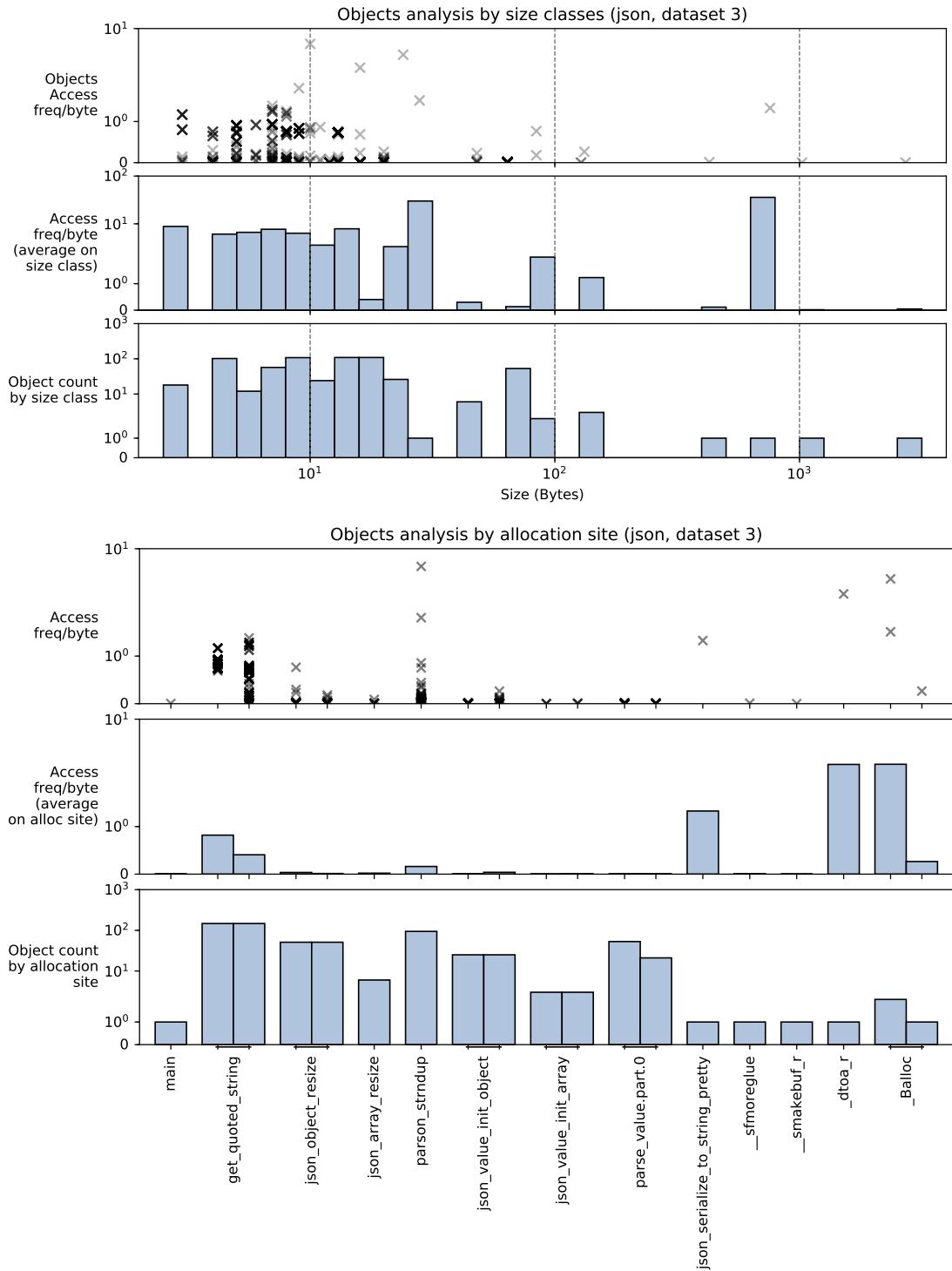


FIGURE B.4 – Json parser Jeu de données 3 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

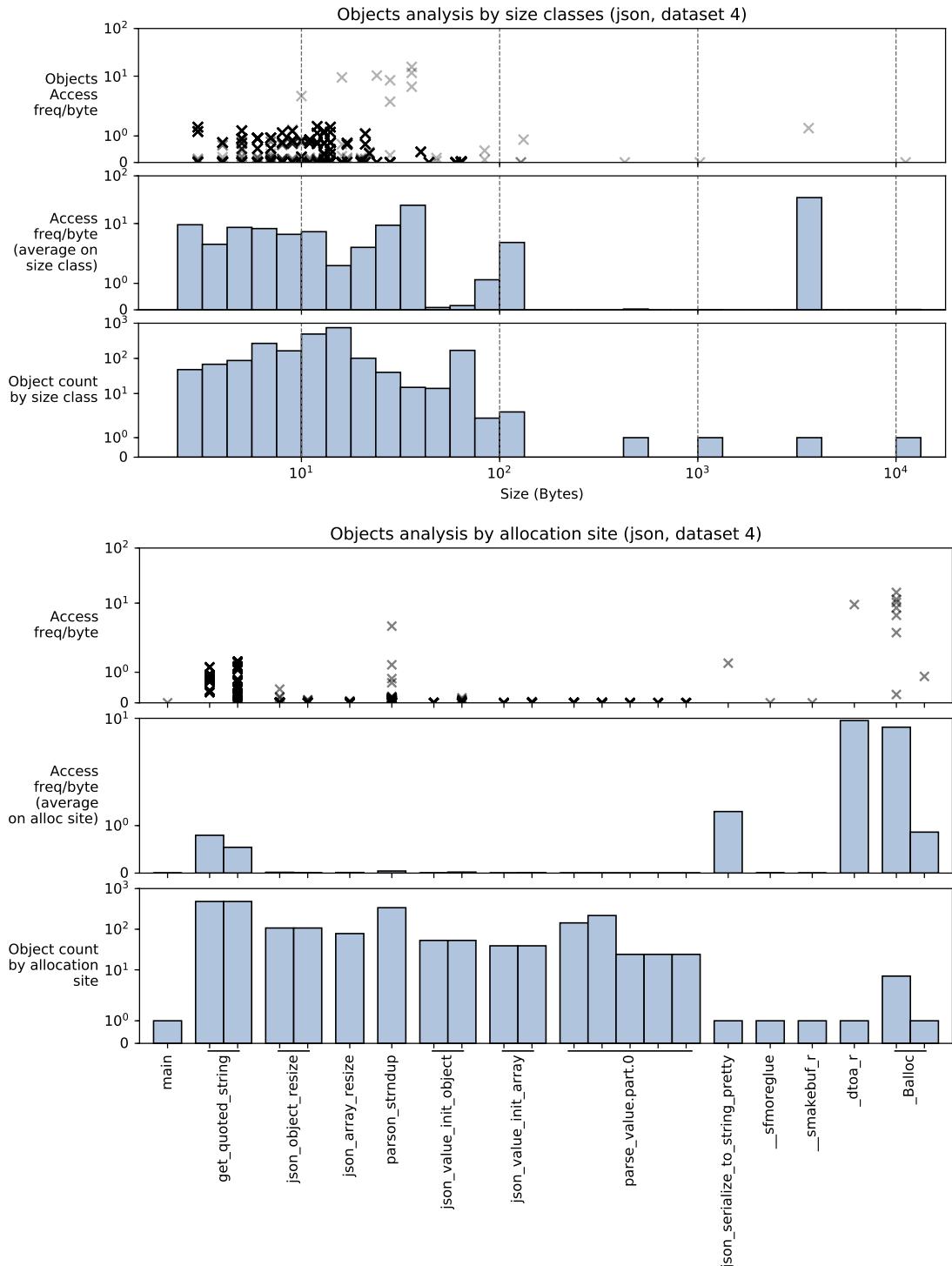


FIGURE B.5 – Json parser Jeu de données 4 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

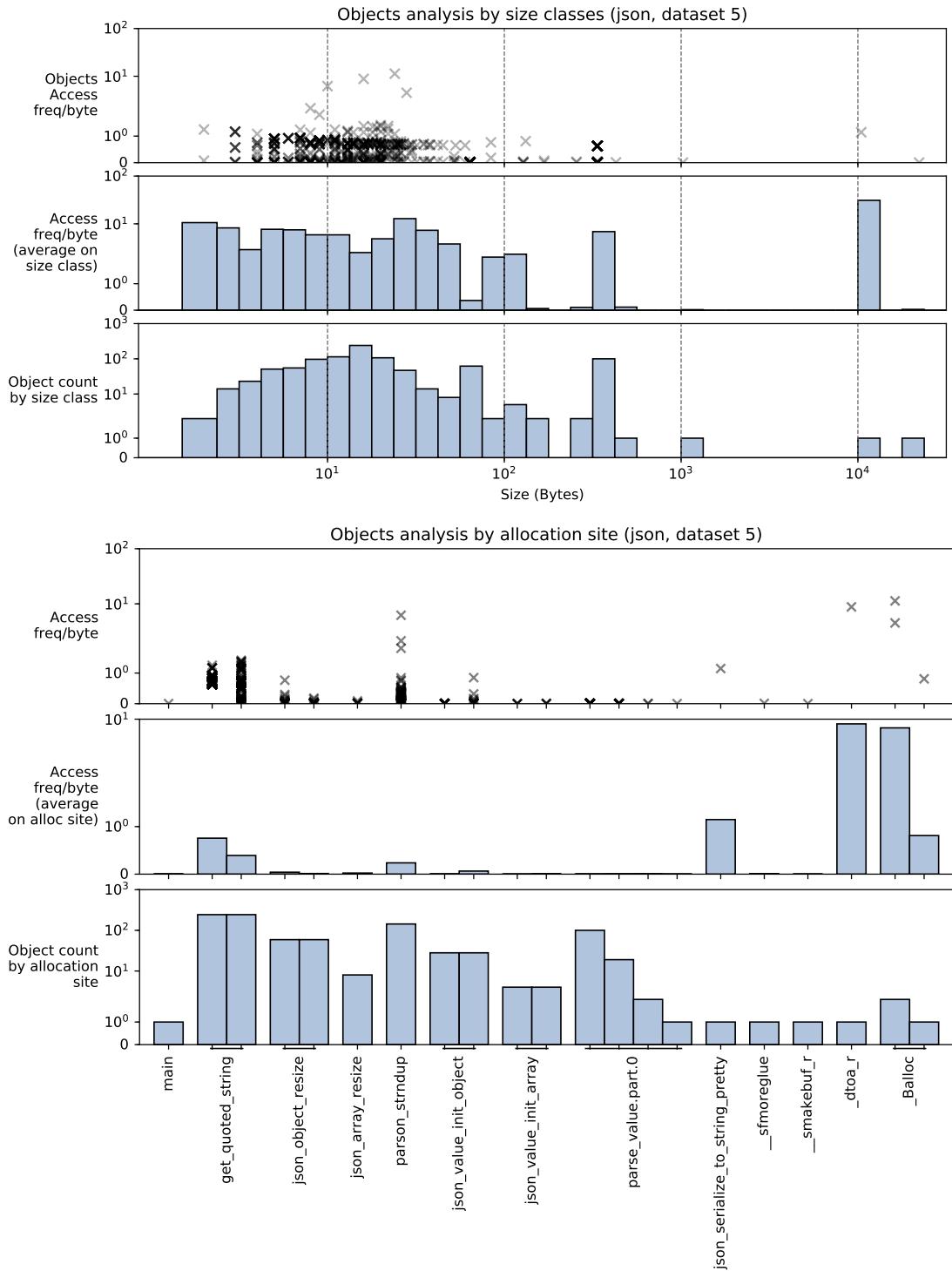


FIGURE B.6 – Json parser Jeu de données 5 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

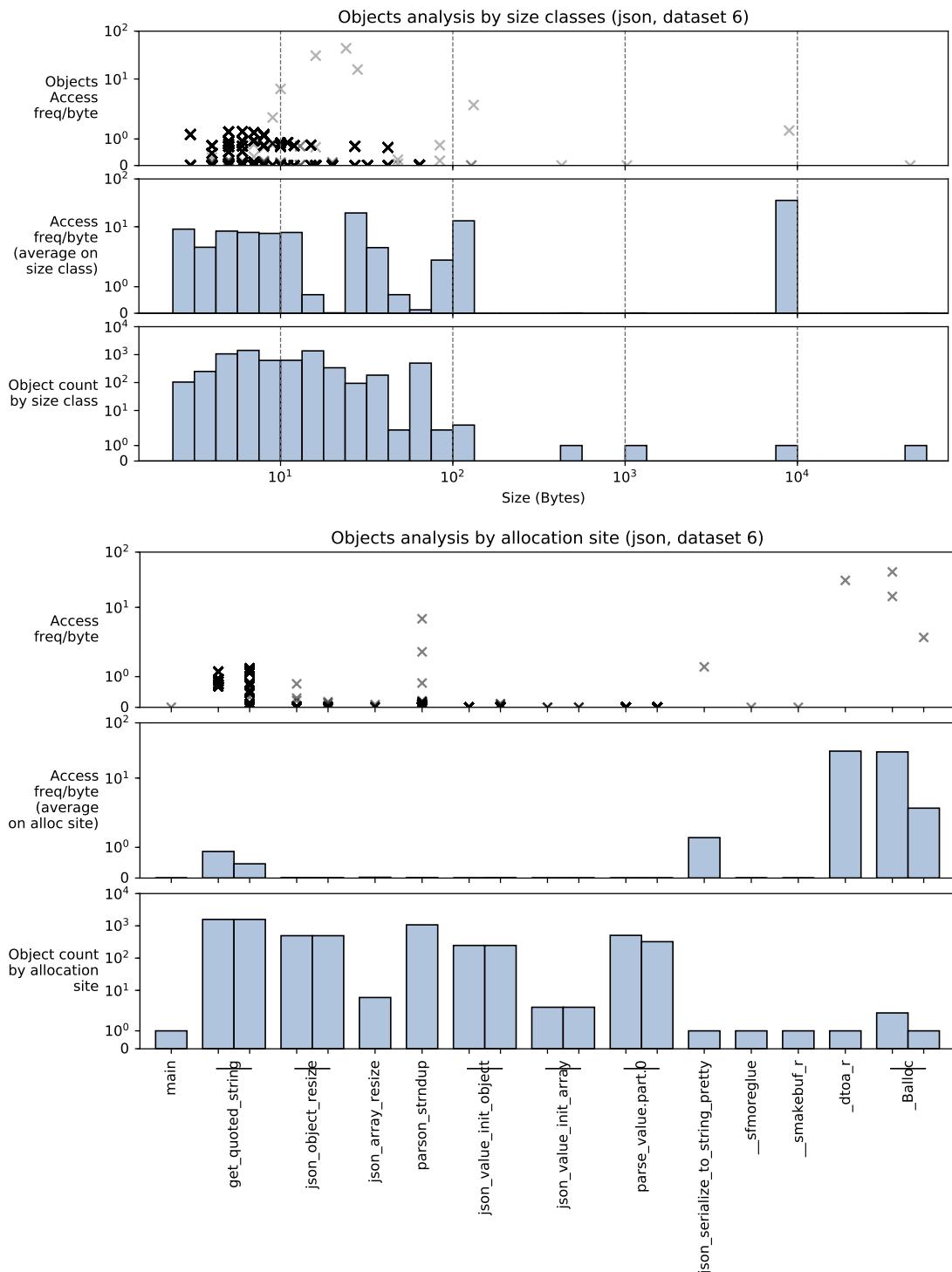


FIGURE B.7 – Json parser Jeu de données 6 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

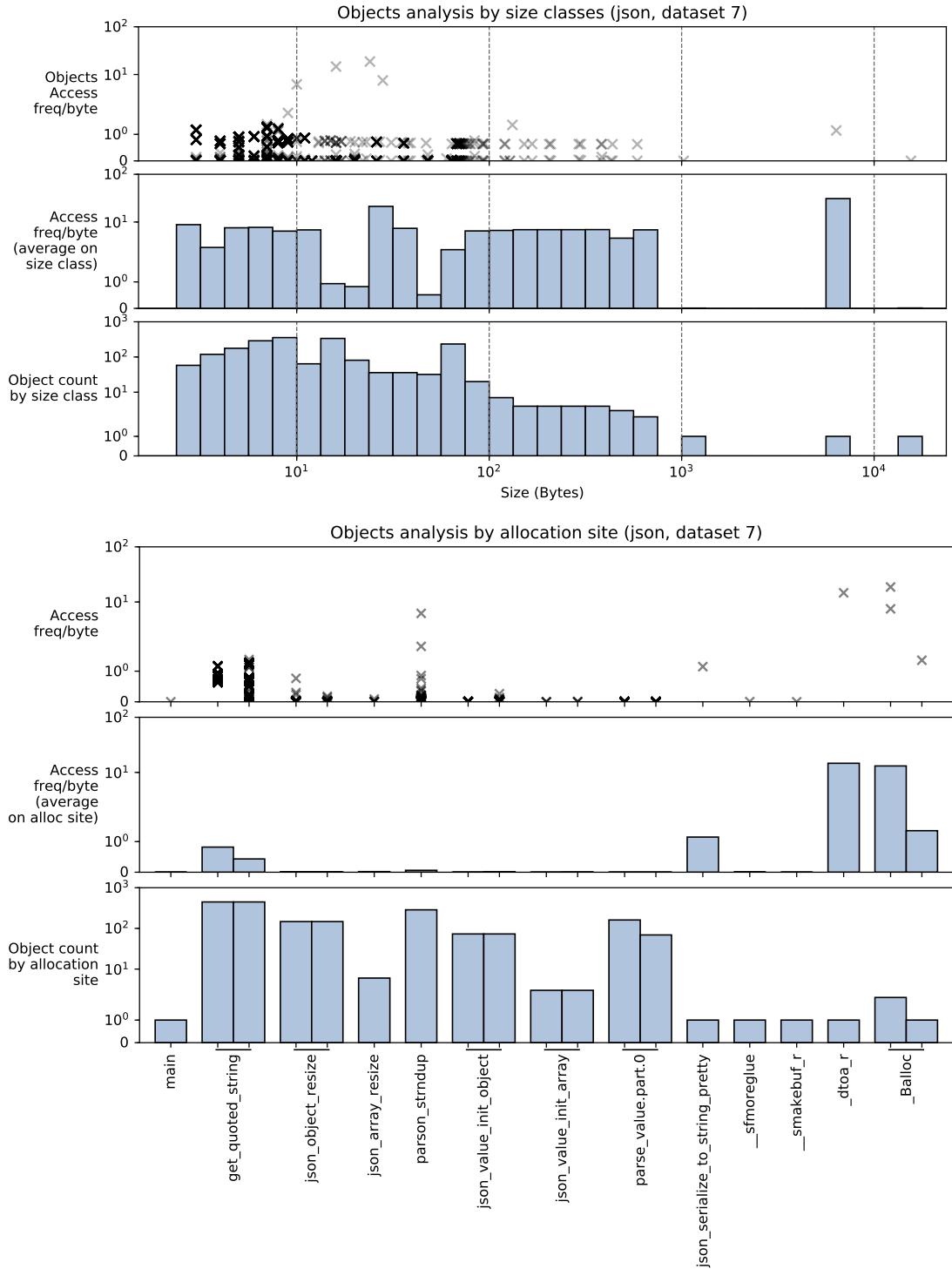


FIGURE B.8 – Json parser Jeu de données 7 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

