


Université des Sciences et Technologies de Lille

Gestion de mémoire à objets pour systèmes embarqués

Mémoire pour l'obtention d'une thèse de doctorat (spécialité Informatique)

par *Kevin Marquet*

Composition du jury

Président :	Michel Banâtre, ACES, Institut de recherche en informatique et systèmes aléatoires
Rapporteurs :	Michel Banâtre, ACES, Institut de recherche en informatique et systèmes aléatoires Daniel Hagimont, IRIT/ENSEEHT, Institut National Polytechnique de Toulouse
Examinateurs :	Laurent Lagosanto, GemAlto Systems Research Labs
Directeur :	David Simplot-Ryl, LIFL, Université des Sciences et Technologies de Lille
Co-Encadrant :	Gilles Grimaud, LIFL, Université des Sciences et Technologies de Lille

Ce document a été rédigé avec Emacs et mis en page avec LaTeX 2ε.

Remerciements

En premier lieu, je tiens à remercier les membres de mon jury pour avoir accepté d'y figurer. Merci donc à Michel Banâtre et Daniel Hagimont d'avoir accepté de rapporter cette thèse. Merci à Laurent Lagosanto d'avoir accepté d'en être l'examinateur.

Mon jury comptait également David Simplot-Ryl, mon directeur de thèse. Je tiens d'abord à le remercier pour m'avoir accepté au sein de l'équipe POPS, me permettant ainsi de découvrir le milieu de la recherche. Je le remercie d'avoir été mon directeur de thèse et pour ses conseils avisés quand je débutais.

Gilles Grimaud a été mon encadrant pendant mes trois années de thèse. Sa gentillesse, sa modestie et son humour ont été un facteur primordial dans le bon déroulement de mes trois années de thèse. Je le remercie pour les discussions qu'il est capable de mener en profondeur mais en toute simplicité, qu'elles concernent la recherche, les cthuuls, ou la ROMization de l'univers... Merci enfin de m'avoir toujours fait confiance et de m'avoir toujours laissé une grande part d'autonomie.

Je tiens à remercier mes petits camarades de la fac, pour avoir partagé à un moment ou à un autre mon bureau ou mon repas. Merci donc à Gnu, pour les coding, les South Park, et les thés gais. Je le remercie moins pour les inner-class*... Merci à Nadia pour les discussions, philosophiques ou non, pour la musique, et pour l'introduction (culinaire notamment) à la culture tunisienne. Merci à elle de m'avoir supporté dans son bureau sans se rendre elle-même trop insupportable. Merci à Antoine Gallais "le gaucho" pour les parties de basket, les manifs et les soirées. Merci également à lui de partager avec moi pas mal d'idées sur un peu tout... Merci à Sigma pour les randos, sa sincérité et, conséquence, les cassages en règle du midi. Merci à Dorina pour les relectures, la salsa, et les remplacements de cours.*

Je remercie également tous les membres de l'équipe (passés ou présents) que je n'ai pas encore cité : Jux, Hervé, Fadila, Jean-Jacques, Simon, François Ingelrest, Vincent, Isabelle, Christophe, Axelle. Tous ont permis de garder une ambiance de travail conviviale et agréable.

Merci enfin à mes relecteurs : ma mère, Amélie et Nadia. Je les remercie tout particulièrement de leur diligence...

Je me dois de remercier tous les amis qui auront su me motiver à quitter le labo, même en pleine session de debug de JITS (!).

Merci donc aux joueurs de badminton. Maud et Laurent pour les soirées et les crêpes-party ; Alexis pour les bandes, et le futur relooking ; Loïc pour les doubles mémorables et moins mémorables ; Aurélia pour le ski et le tournoi de Villeurbanne ; Nico, Shoun, Maud et Laurent pour le ski ; Georges pour nous avoir fait souffrir si efficacement pendant 3

ans ; Stef' pour sa continuelle bonne humeur. Merci à tous les autres joueurs : la petite Perrine, Alexandre, Boule et les autres.

Parmi les chtis qui se sont expatriés, je voudrais remercier Stephou et sa touffe de cheveux, pour l'alpinisme et le vtt, ainsi que Fredou pour les soirées et l'hébergement à Grenoble.

Merci également à tous ceux que j'ai peu vu pendant ma thèse mais à qui je pense régulièrement : Sam, Ève, Sarah... A bientôt je l'espère.

Une pensée toute particulière pour Clément, qui je le souhaite ne tardera plus à écrire cette même section pour son propre compte.

Un grand merci à tous mes frérots ; inutile de préciser pourquoi, ils savent bien pourquoi ils sont indispensables.

Amélie

Je conclue par un grand merci à mes parents pour leur présence dans toutes les situations importantes et plus où moins pénibles. Je pense toujours m'en sortir seul ; merci de m'avoir montré que j'avais tant besoin de vous.