Dijkstra

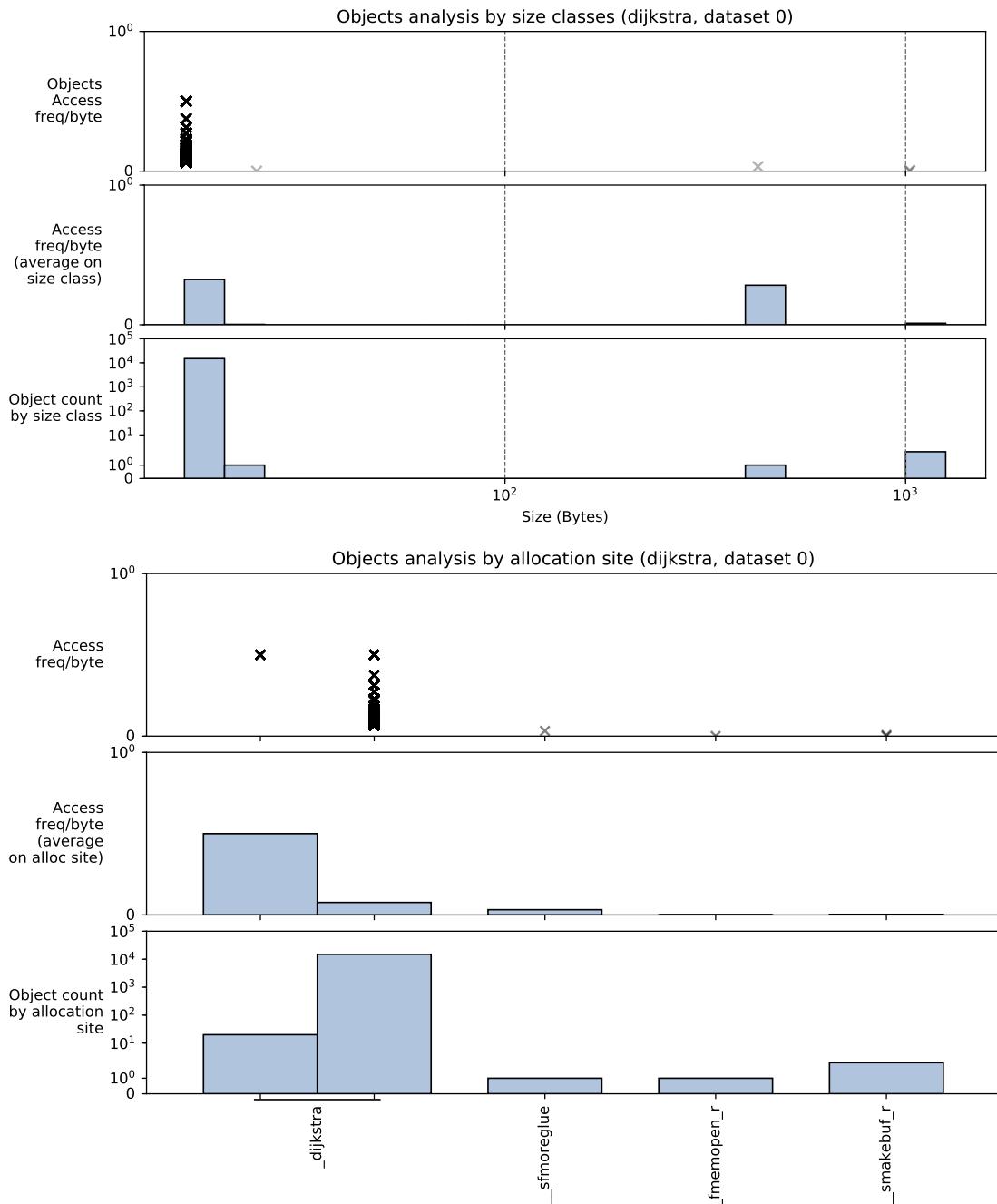


FIGURE B.9 – Dijkstra Jeu de données 0 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

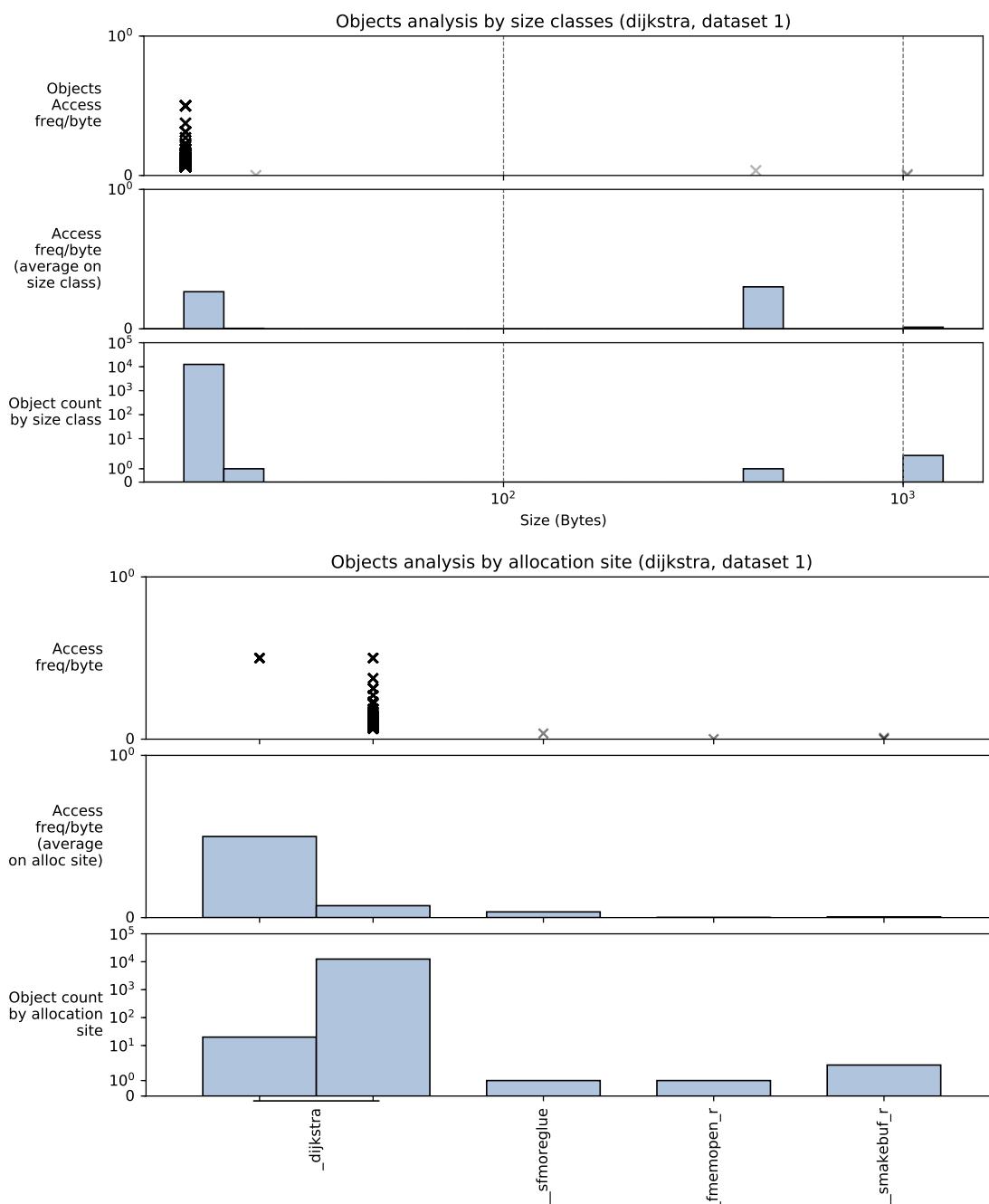


FIGURE B.10 – Dijkstra Jeu de données 1 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

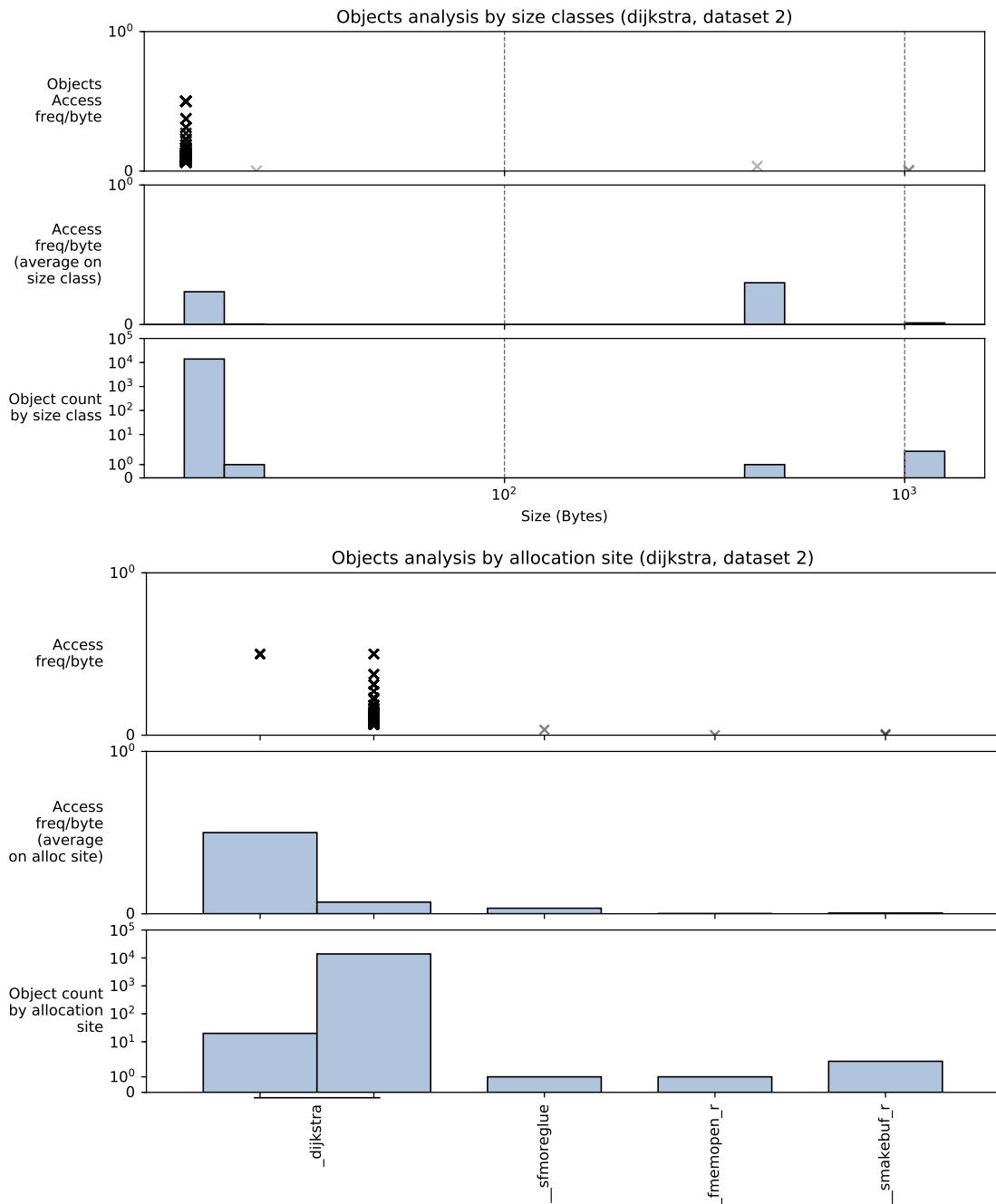


FIGURE B.11 – Dijkstra Jeu de données 2 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

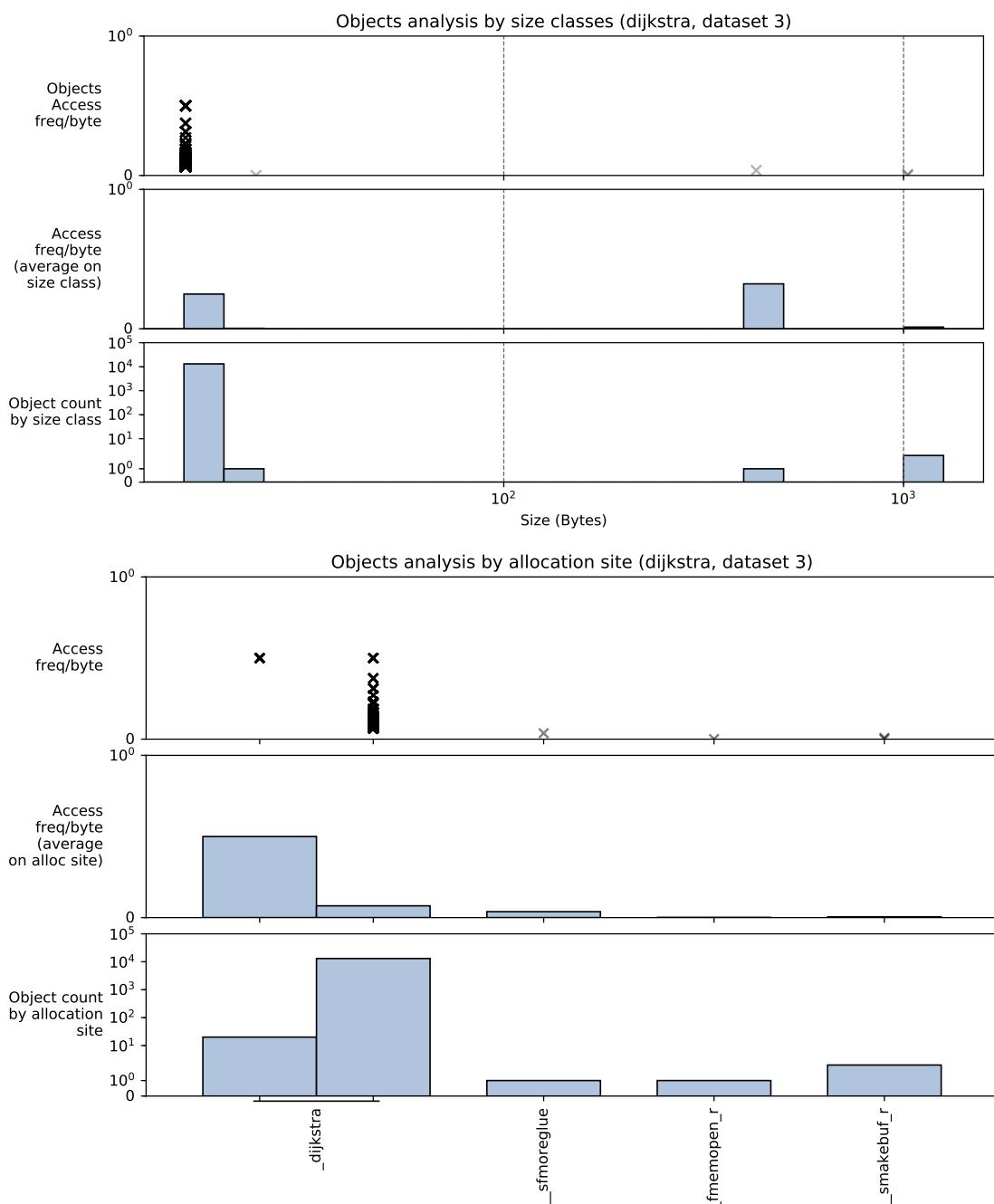


FIGURE B.12 – Dijkstra Jeu de données 3 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

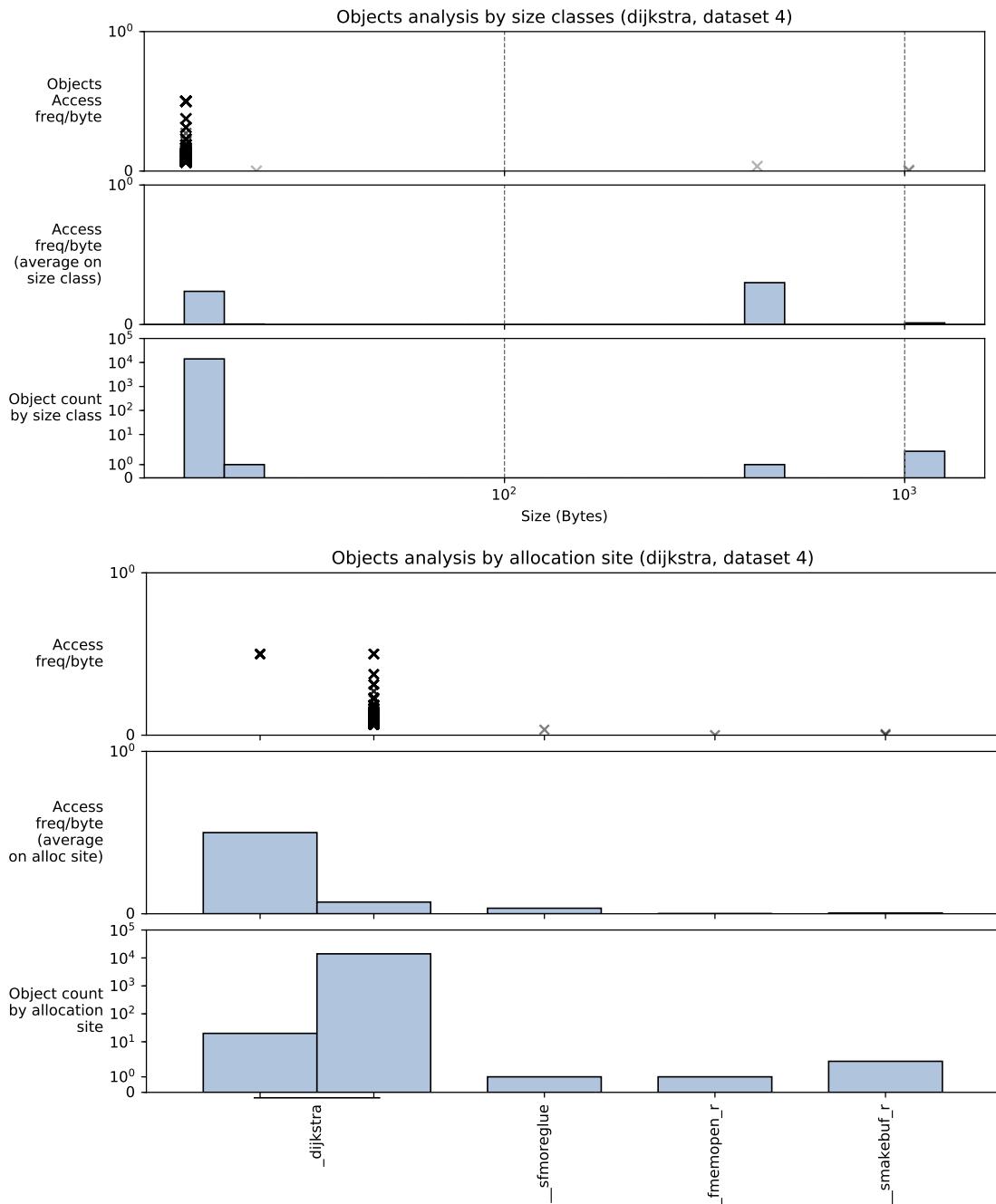


FIGURE B.13 – Dijkstra Jeu de données 4 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

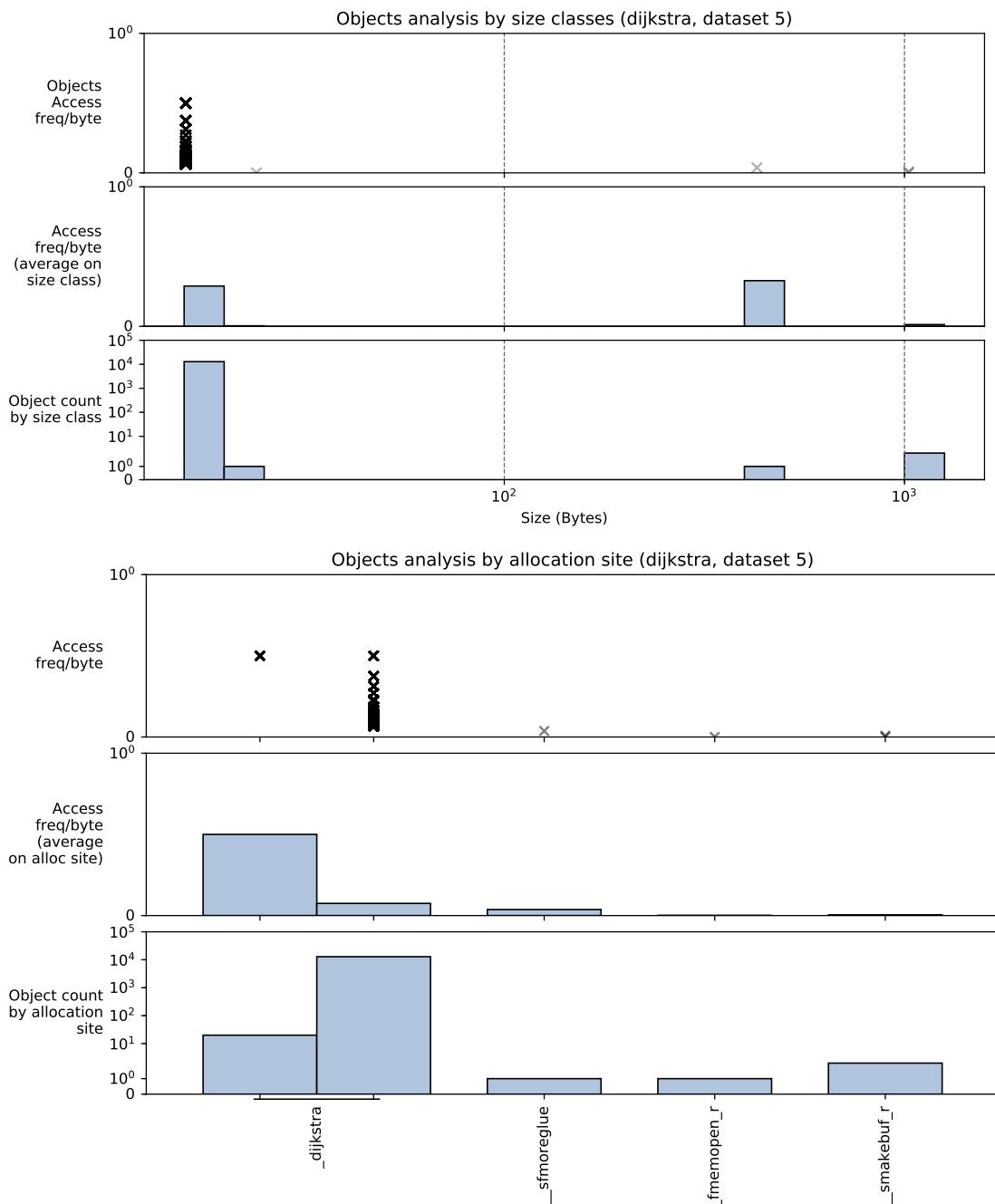


FIGURE B.14 – Dijkstra Jeu de données 5 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

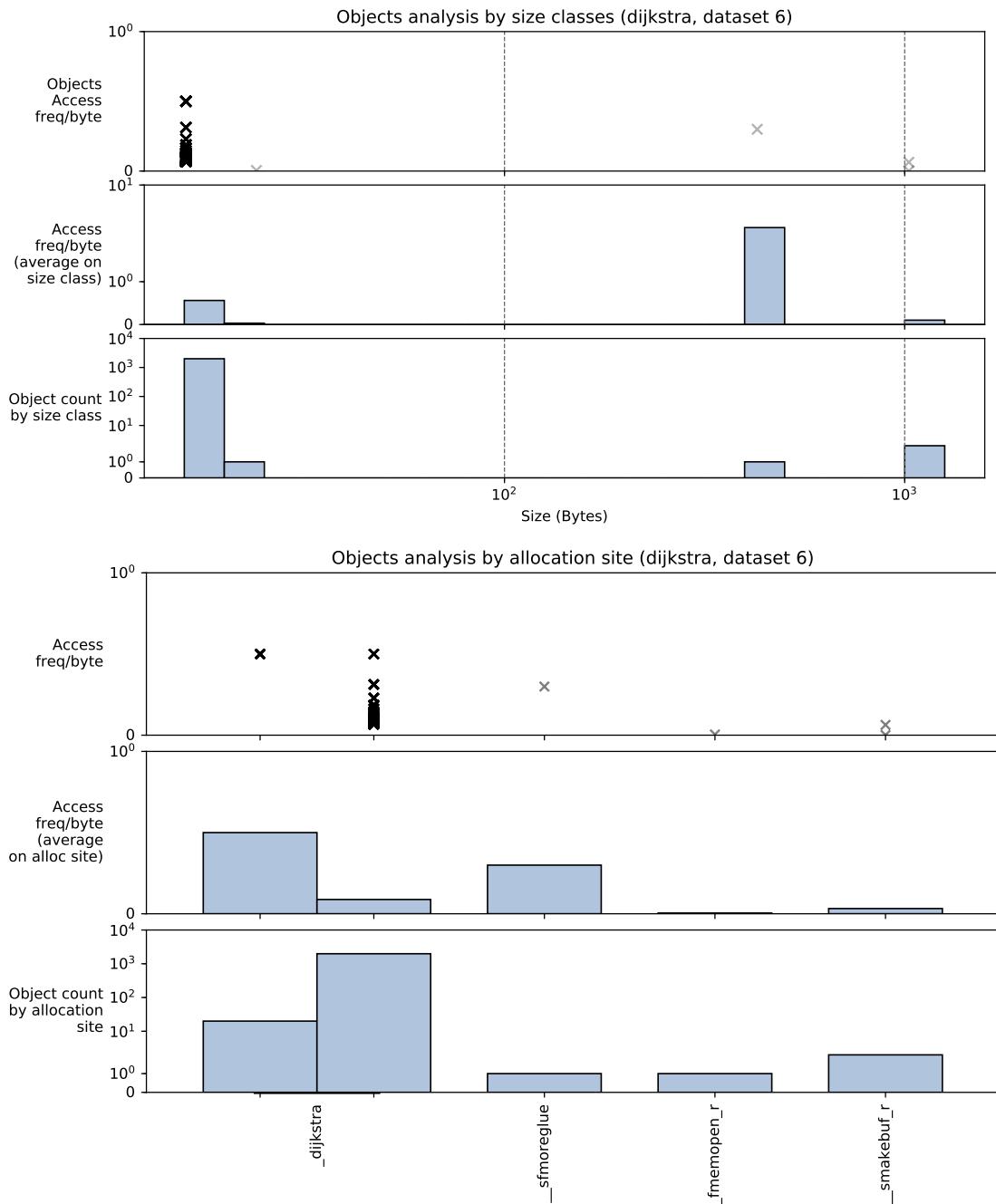


FIGURE B.15 – Dijkstra Jeu de données 6 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

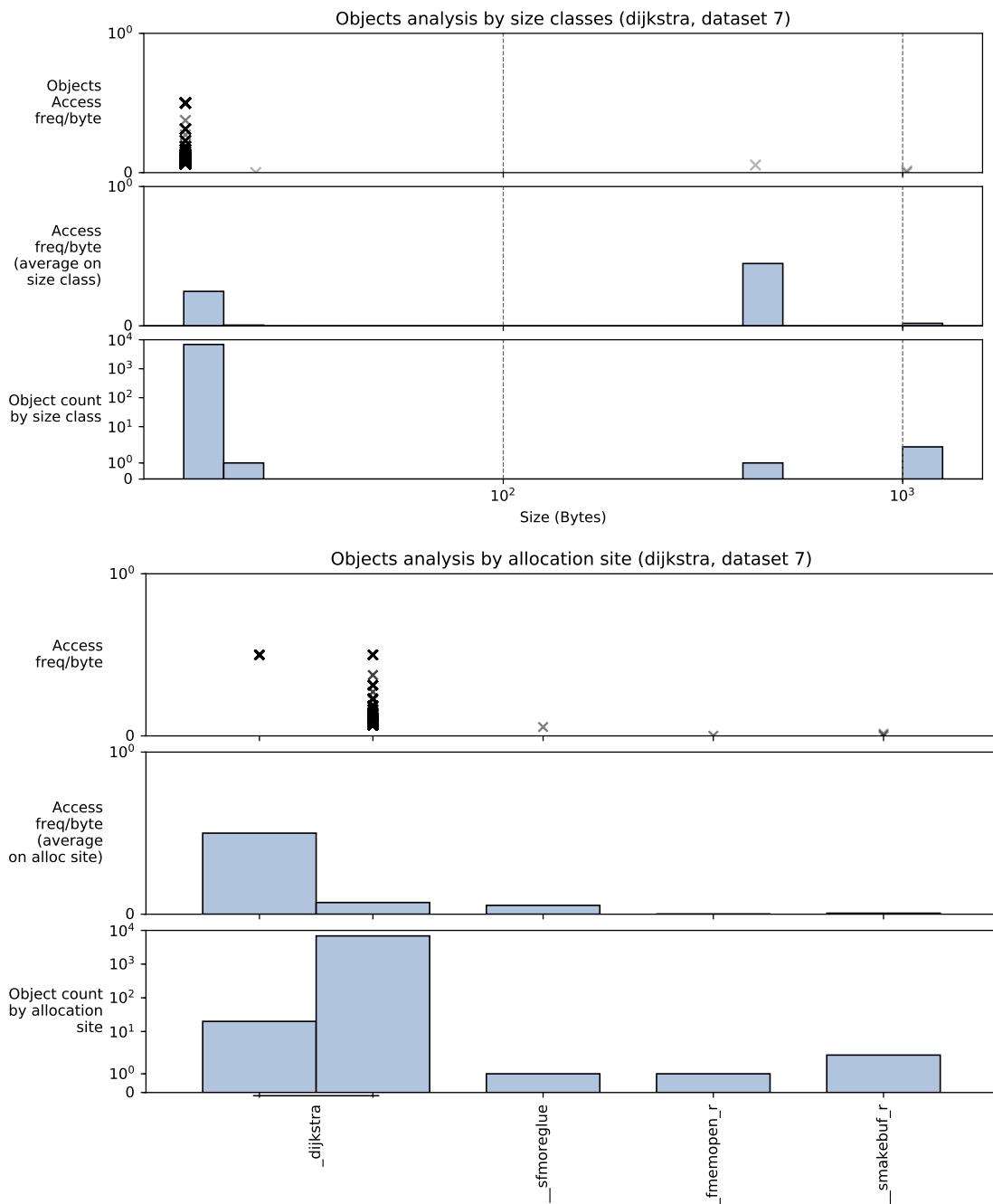


FIGURE B.16 – Dijkstra Jeu de données 7 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

Ecdsa

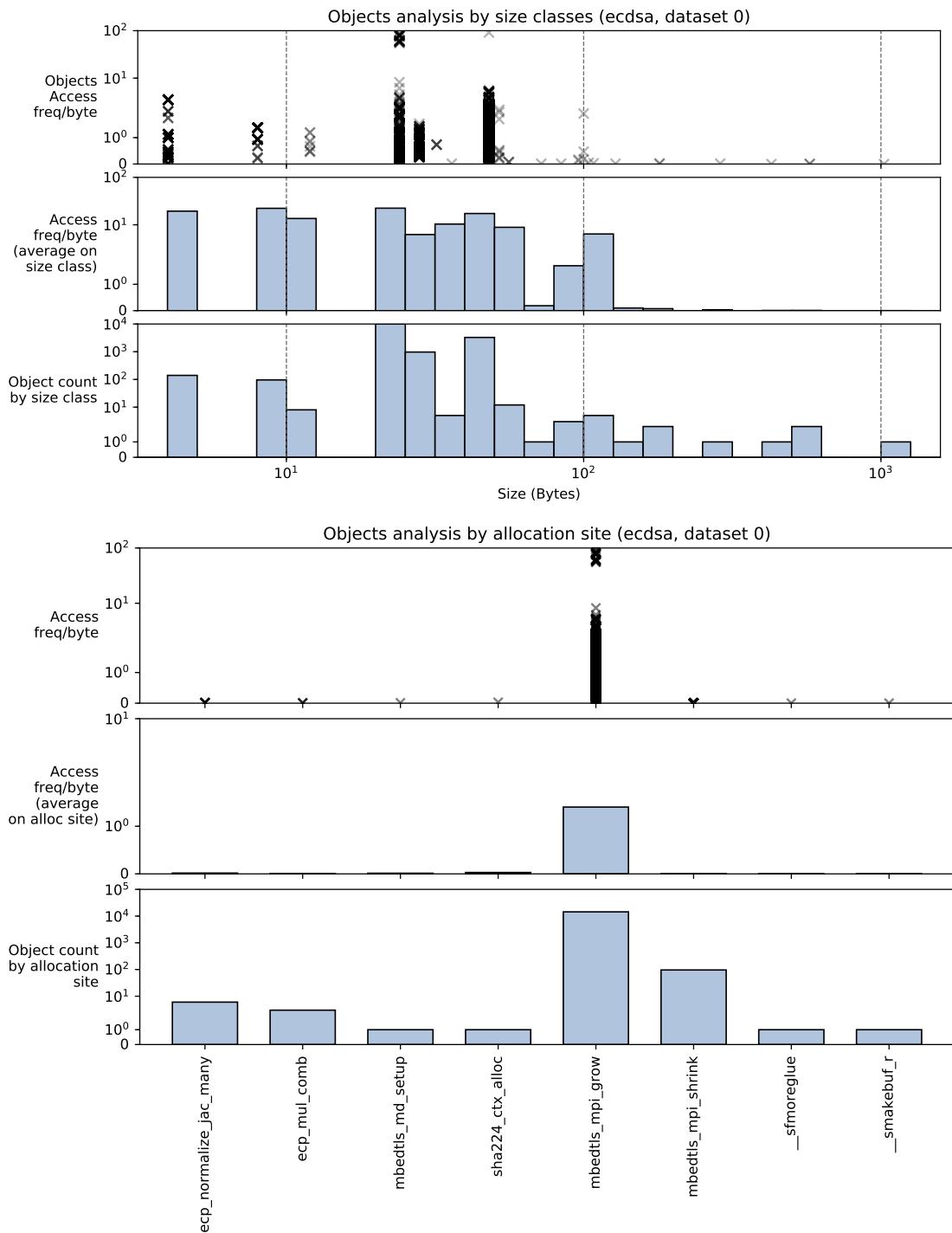


FIGURE B.17 – Ecdsa Jeu de données 0 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

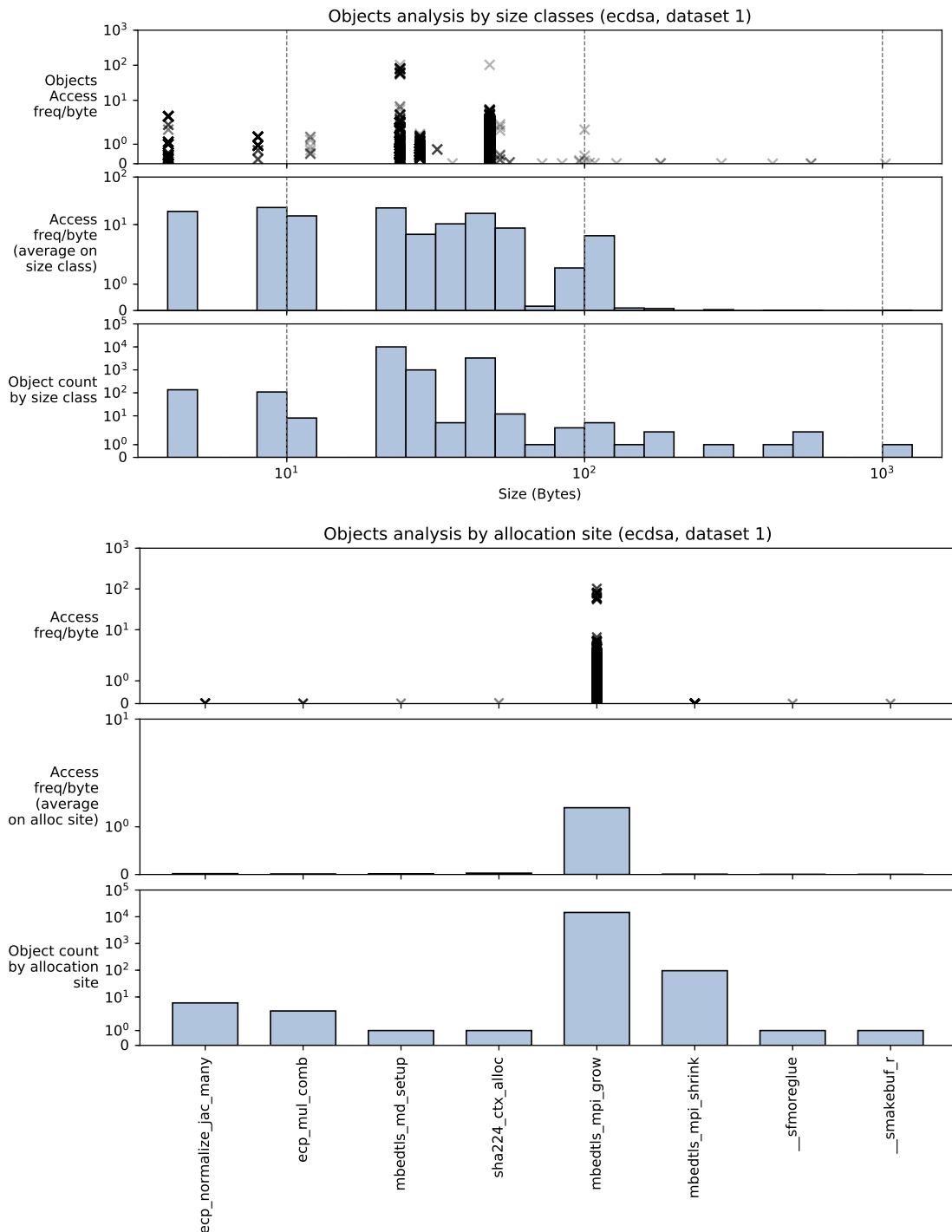


FIGURE B.18 – Ecdsa Jeu de données 1 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

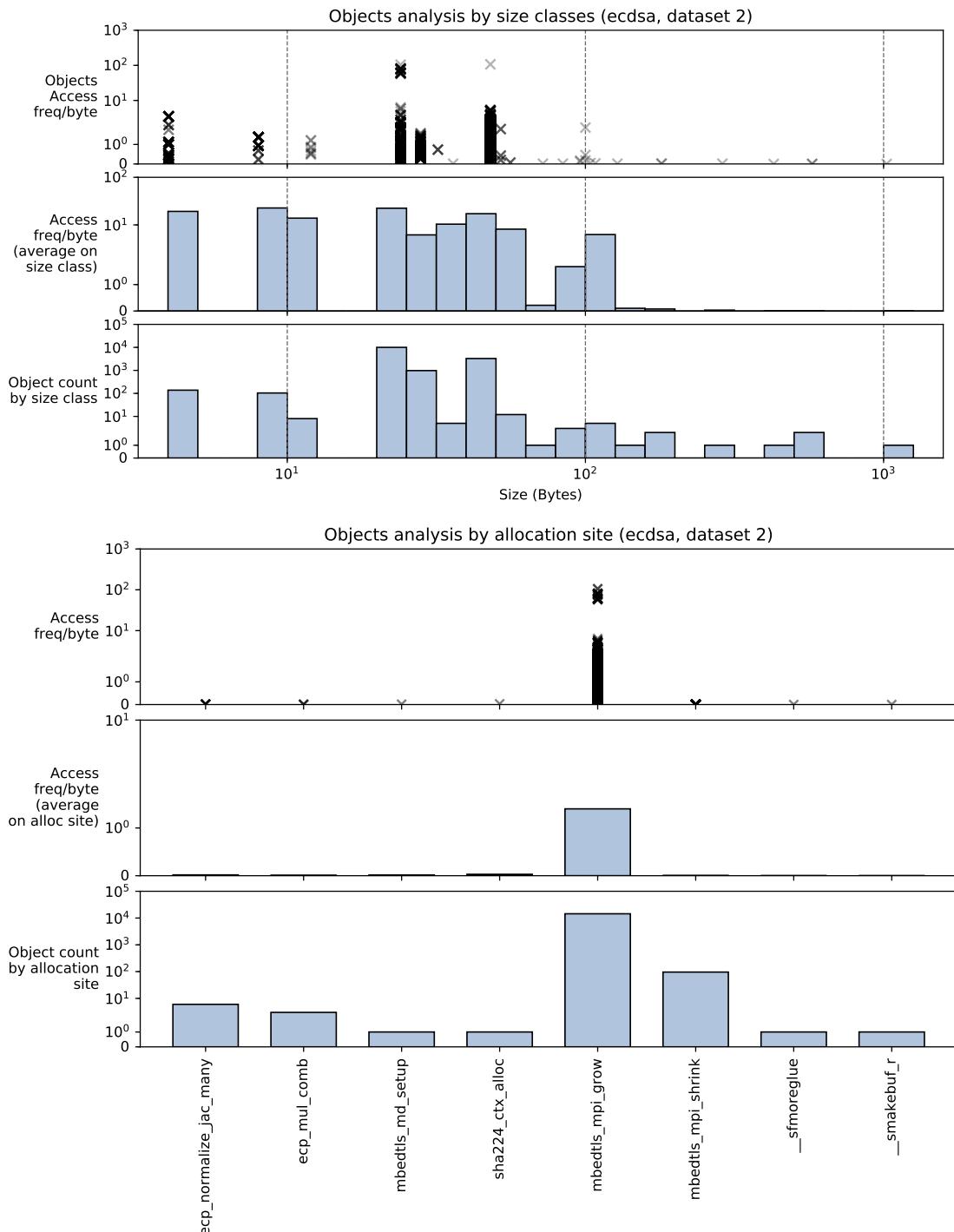


FIGURE B.19 – Ecdsa Jeu de données 2 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