Table des matières

Premier Chapitre

Introduction

« C'est à l'heure du commencement qu'il faut tout particulièrement veiller à ce que les équilibres soient précis. »

Franck Herbert,
Dune.

« Le commencement est beaucoup plus que la moitié de l'objectif. »

Aristote.

Le lecteur trouvera dans ce chapitre une introduction à la fois aux travaux menés et à la lecture du présent document.

Contexte

Les systèmes informatiques sont de plus en plus présents dans notre quotidien, sous la forme de téléphones portables, cartes à microprocesseur, ou autres PDA [1]. Ces appareils sont constitués d'un ensemble formé de matériel et de logiciel qui prend aujourd'hui le nom de « système embarqué », « système autonome » ou encore « informatique enfouie ». Si ces appellations sont différentes, elles expriment une même chose : ces systèmes informatiques sont particuliers. Ils ne correspondent pas à la vision que l'on peut avoir du conventionnel ordinateur personnel.

En effet, chacun de ces appareils est caractérisé une ou plusieurs fonctions précises et a été conçu dans ce but. En particulier, ils répondent, matériellement, à plusieurs besoins de l'utilisateur : communiquer avec d'autres appareils (par BlueTooth, Wifi, Wimax, etc.), fournir une connexion à l'Internet, savoir stocker diverses données, afficher une image ou encore jouer de la musique. Ces appareils respectent également des contraintes telles qu'une consommation énergétique faible, un faible coût de fabrication ou encore des dimensions réduites. Ces besoins et contraintes provoquent l'évolution rapide du matériel, afin de fournir toujours plus de fonctionnalités : persistance des données, tolérance aux pannes, sécurité des données, durée de vie, performances accrûes (plus de mémoire pour stocker, plus de puissance de calcul pour rendre plus de services, une taille plus petite pour tenir dans la poche).

Parallèlement à l'évolution matérielle, le monde du logiciel a lui aussi évolué, afin de fournir des technologies permettant le développement plus rapide d'applications plus sûres et plus performantes. Cependant, une grande partie de ces nouvelles technologies apparaît également comme étant plus complexes et ne s'applique aujourd'hui que sur des appareils au matériel standard, proche de l'ordinateur personnel. On peut citer l'utilisation de langages de haut niveau, ou encore l'utilisation de machines virtuelles (Java, .Net). Ces techniques constituent des solutions intéressantes, et sont utilisées massivement sur des machines relativement puissantes. Cependant, elles sont aujourd'hui inexploitées dans la majorité des systèmes embarqués, en raison de leur complexité et de la spécificité matérielle des appareils.

Orientation des travaux

Mes travaux s'inscrivent dans la cadre des problématiques plus générales adressées par l'équipe-projet POPS LIFL/INRIA, dans le cadre de l'élaboration d'outils intelligents de conception de systèmes embarqués. Ces problématiques concernent entre autres la mise au point de solutions générales pour l'utilisation de machines virtuelles sur matériels contraints. Le but de mon travail est de proposer de telles solutions en ce qui concerne la gestion de la mémoire.

Depuis de nombreuses années, des mécanismes efficaces de gestion de la mémoire, tels que l'allocation et la libération automatique des données, ont été mis au point et sont cou-

1. Personnal Digital Assistant.

ramment utilisés dans l'informatique généraliste. Toutefois, aujourd'hui, la gestion de la mémoire reste manuelle et spécifique à chaque système embarqué. Les spécificités des systèmes embarqués, tant matérielles que logicielles, sont au cœur des problèmes qui rendent difficiles l'intégration des mécanismes récents de gestion de mémoire. Dans ce contexte, l'objectif général est de fournir des solutions de gestion mémoire pour l'embarqué aussi générales qu'elles peuvent l'être sur les machines traditionnelles. Plus particulièrement, les travaux présentés dans ce document concernent la gestion de mémoire *à objets*, principalement associée à l'utilsation de language de haut niveau comme Java ou C# qui connaissent actuellement un essor important et rencontrent un franc succès.

Cette problématique générale de gestion de la mémoire a été décomposée en trois étapes :

1. Identifier les obstacles à la mise en œuvre d'une telle manière de gérer la mémoire.
2. Revoir les études théoriques existantes concernant la gestion de mémoire à objets, en prenant en compte les spécificités des systèmes embarqués.
3. Proposer et évaluer des solutions aux divers problèmes identifiés.