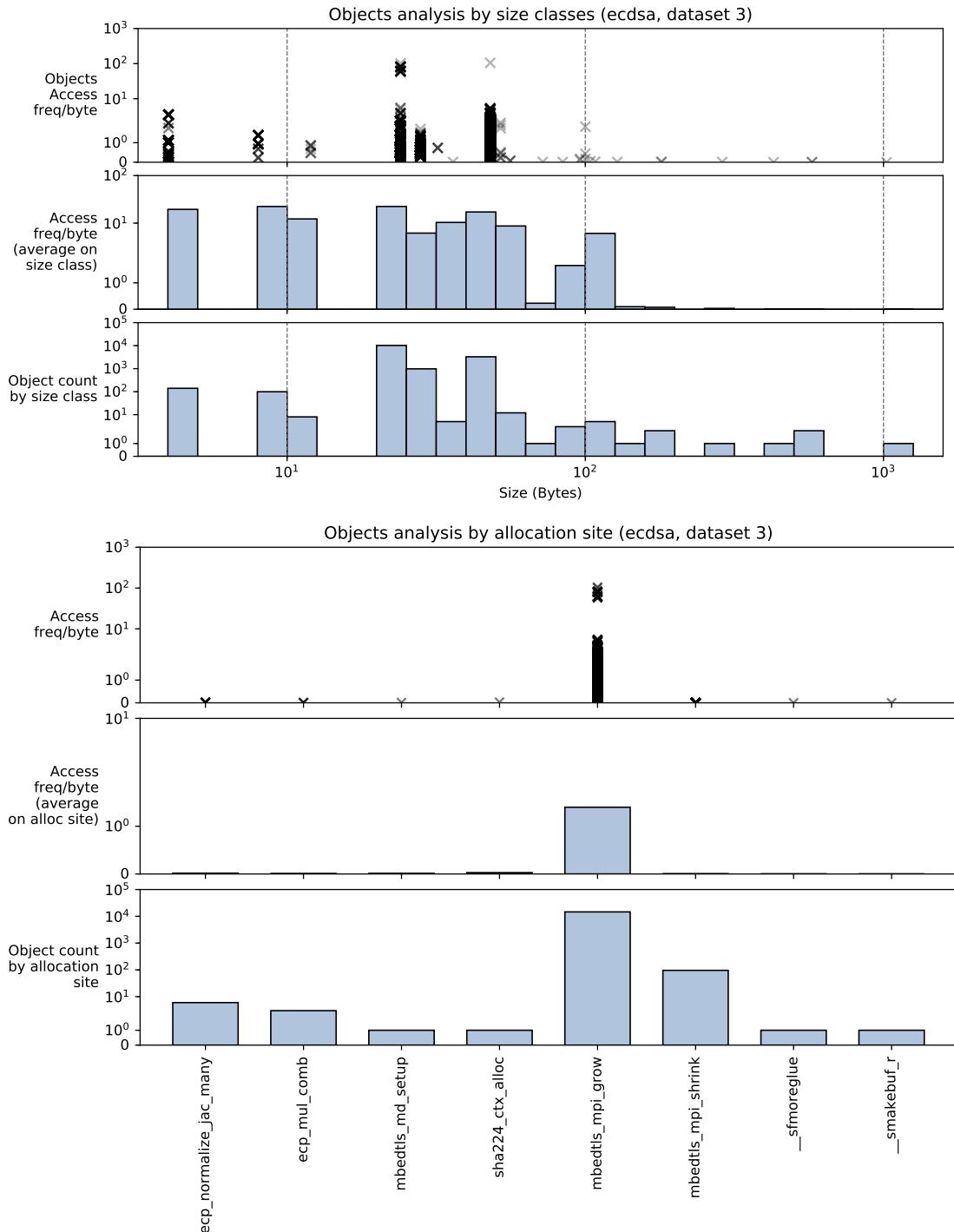


FIGURE B.20 – Ecdsa Jeu de données 3 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

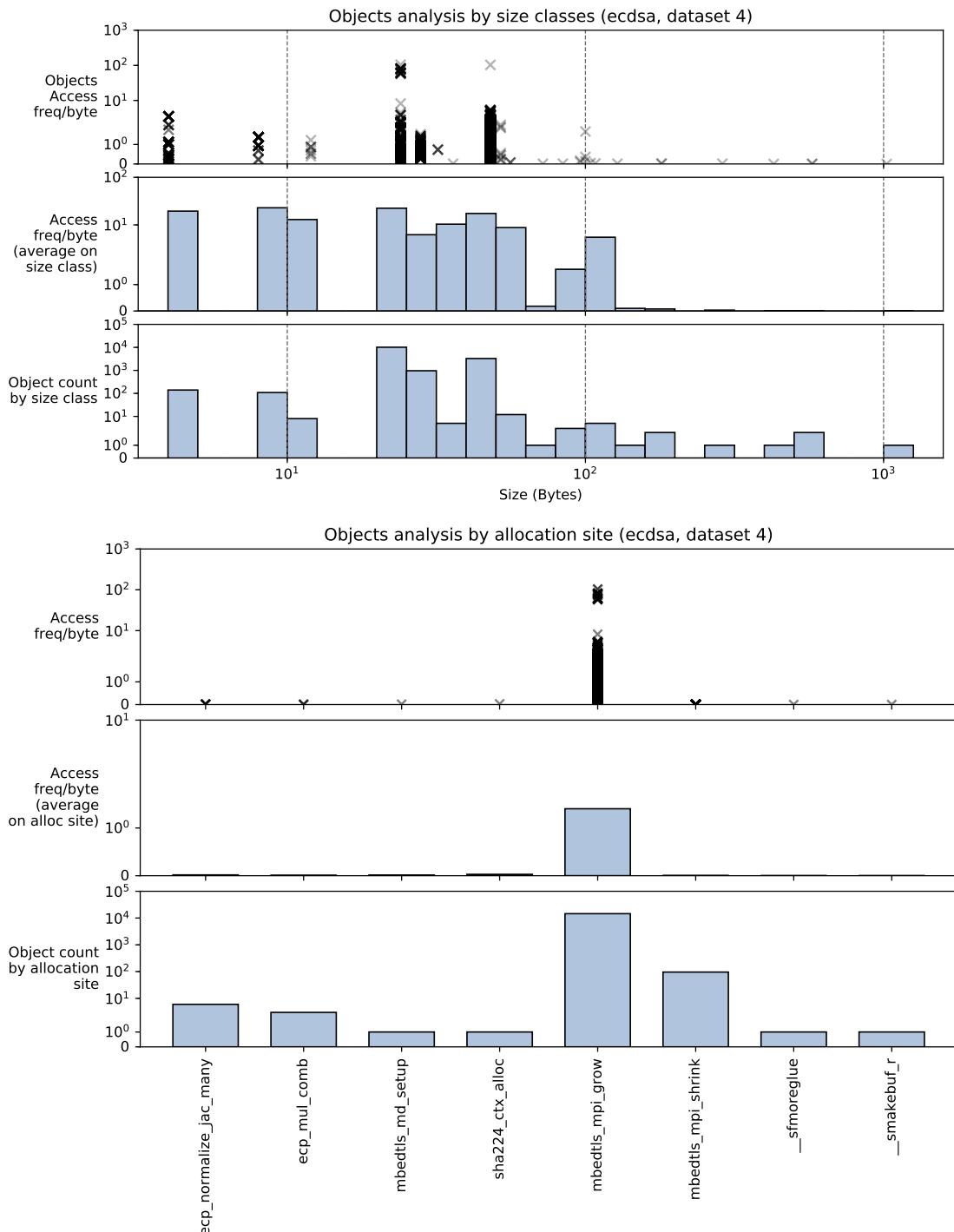


FIGURE B.21 – Ecdsa Jeu de données 4 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

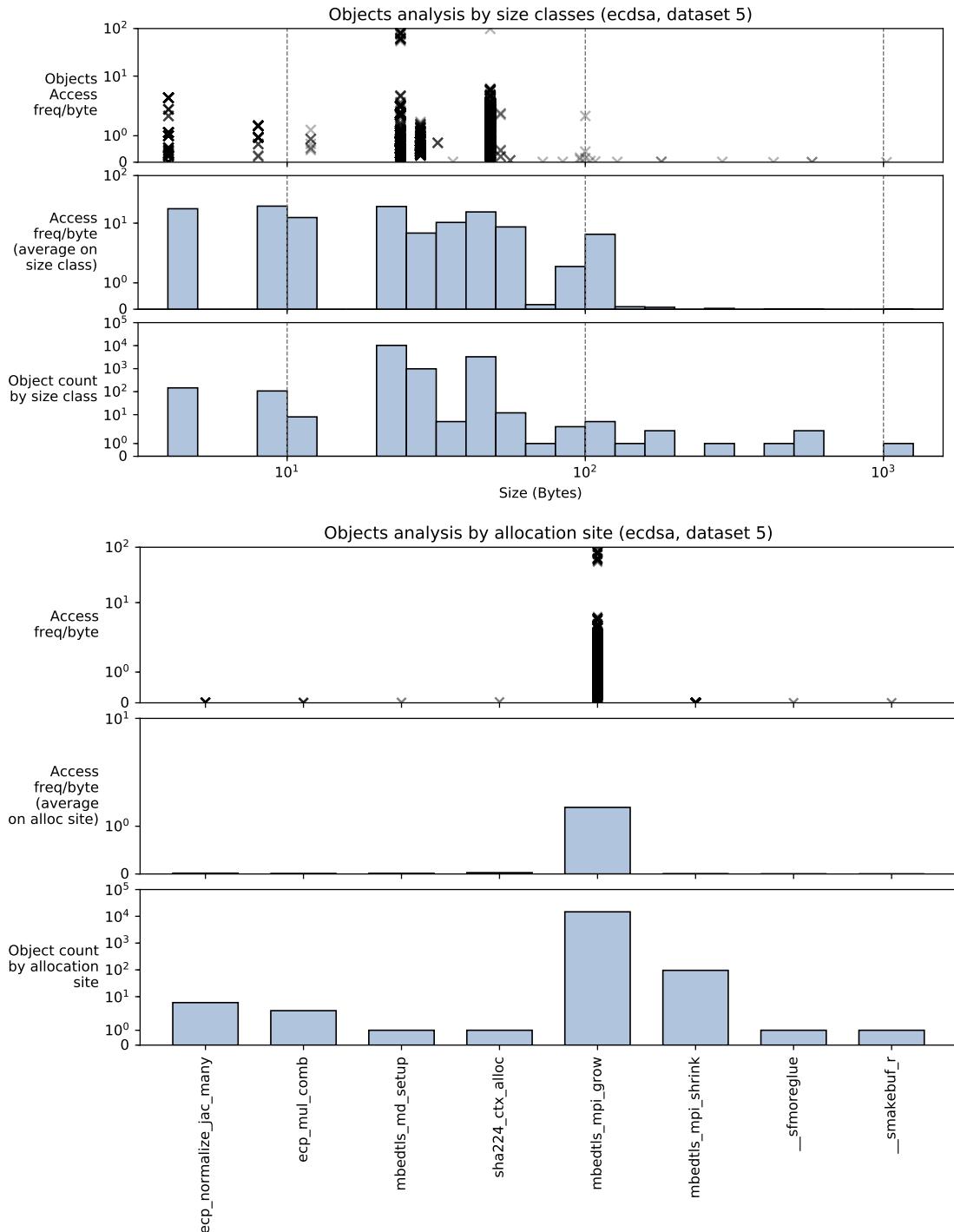


FIGURE B.22 – Ecdsa Jeu de données 5 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

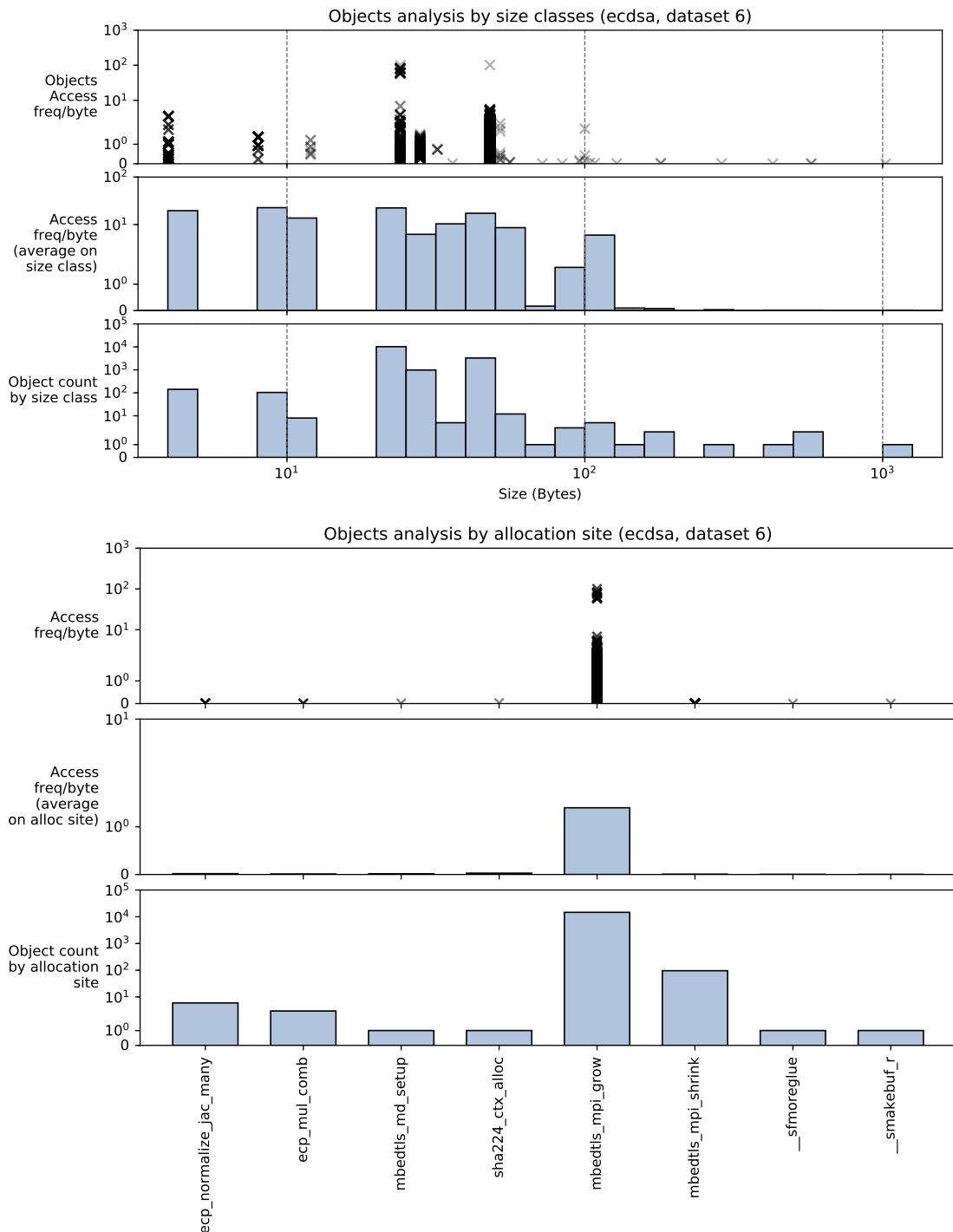


FIGURE B.23 – Ecdsa Jeu de données 6 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

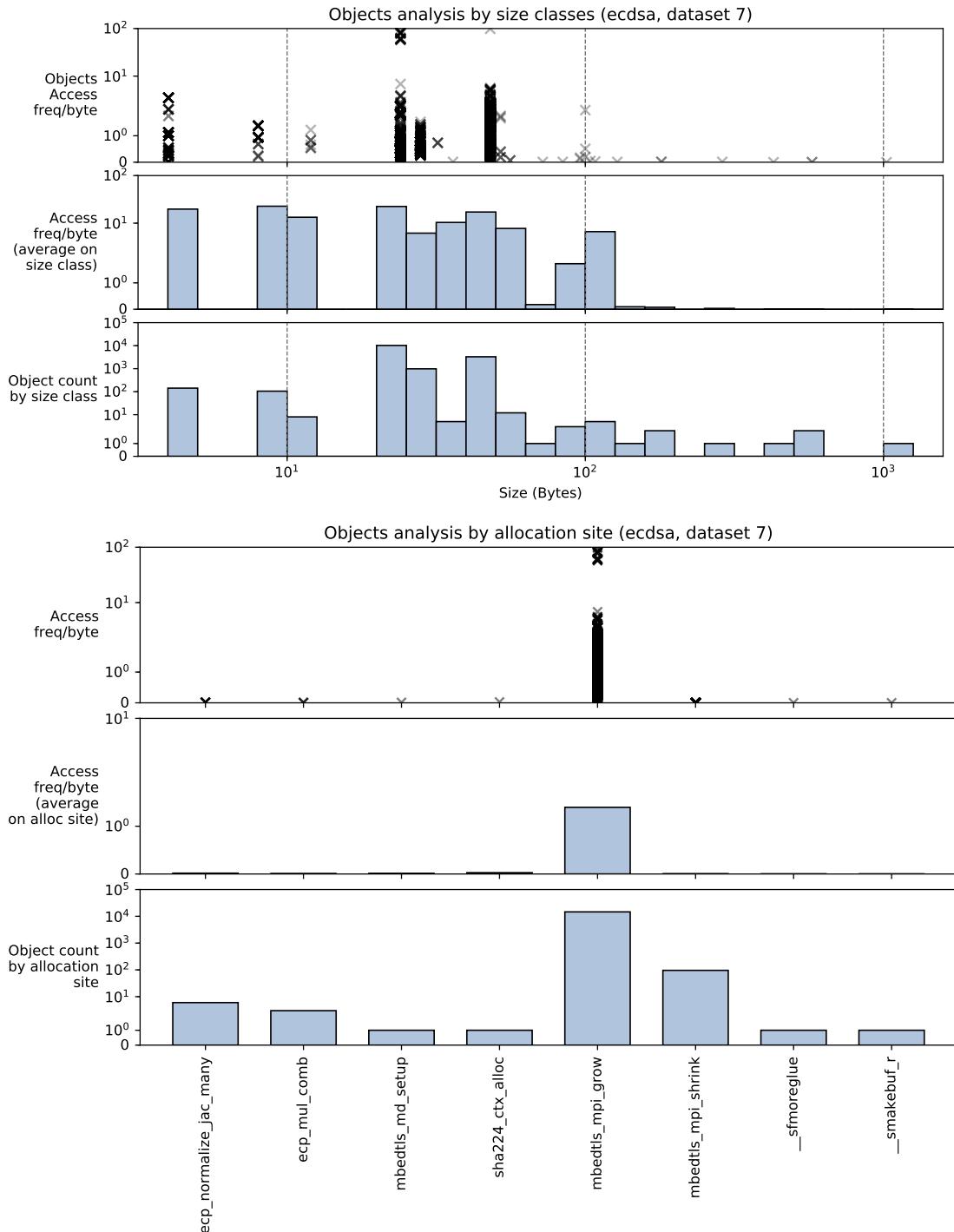


FIGURE B.24 – Ecdsa Jeu de données 7 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

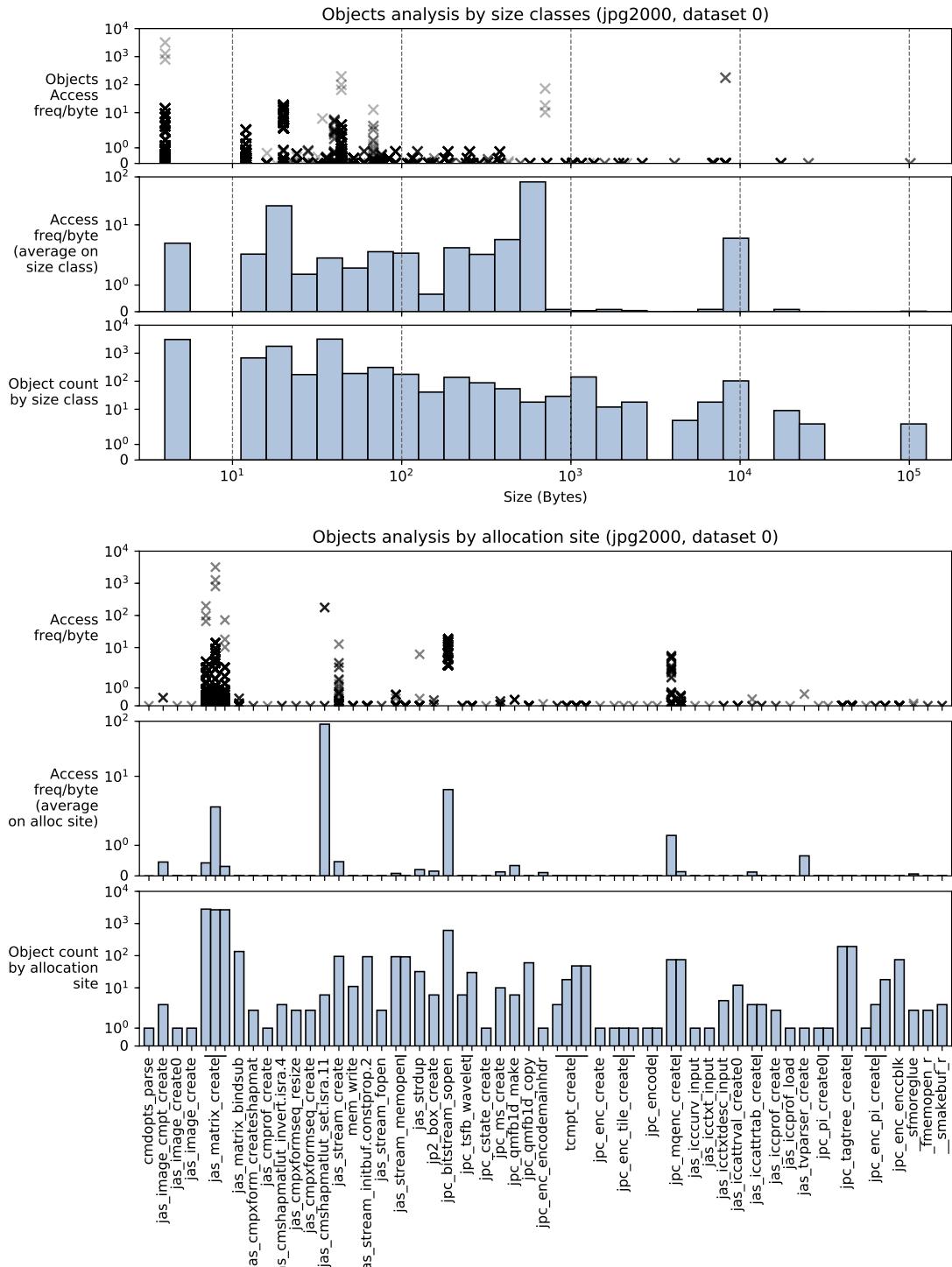
Jpg 2000

FIGURE B.25 – Jpg2000 Jeu de données 0 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

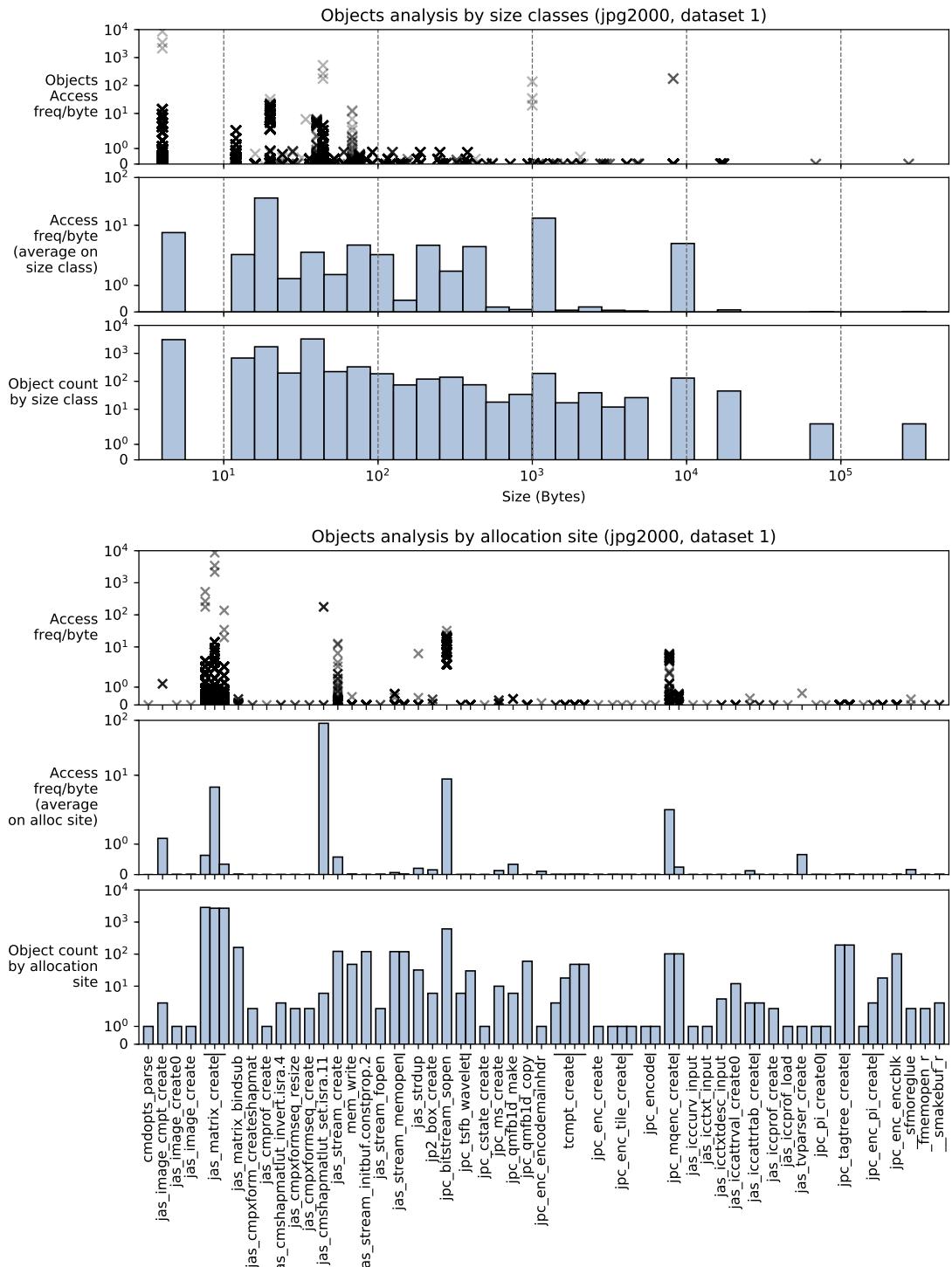


FIGURE B.26 – Jpg2000 Jeu de données 1 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

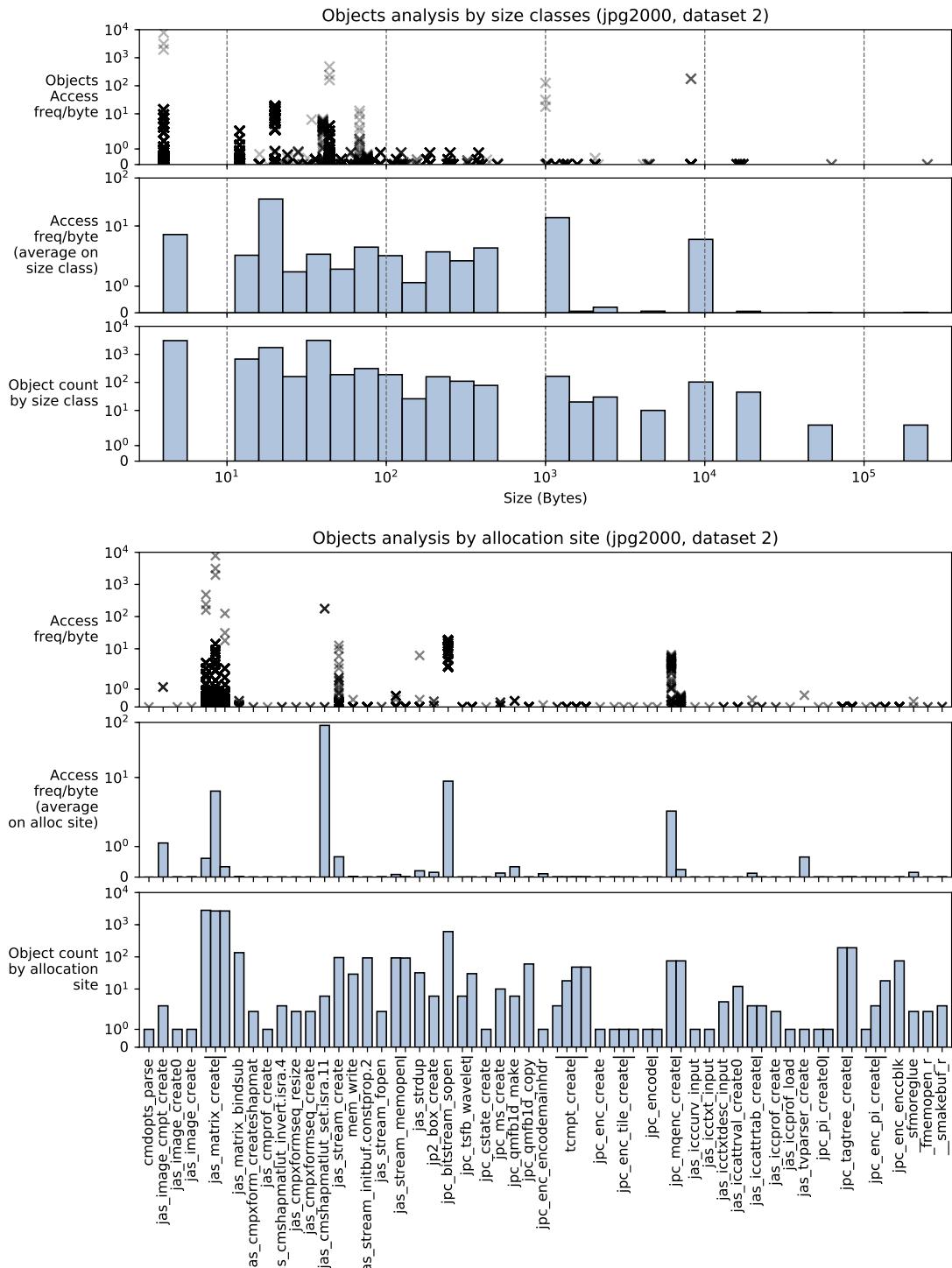


FIGURE B.27 – Jpg2000 Jeu de données 2 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

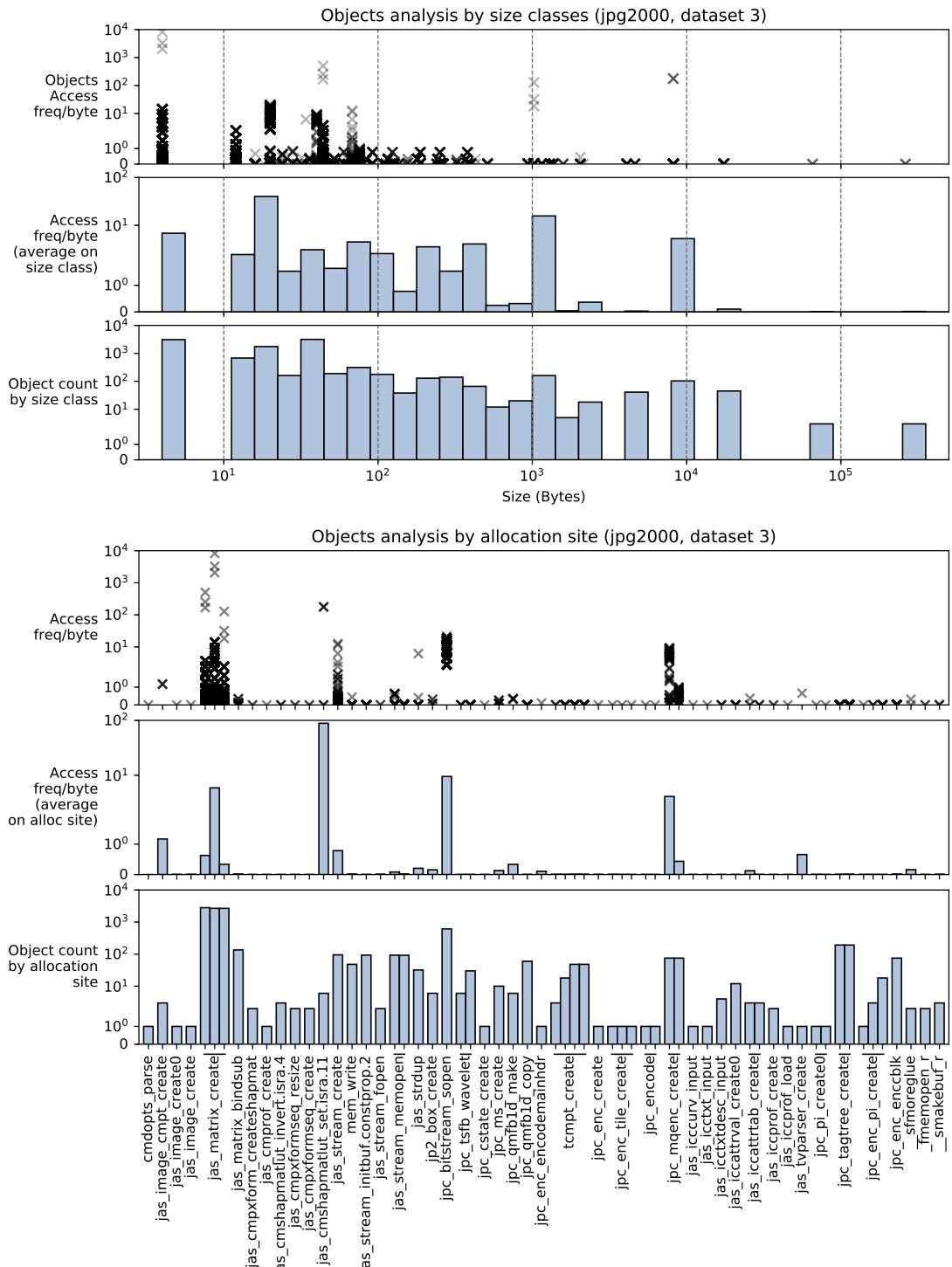


FIGURE B.28 – Jpg2000 Jeu de données 3 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

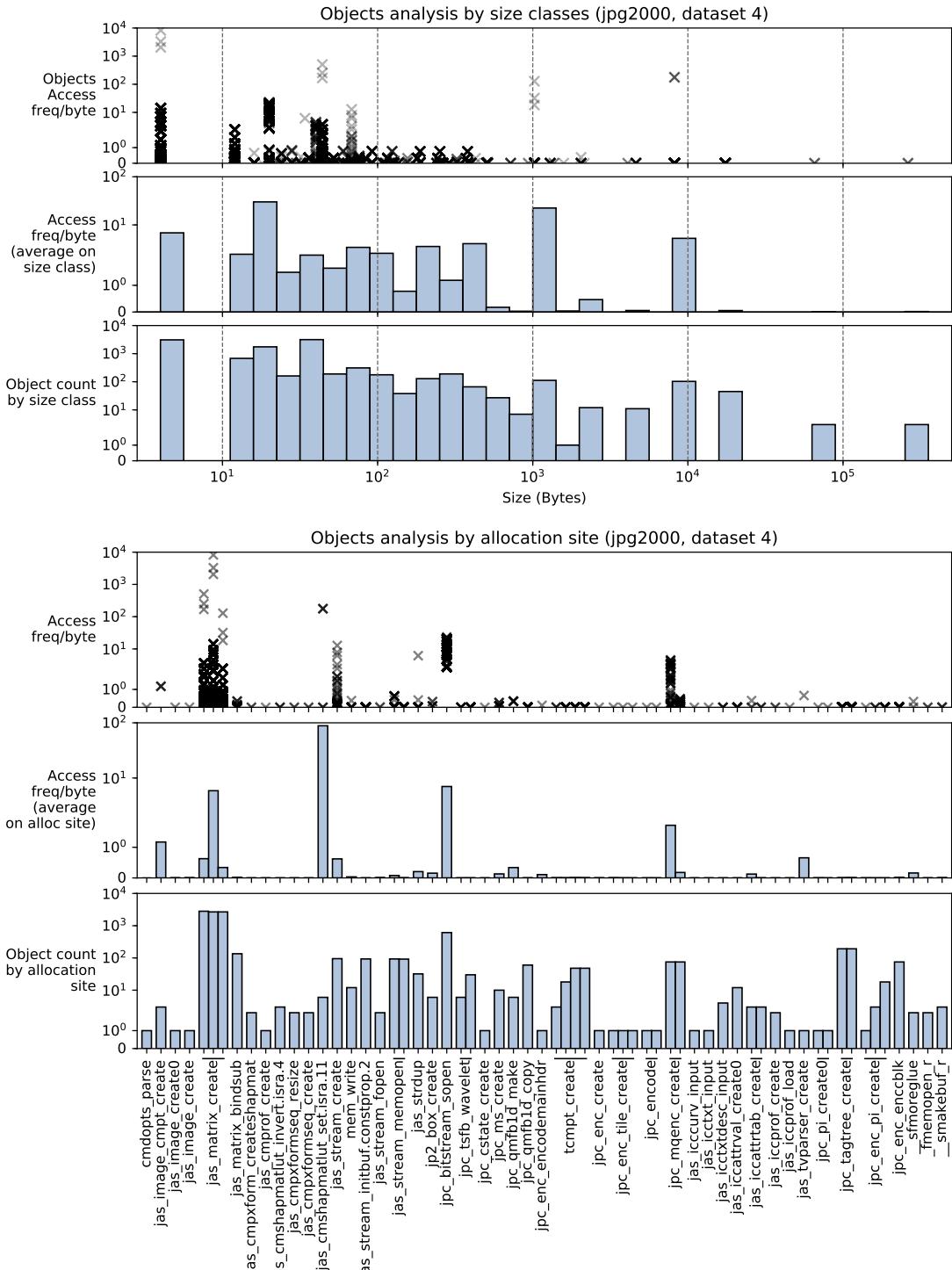


FIGURE B.29 – Jpg2000 Jeu de données 4 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

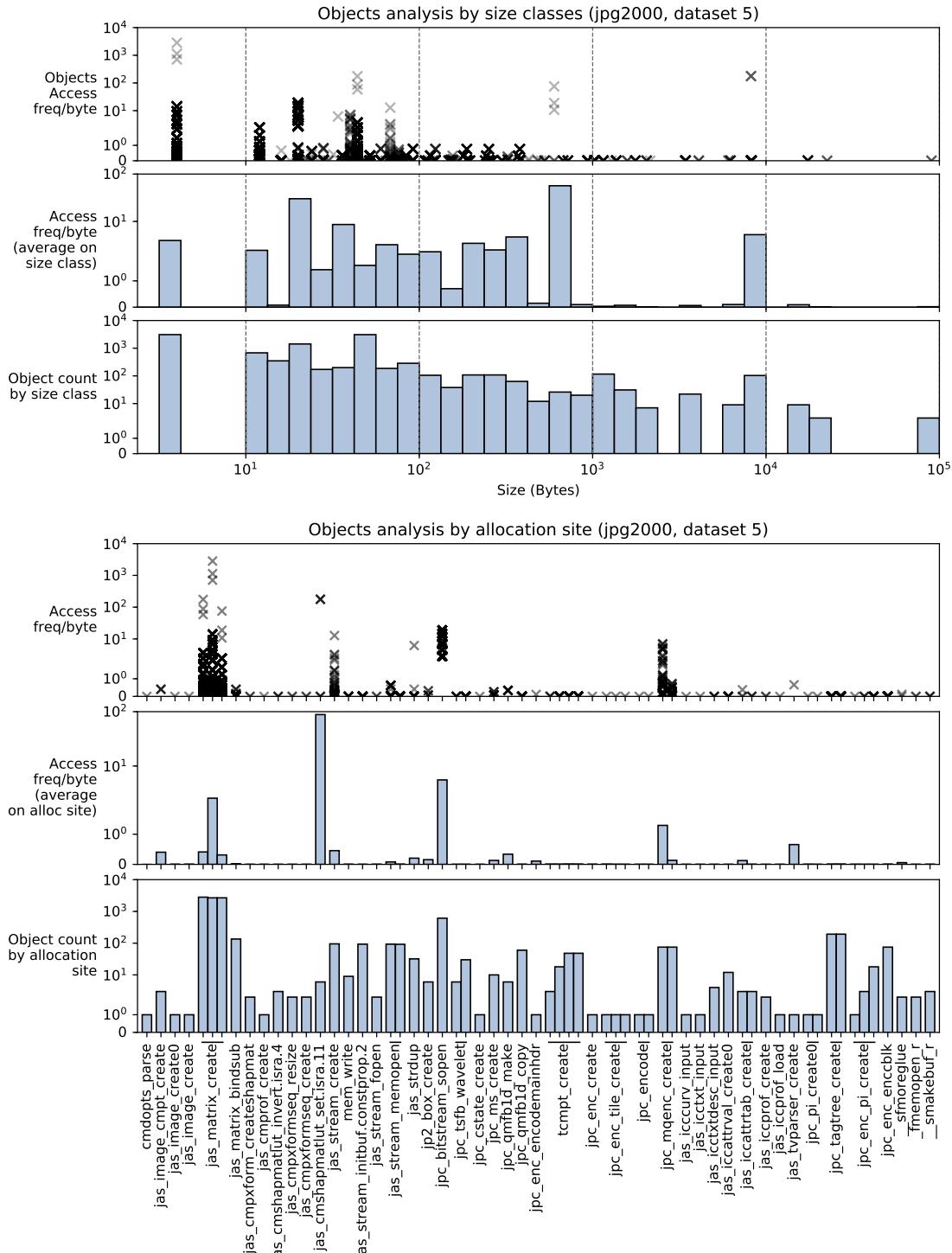


FIGURE B.30 – Jpg2000 Jeu de données 5 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

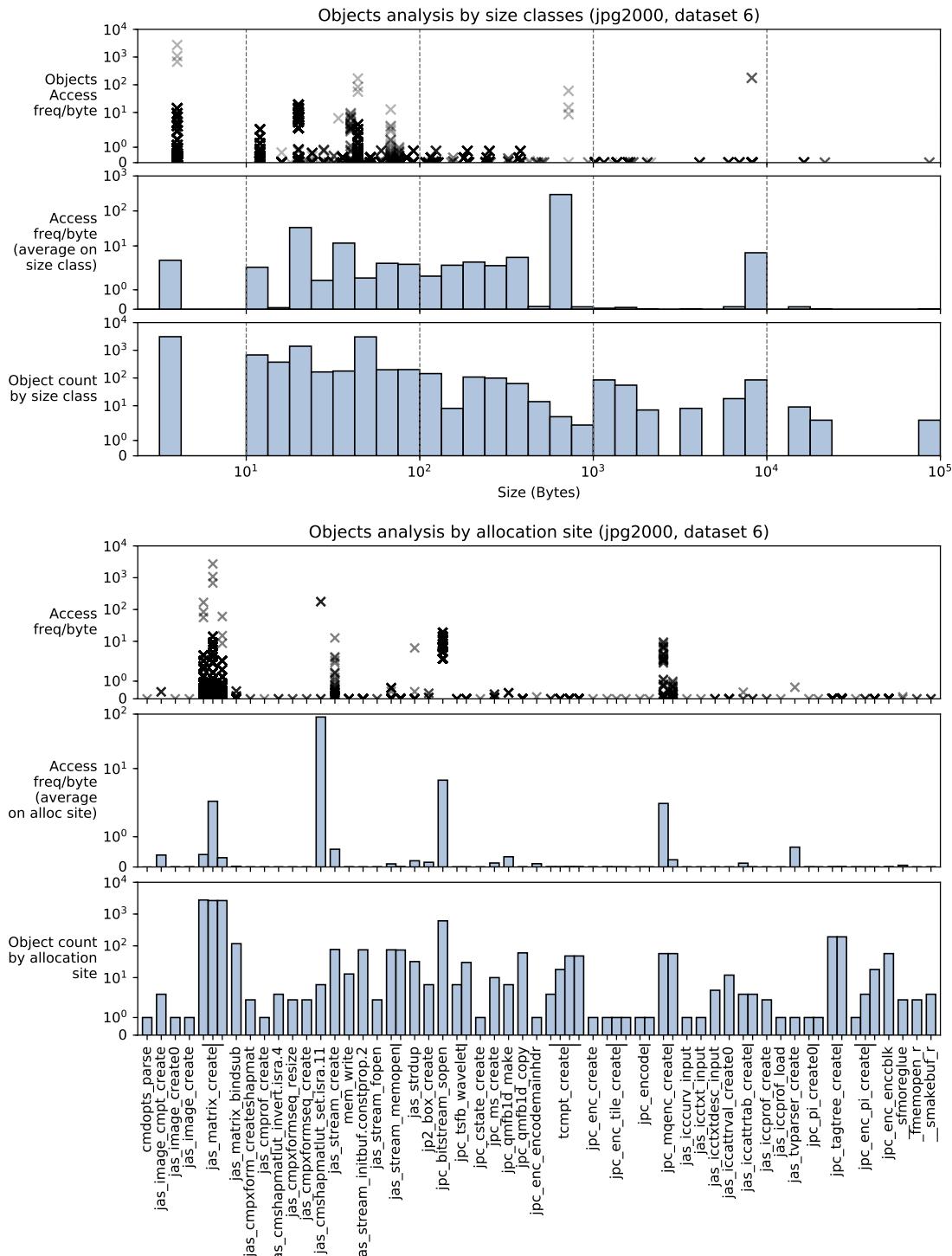


FIGURE B.31 – Jpg2000 Jeu de données 6 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

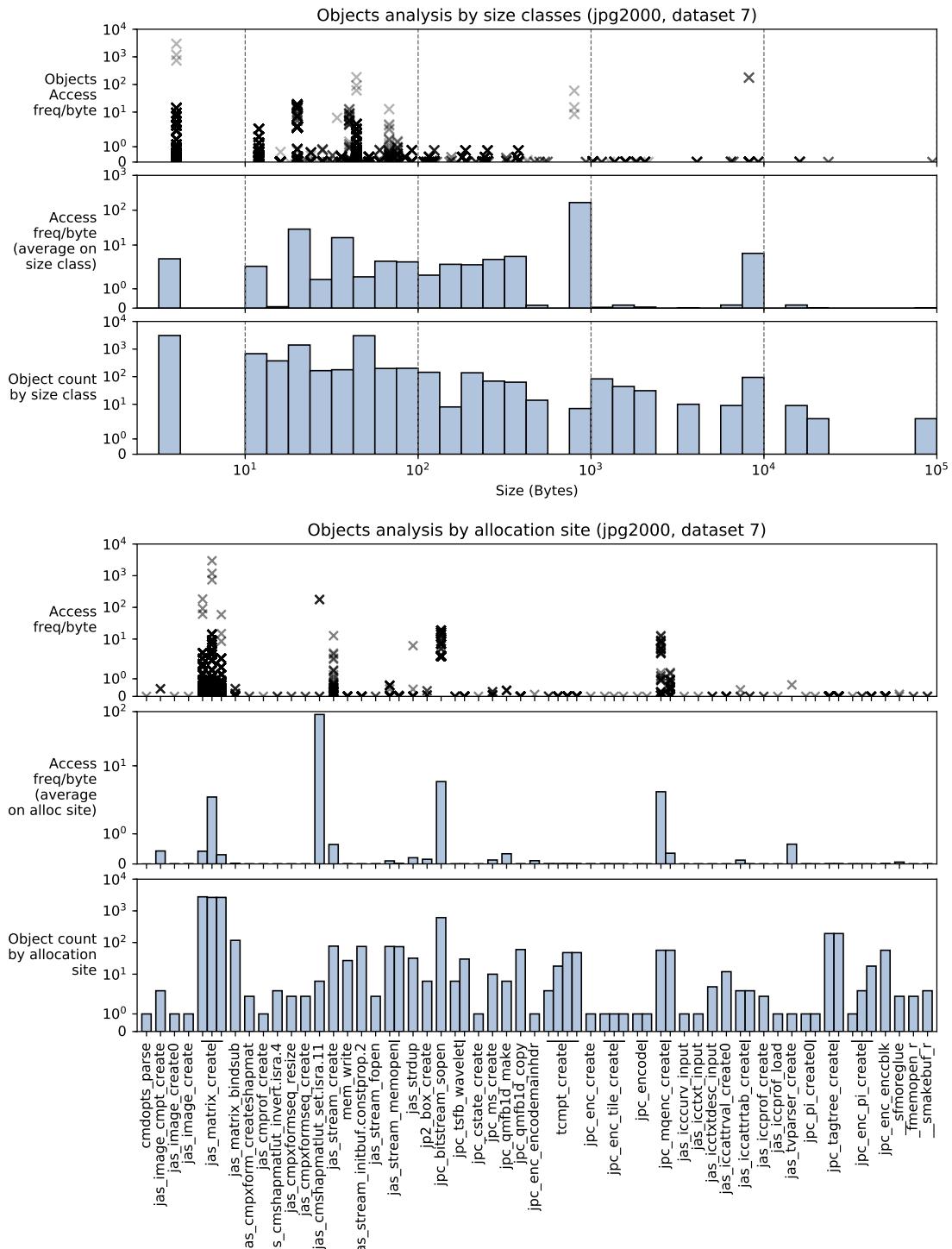


FIGURE B.32 – **Jpg2000** Jeu de données 7 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

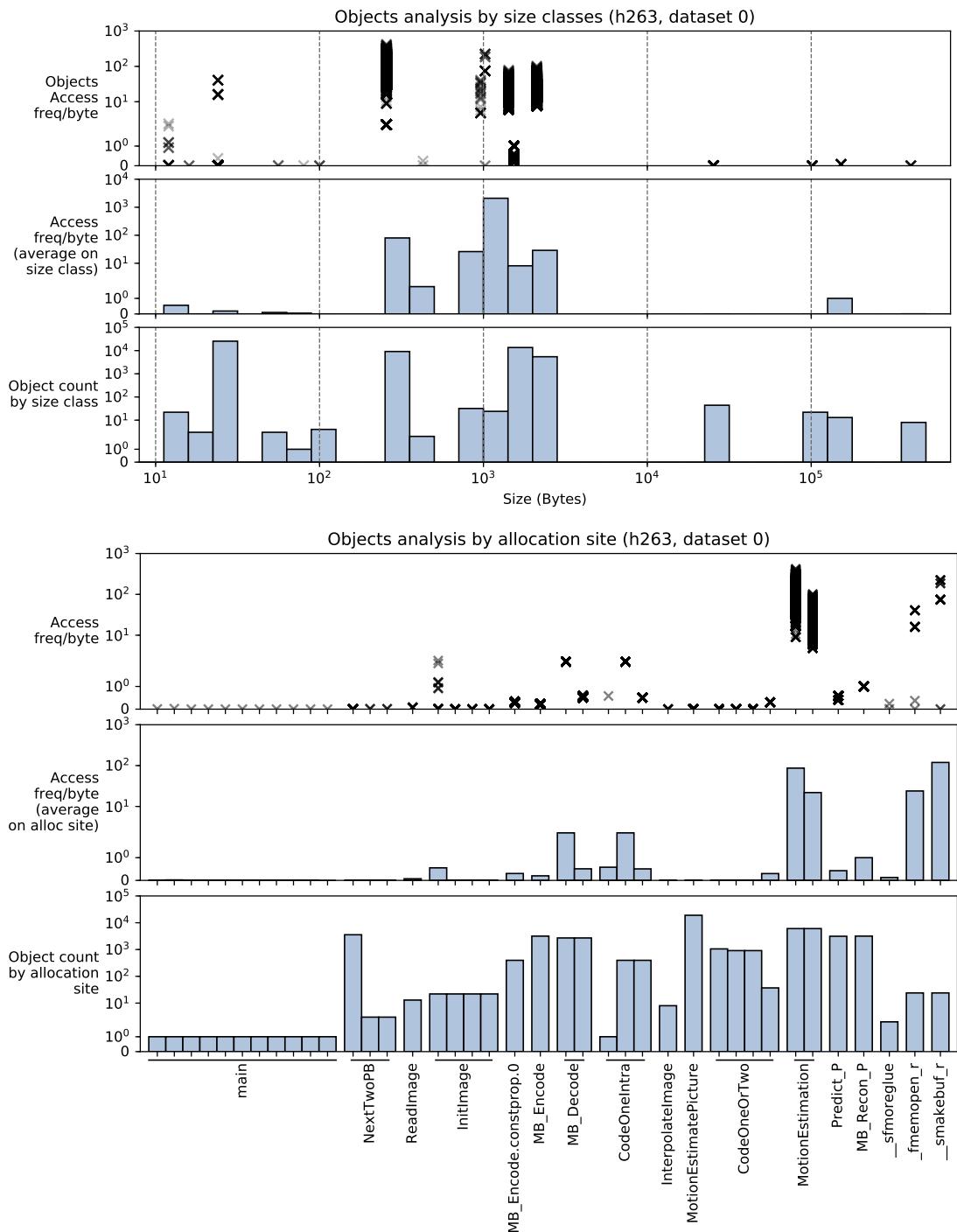
H263

FIGURE B.33 – H263 Jeu de données 0 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

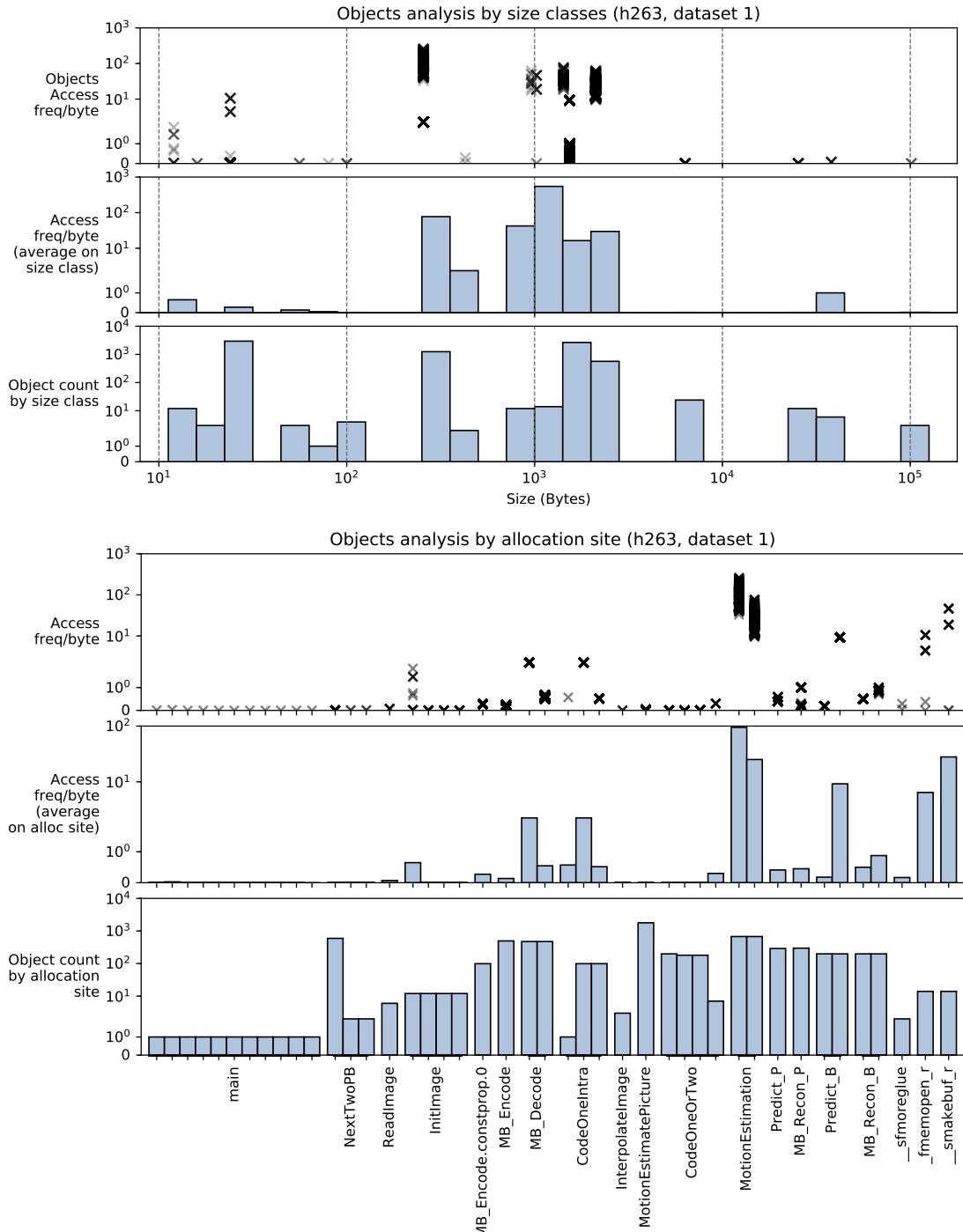


FIGURE B.34 – H263 Jeu de données 1 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

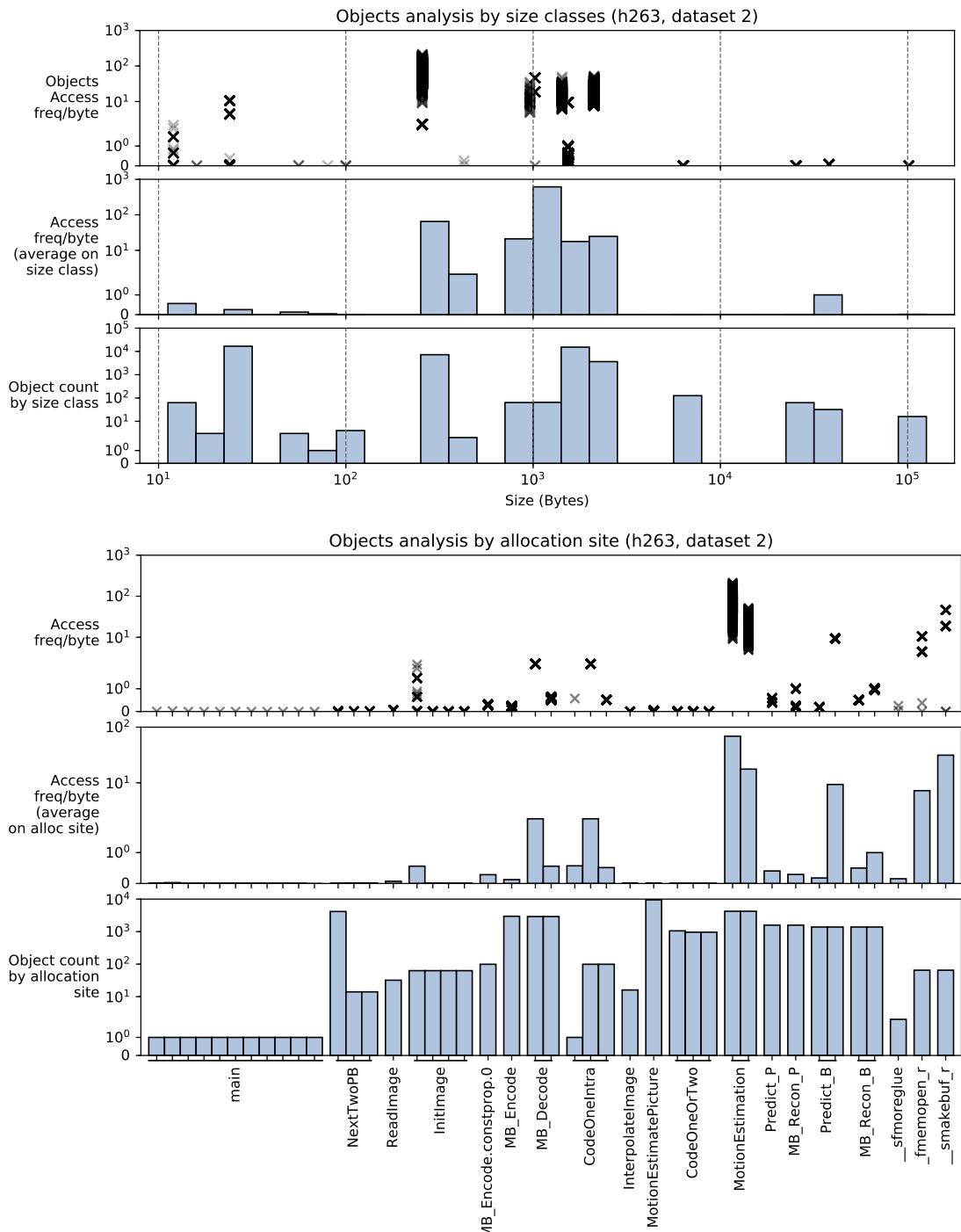


FIGURE B.35 – H263 Jeu de données 2 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

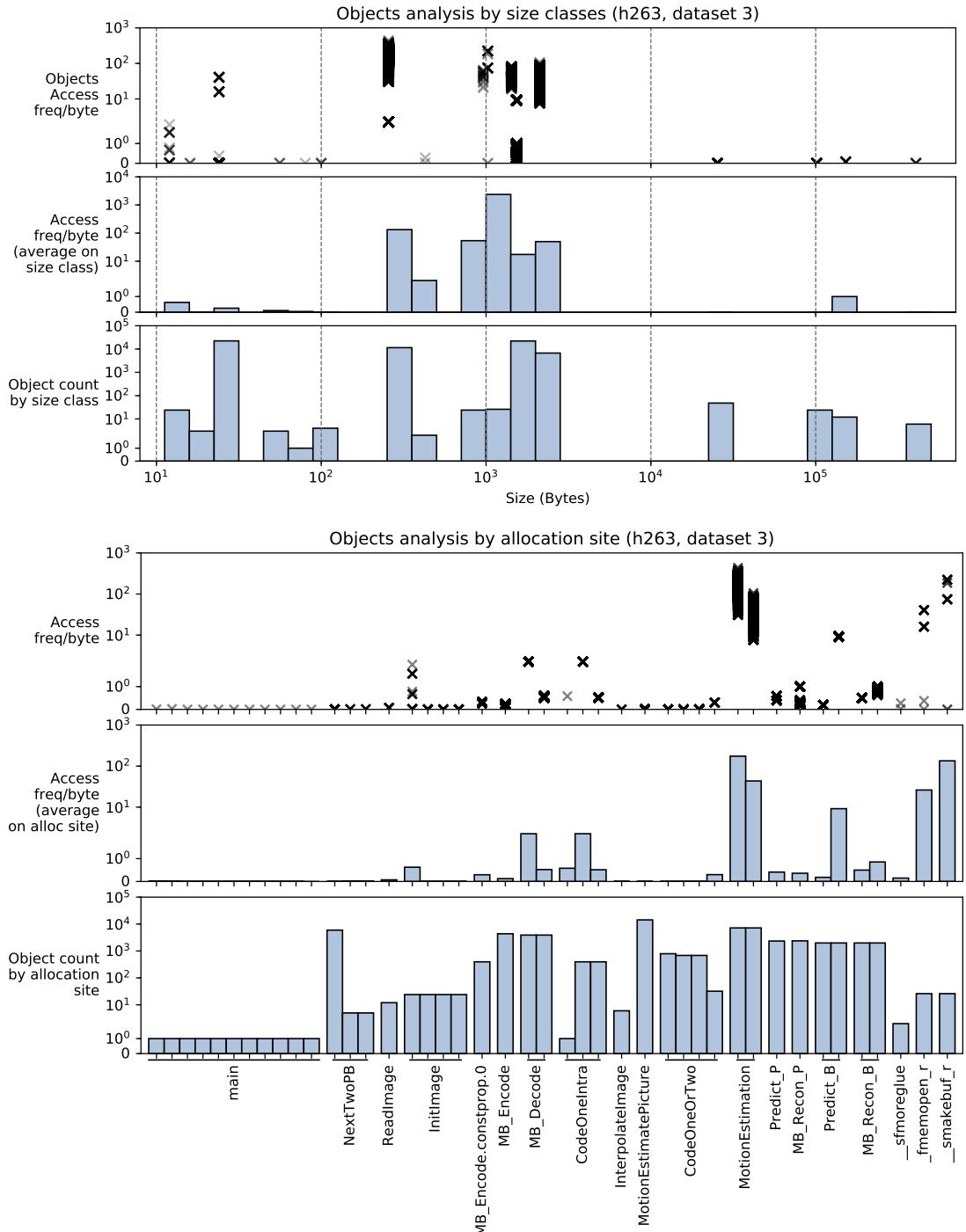


FIGURE B.36 – H263 Jeu de données 3 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

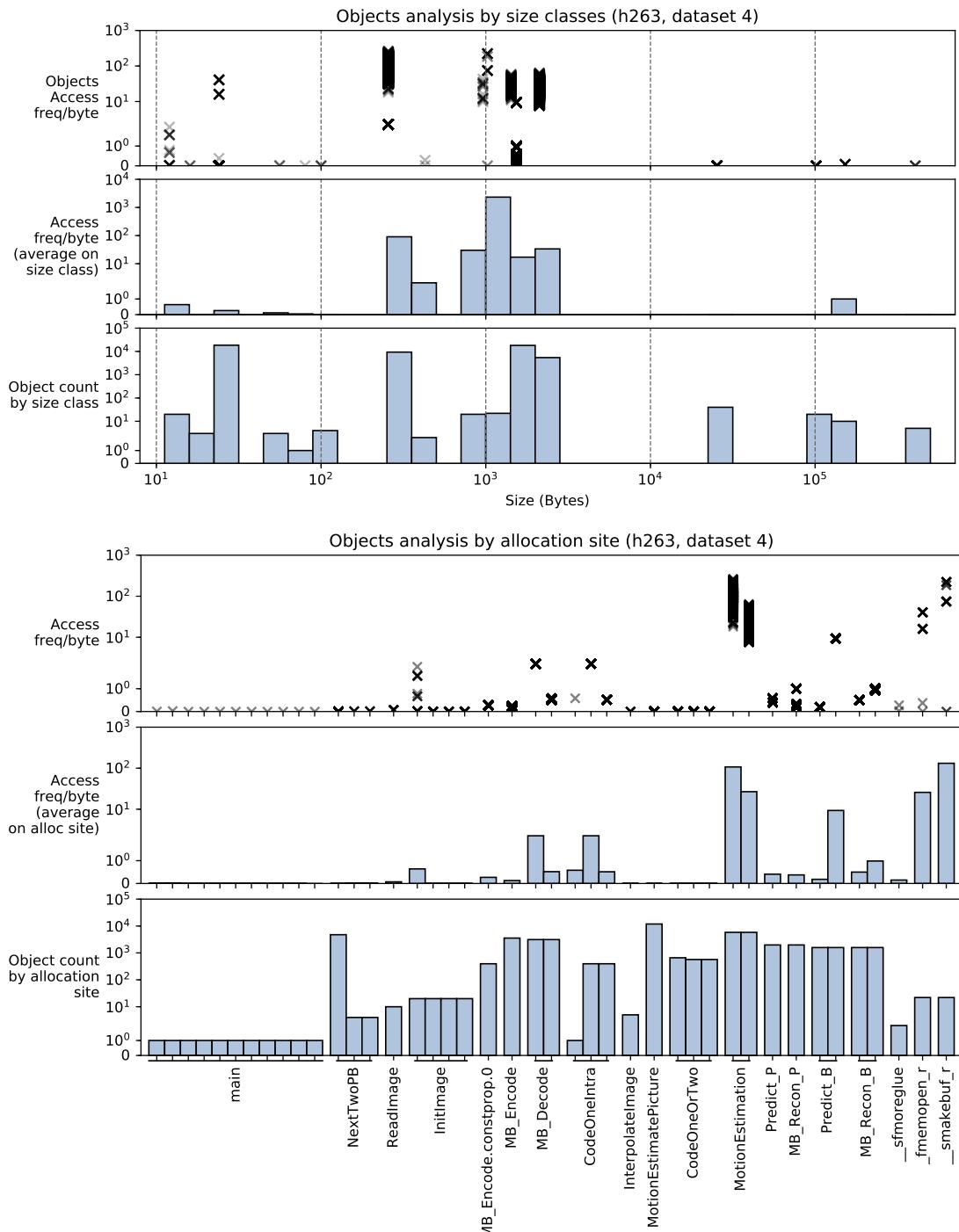


FIGURE B.37 – H263 Jeu de données 4 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

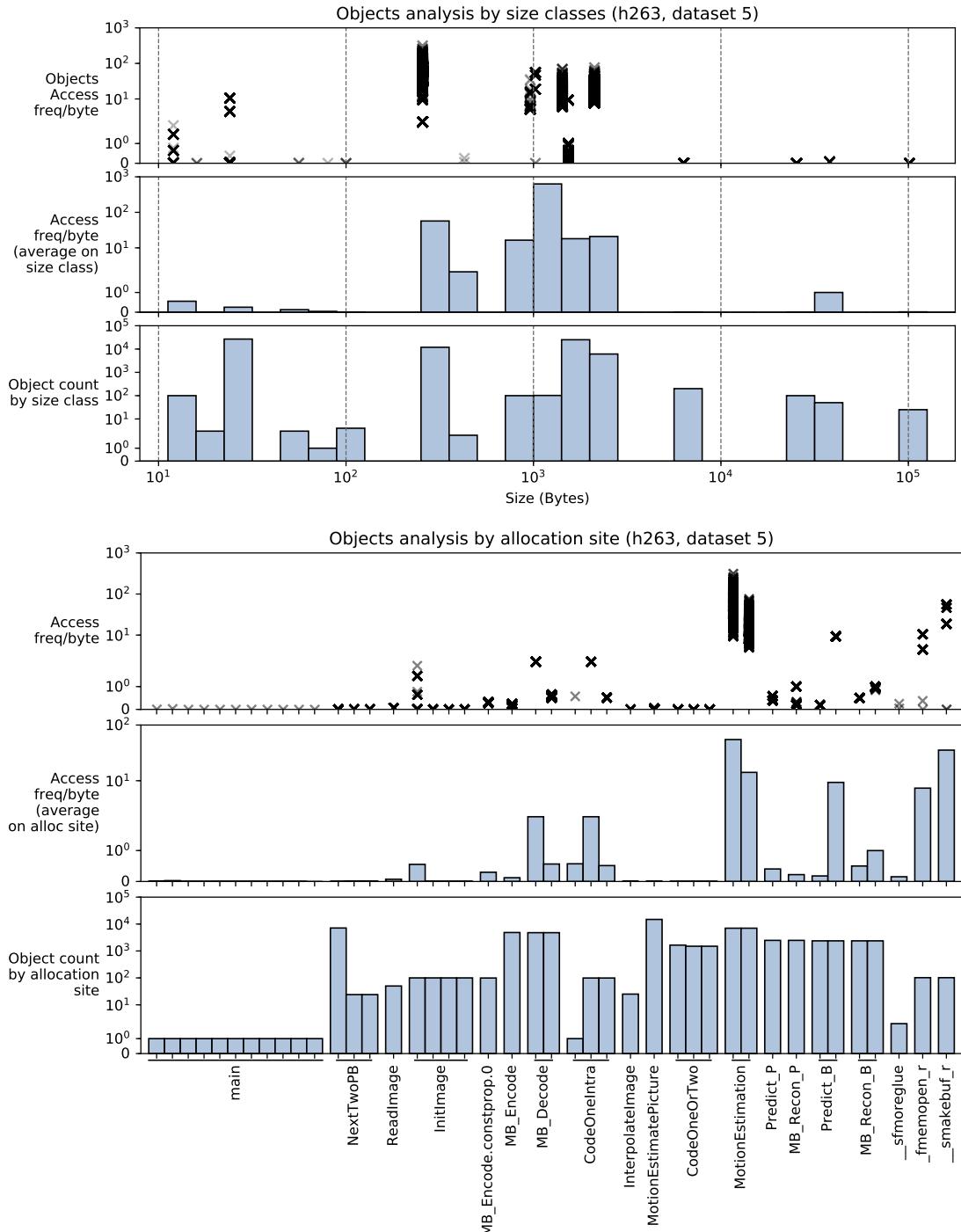


FIGURE B.38 – H263 Jeu de données 5 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

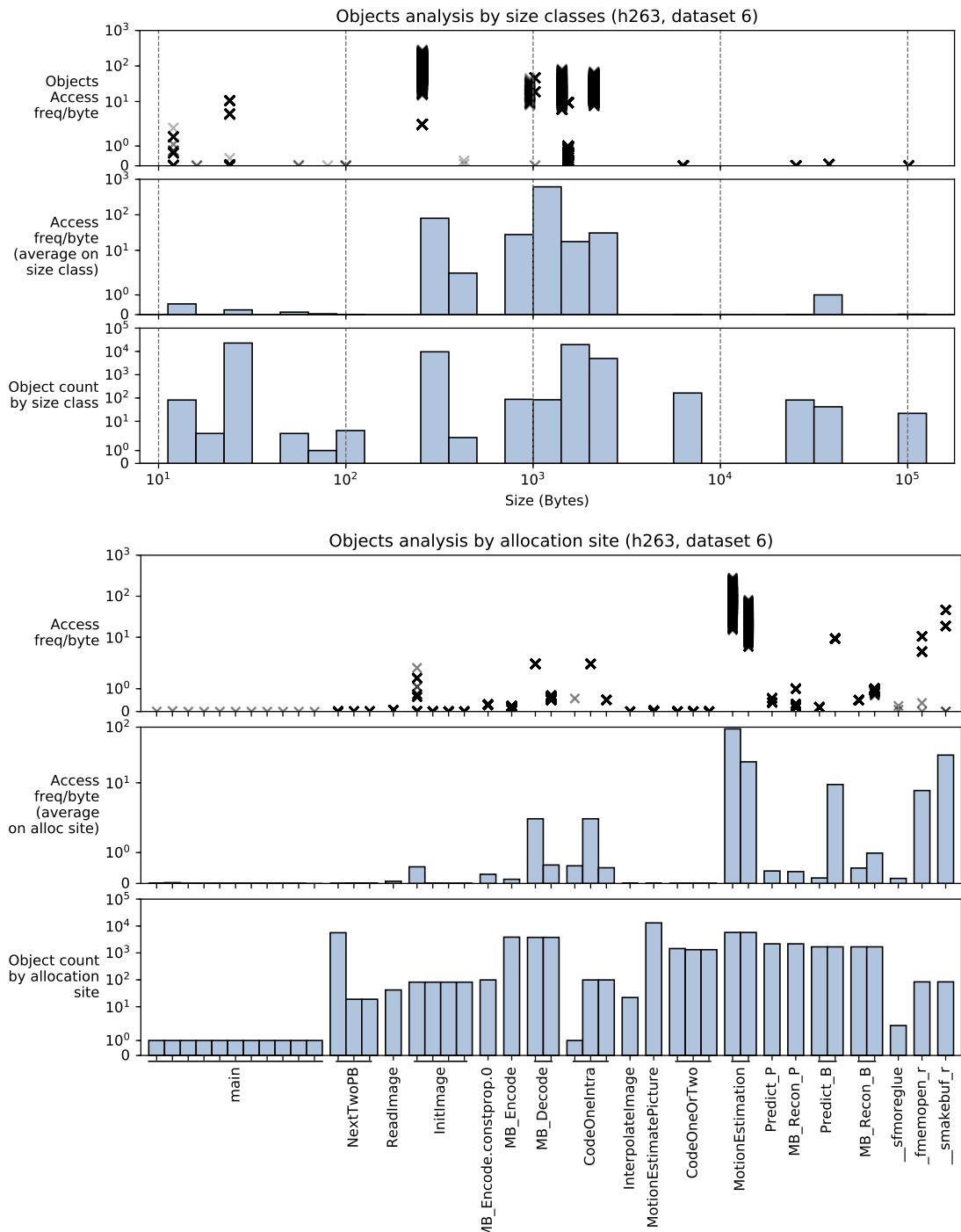


FIGURE B.39 – H263 Jeu de données 6 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

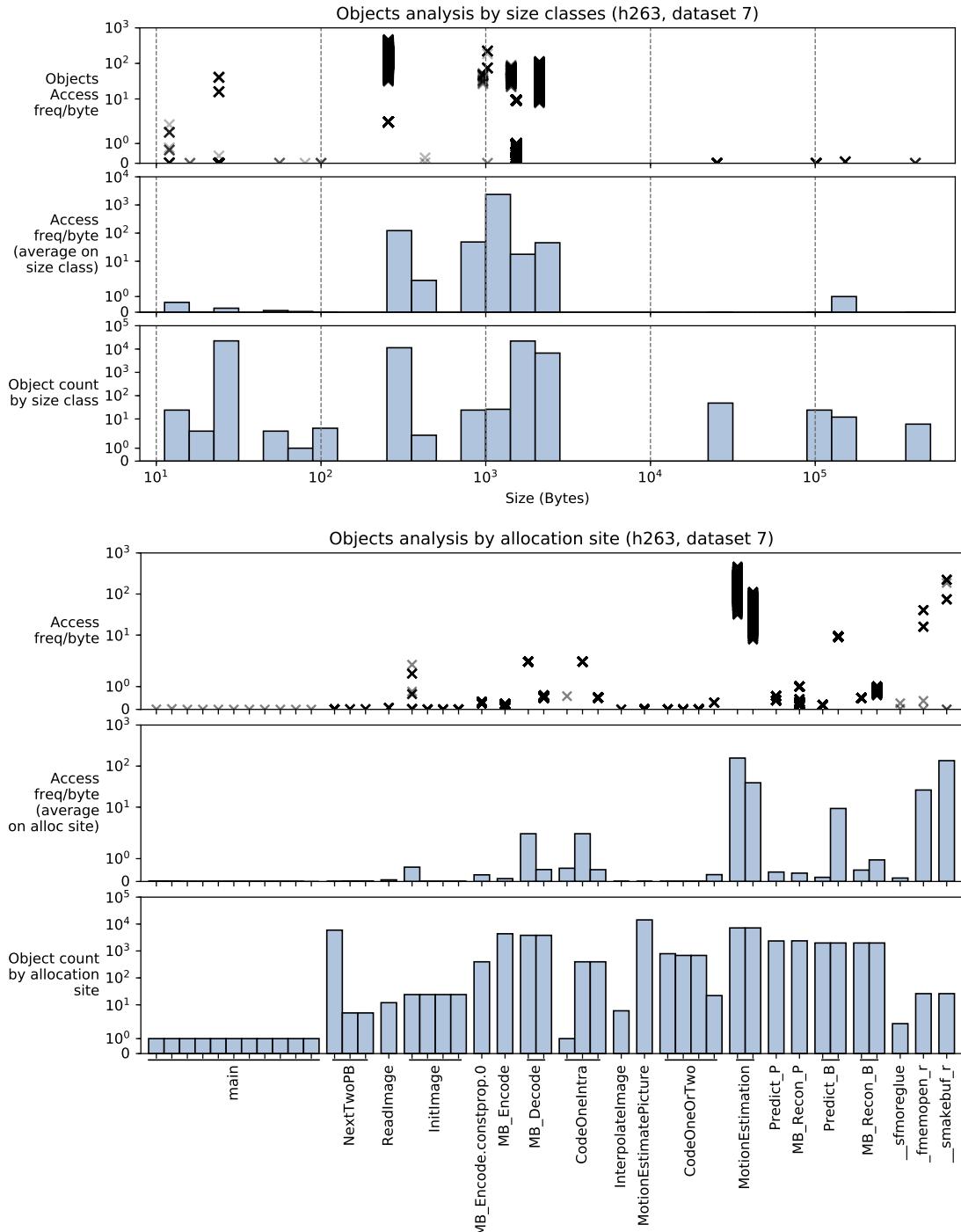


FIGURE B.40 – H263 Jeu de données 7 : analyse des objets par classe de taille et par site d’allocation – exécutions de référence.

Annexe C

Résultats en performance par jeu de données

Pour chaque application nous fournissons ici pour chaque jeu de données d'évaluation (d0, d1, d2 et d3) le graphe des performances présentant la réduction du temps d'exécution par rapport à l'exécution de référence en fonction du pourcentage de mémoire rapide dans l'architecture du tas.

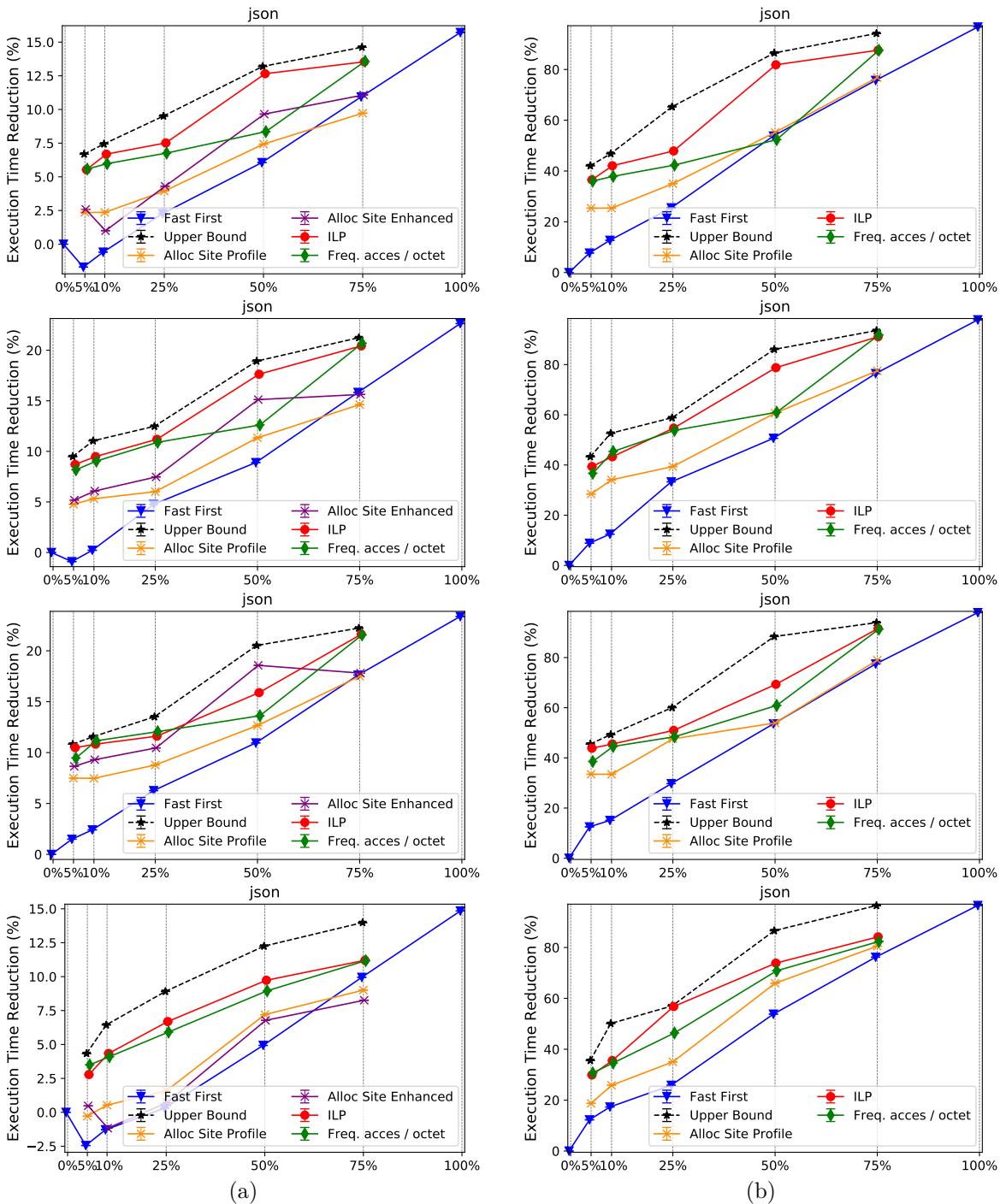


FIGURE C.1 – Json Parser : Évaluation des performances – par jeu de données
 Technologies mémoires : (a) scénario "MRAM", (b) scénario "ReRAM"
 Jeux de données : 0 à 3 de haut en bas

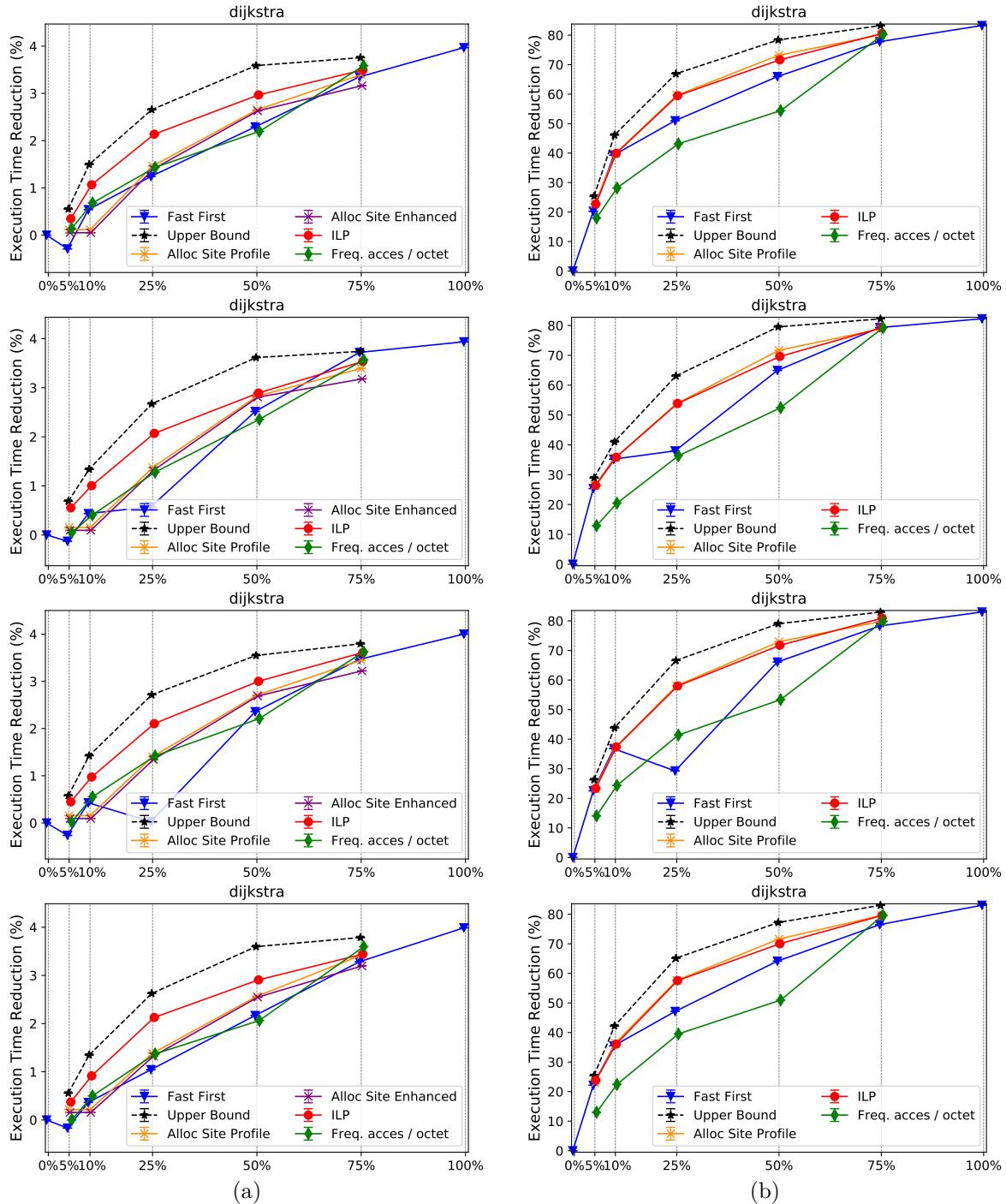


FIGURE C.2 – Dijkstra : Évaluation des performances – par jeu de données
 Technologies mémoires : (a) scénario "MRAM", (b) scénario "ReRAM"
 Jeux de données : 0 à 3 de haut en bas

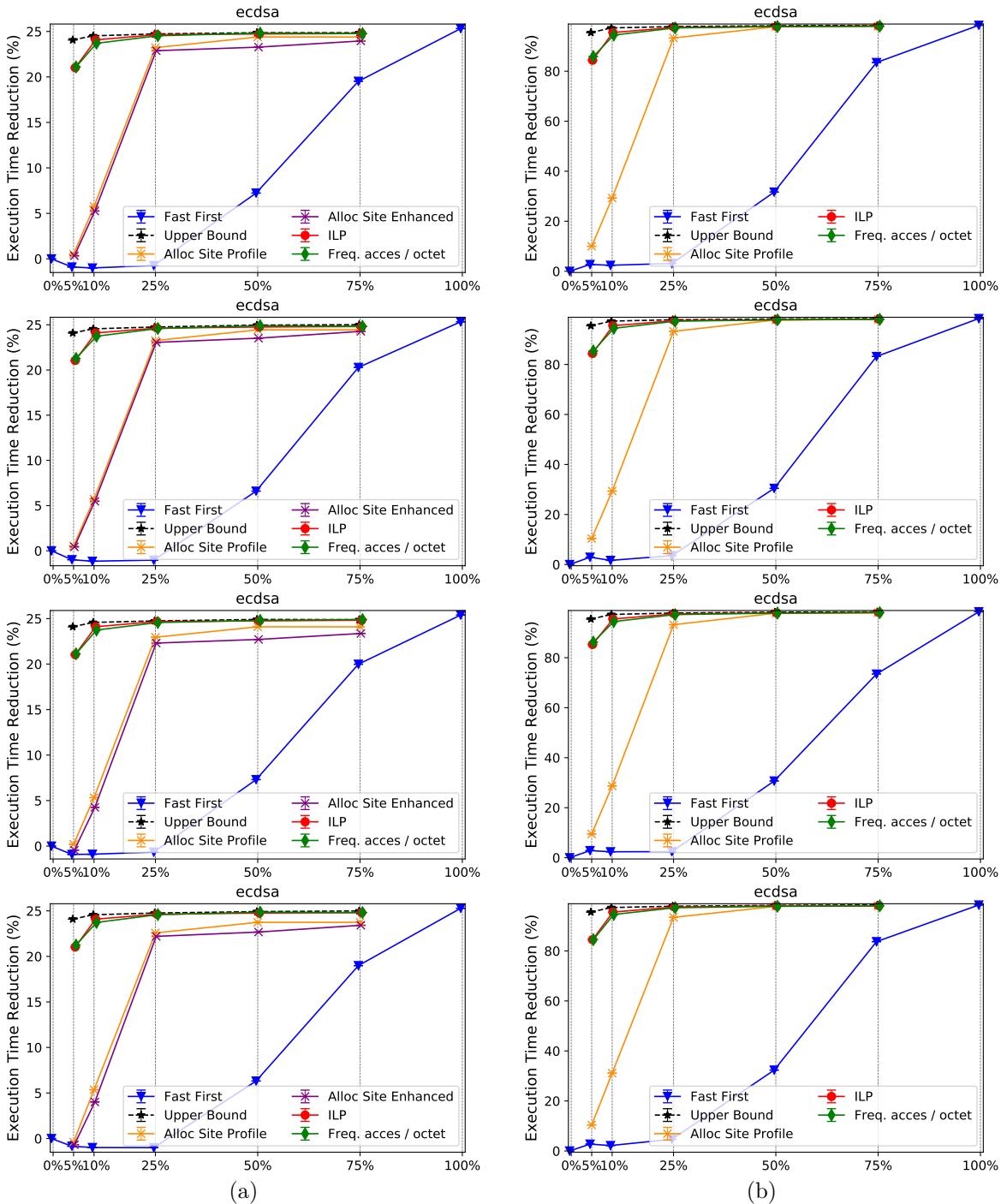


FIGURE C.3 – Ecdsa : Évaluation des performances – par jeu de données
 Technologies mémoires : (a) scénario "MRAM", (b) scénario "ReRAM"
 Jeux de données : 0 à 3 de haut en bas

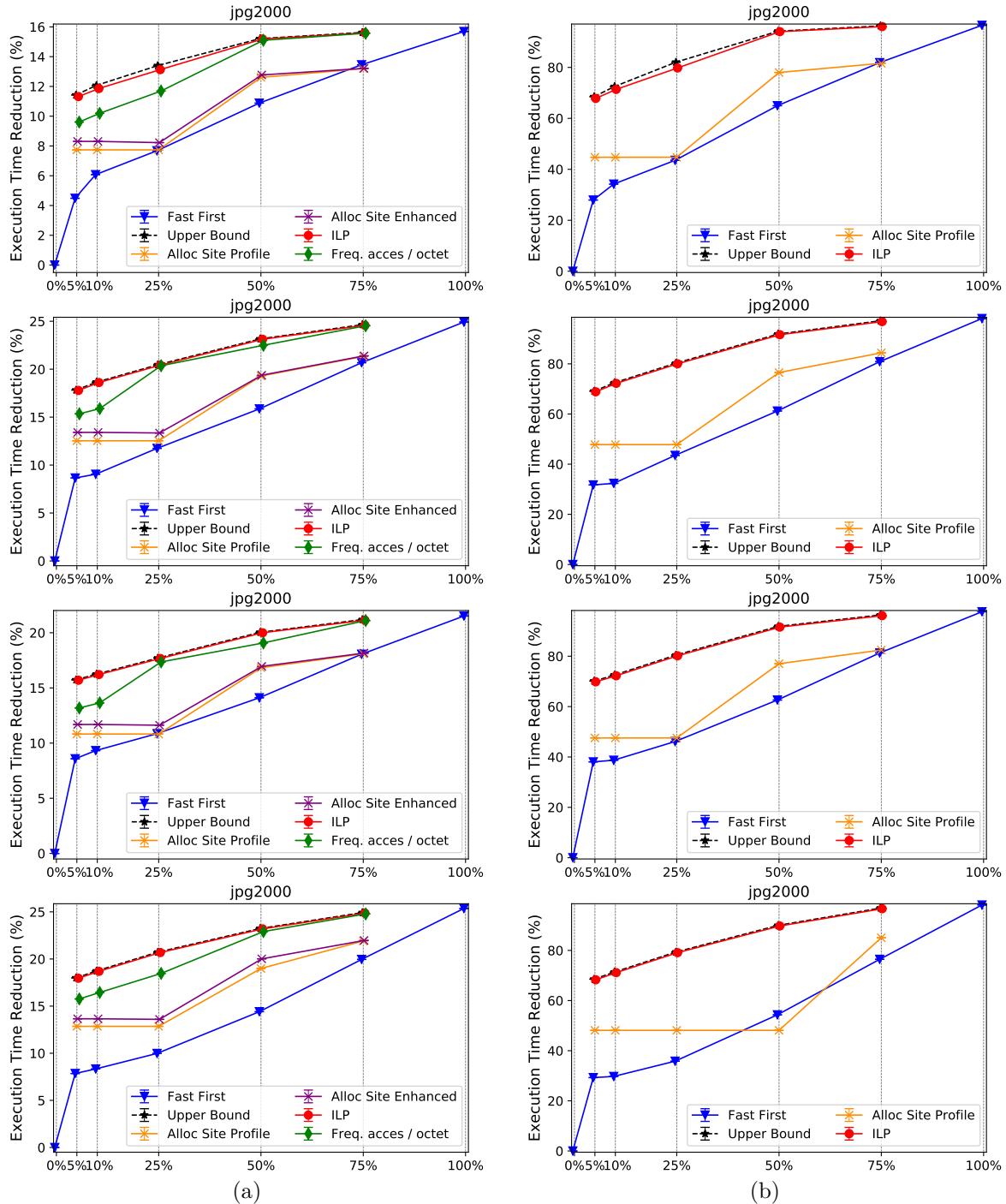


FIGURE C.4 – Jpg2000 : Évaluation des performances – par jeu de données
 Technologies mémoires : (a) scénario "MRAM", (b) scénario "ReRAM"
 Jeux de données : 0 à 3 de haut en bas

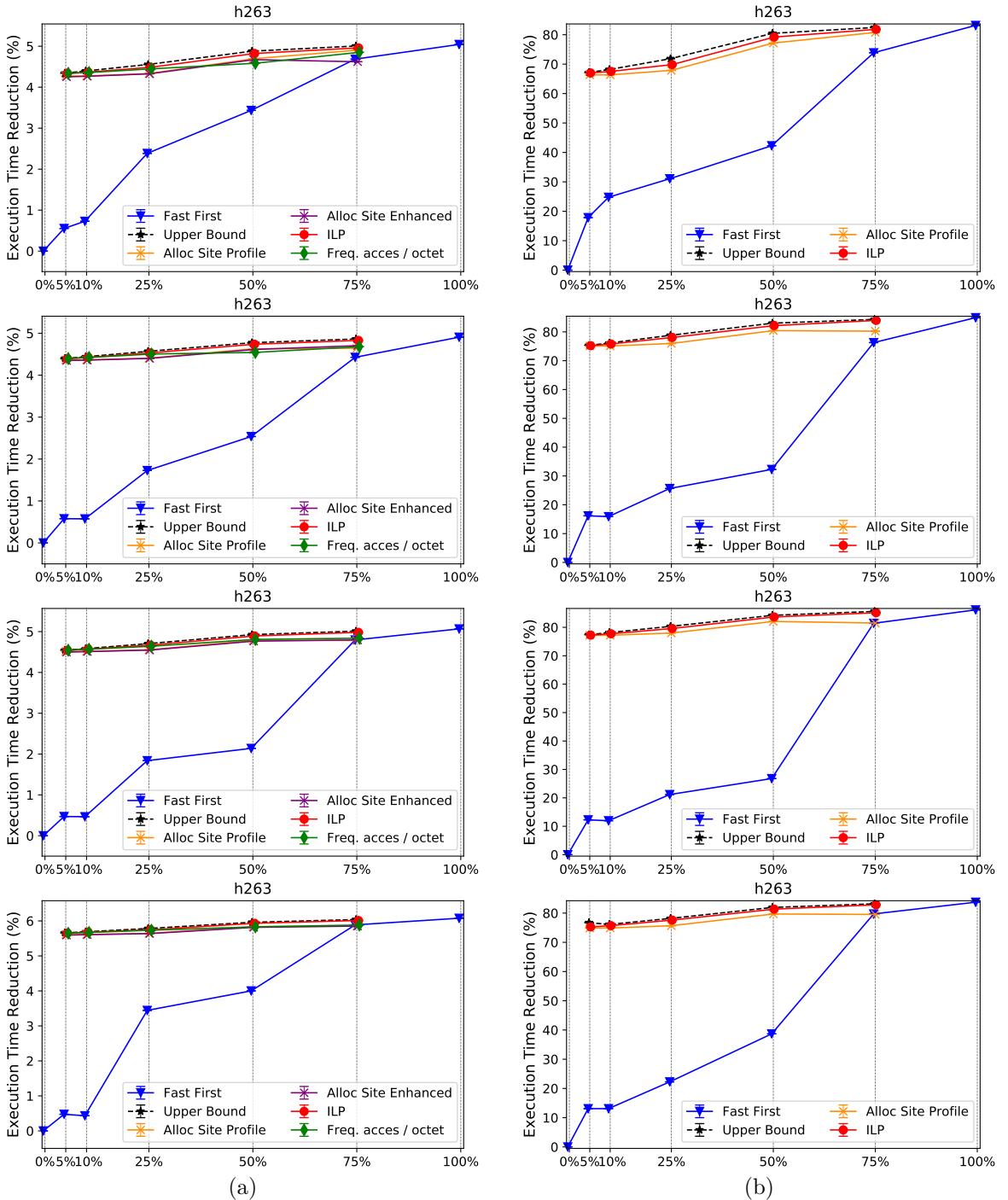


FIGURE C.5 – **H263 : Évaluation des performances** – par jeu de données
 Technologies mémoires : (a) scénario "*MRAM*", (b) scénario "*ReRAM*"
 Jeux de données : 0 à 3 de haut en bas

Index

- adressage virtuel, **31**, 48
- architecture de référence, **71**
- architecture idéale, **72**
- architecture rapide, **71**
- bare metal, **33**
- best fit, 24, **24**
- Board Support Package, **33**
- boundary tags, **24**
- conception conjointe matériel-logiciel, **30**, 71, 73, 76, 113
- développement sur machine nue, **33**
- deffered coalescing, **24**
- dispatcheur, 73, **73**, 102
- DLMalloc, **24**, 34, 53, 71, 106, 108
- DMA, **55**
- DRAM, **38**
- early fail, **79**, 102, 104
- EEPROM, **38**
- empreinte mémoire, **23**, 74, 81, 106 en ligne, **73**, 90, 94, 115, 117
- environnement d'exécution, **19**, 20
- exécution équivalente, **89**, *voir* exécutions équivalentes
- exécution de référence, **71**, 89, 102, 114
- exécution rapide, **71**, 110
- exécutions équivalentes, 70, **70**, 71, **89**, 117
- fallback, **73**, 105, 109, 121, 122
- Fast First, **102**, 110, 120
- first fit, **24**
- Flash, **38**
- gestion mémoire automatique, **22**, 56
- gestion mémoire manuelle, **22**
- Hardware Abstraction Layer, **33**
- hors ligne, **73**, 90, 101, 104, 106, 113–115, 117
- HPC, **55**, 75
- ILP, **54**, 133
- IoT, **30**, 81
- mémoire de stockage, **19**
- mémoire de travail, **19**, 66
- mémoire idéale, **39**, 67, 72, 107
- mémoire lente, **65**, 71, 74
- mémoire rapide, **65**, 71, 74
- MMU, **31**, 48, 50
- Normally-Off, **40**, 141, 142
- NUMA, **56**
- NVRAM, **40**
- NVRAMs, **40**, *voir* NVRAM, 54
- objets, **21**
- overlap, **89**
- pic, **23**
- placement mémoire, **62**
- plateau, **23**
- problème de placement, **62**
- profil d'accès, **61**, *voir* profil d'accès au tas, 81, 87, 90, 117, 129
- profil d'accès au tas, **61**, 70, 89, 101, 104, 112, 113
- profil d'allocation, **21**, 22, 23, 26, **54**, 61, 62, 81, 87, 89, 90, 101, 106, 117, 129, 132, 135, 136
- profils d'allocation, 54, **54**, *voir* profil d'allocation
- rampe, **23**
- runtime, **19**, 20, 21
- Scratch Pad, **31**, 47, 48, 51, 53, 54, 75–77
- segregated free list, **24**
- site d'allocation, **21**, **53**
- sites d'allocation, **53**, *voir* site d'allocation
- SPM, **31**, *voir* Scratch Pad, 48, 49, 51–55, 57, 76
- SRAM, **37**
- stack frame, **52**
- stratégie de placement, **71**
- stratégie ILP, **109**, 110
- stratégie ILP Upper Bound, **109**, 110
- système d'exploitation, **19**, 21
- tas, **21**
- tas lent, **66**, 71, 114
- tas rapide, **66**, 71, 114

tiny embedded, **34**

WCET, **30**, 55



INSA

FOLIO ADMINISTRATIF

THESE DE L'UNIVERSITE DE LYON OPEREE AU SEIN DE L'INSA LYON

NOM : DELIZY

DATE de SOUTENANCE : 19 / 12 / 2019

Prénoms : Tristan, Grégoire

TITRE : Gestion de la mémoire dynamique pour les systèmes embarqués avec mémoire hétérogène

NATURE : Doctorat

Numéro d'ordre : AAAALYSEIXXXX

Ecole doctorale : 512 - InfoMaths

Spécialité : Informatique

RESUME :

La réduction de la consommation énergétique des systèmes embarqué est un enjeu majeur de la réalisation de l'Internet des Objets. Les mémoires émergentes NVRAMs présentent notamment le potentiel de consommer peu et d'être denses, mais les différentes technologies souffrent encore de désavantages spécifiques comme une latence d'écriture élevée ou une faible endurance. Pour contrebalancer ces désavantages, les concepteurs de systèmes embarqués tendent à juxtaposer différentes technologies sur une même puce.

Cette thèse s'intéresse aux interactions entre l'allocation mémoire dynamique et l'hétérogénéité mémoire.

Notre objectif est de fournir au programmeur d'applications embarquées un mécanisme logiciel transparent pour exploiter cette hétérogénéité mémoire.

Nous proposons un simulateur au cycle près de plateformes embarquées intégrant des technologies mémoire variées qui montre que les stratégies de placement des objets alloués dynamiquement ont un impact important. Nous montrons également que des gains intéressants peuvent être dégagés même avec une faible proportion de la mémoire utilisant une technologie à faible latence mais uniquement en utilisant une stratégie intelligente pour le placement entre les différentes banques mémoires.

Nous fournissons une stratégie efficace basée sur le profilage de l'application dans notre simulateur.

MOTS-CLÉS :

Gestion mémoire dynamique, Systèmes embarqués, NVRAM, Architecture

Laboratoire (s) de recherche : CITI Lab – Centre of Innovation in Telecommunication and Integration of Service

Directeur de thèse : Tanguy Risset, Professeur des Universités, INSA de Lyon

Président de jury : Sentieys, Olivier, Professeur des Universités, Université de Rennes

Composition du jury :

Sentieys, Olivier, Professeur des Universités, Université de Rennes - Examinateur

Belleudy, Cécile, Maître de Conférences HDR, Université Nice Sophia Antipolis - Rapportrice

Torres, Lionel, Professeur des Universités, Université Montpellier - Rapporteur

Salagnac, Guillaume, Maître de Conférences, INSA de Lyon - Examinateur

Risset, Tanguy, Professeur des Universités INSA de Lyon - Directeur de thèse

Moy, Matthieu, Maître de Conférences HDR Université Claude Bernard Lyon 1 - Co-directeur de thèse