Je me suis attaché au cours de ma thèse à garder ces propositions les plus indépendantes possibles : tout d'abord, indépendantes d'hypothèses que l'on pourraient juger trop fortes ; indépendantes de la plateforme d'expérimentations également ; et enfin indépendantes entre elles. La forme de ce document témoigne de cette dernière intention. Ainsi, les différentes évaluations réalisées ne sont pas réunies en un seul chapitre. De la même manière, la plupart des références aux travaux existants sont faites au début de chaque chapitre.

Sommaire

Quatre sous-problèmes ont été traités durant ma thèse :
- l'étude des mécanismes de ramasse-miettes pour l'embarqué [Marq 07c] ;
- la mise au point d'une nouvelle architecture de gestionnaire mémoire, basée sur une approche novatrice [Marq 07d, Marq 07e] ;
- la gestion poussée du placement des données [Marq 07a, Marq 07b, Marq 05b] ;
- le déploiement des systèmes embarqués à objets [Marq 05a, Grim 05].

Ce document décrit les problèmes et solutions apportées, au travers, outre cette introduction, les six chapitres suivants.

Le **chapitre 2** détaille la problématique adressée par mes travaux. Les différents sous-problèmes y sont présentés. Les particularités des systèmes embarqués y sont détaillées, ainsi que les difficultés de la gestion de la mémoire pour ces systèmes embarqués.

Le **chapitre 3** présente une étude de la complexité des algorithmes de gestion automatique de la mémoire. Cette étude considère particulièrement les spécificités de systèmes embarqués. Les résultats et conclusions tirés de cette étude guident partiellement la mise au point des solutions présentées dans la suite.

Le **chapitre 4** détaille une nouvelle architecture de gestionnaire mémoire permettant d'adresser les difficultés relatives à l'utilisation d'algorithmes de gestion automatique de la mémoire dans le monde de l'embarqué.

Le **chapitre 5** décrit une solution orientée langage au problème du placement des données. Il détaille le langage dédié mis au point, décrit le processus de compilation de ce langage ainsi que le fonctionnement à l'exécution du code généré.

Le **chapitre 6** traite du problème du déploiement du système embarqué, du point de vue de la gestion mémoire. Des optimisations y sont présentées, ainsi que l'implantation des solutions proposées dans un véritable système embarqué.

Enfin, le **chapitre 7** résume les contributions des travaux , en aborde les limites, et ouvre des perspectives aux solutions proposées.

Deuxième Chapitre

Problématique

« Il y a bien moins de difficultés à résoudre un problème qu'à le poser. »

Joseph de Maistre.

Alors que le chapitre précédent a introduit le sujet de ce mémoire de thèse, le lecteur trouvera dans celui-ci un développement de la problématique adressée. L'architecture matérielle très particulière des cibles visées est tout d'abord présentée afin de bien comprendre quelles peuvent être les difficultés relatives à la mise au point de logiciels gérant de tels appareils. Ensuite, les solutions logicielles sont détaillées afin de spécifier le contexte dans lequel on se situe. Enfin, les problèmes liés à la gestion mémoire dans le cadre donné sont exposés.

2.1 Cibles visées

Les travaux présentés dans la suite de ce document concernent une certaine classe d'appareils. Les spécificités de ces appareils sont maintenant présentées.

2.1.1 Matériel embarqué

Un grand nombre d'appareils peuvent être désignés comme *systèmes embarqués*, à tel point qu'il est difficile de donner une définition précise de ce terme. On peut cependant s'appuyer sur la littérature [Gajs 94, Gans 03, Edwa 03] pour en obtenir des caractéristiques plus ou moins communes :

Autonome Un système embarqué dispose de l'ensemble des éléments physiques nécessaires à son fonctionnement (processeur, mémoire, périphériques d'entrées/sorties), parfois même de sa source d'énergie limitée, et est destiné à fonctionner sans administration ou parfois même sans mise à jour courante ;

Ressources limitées Un grand nombre d'appareils ont des ressources limitées, en terme de puissance de calcul, énergie ou encore mémoire, à cause de leur petite taille ou de leur fonction ;

Dédié à une ou plusieurs tâches précises Un système embarqué est généralement conçu pour une ou plusieurs tâches précises. La conception de tous ses composants est ainsi orientée dans un but précis, ce qui peut mener à l'utilisation de matériels très variés.

Conceptions logicielle et matérielle fortement couplées Les composants matériels mais aussi logiciels sont choisis afin de servir au mieux les tâches auxquelles est destiné un système embarqué. Les contraintes matérielles et logicielles mènent à un modèle de conception où ces deux parties sont fortement couplées.

Les systèmes correspondant à ces critères sont nombreux, et répondent à des besoins variés. Ce type d'appareil va des systèmes gérant le pilotage d'avions jusqu'aux cartes à puce, en passant par les lecteurs de musique ou encore les capteurs de terrain.

En ce qui concerne les appareils grand public, la miniaturisation des composants électroniques mène d'ailleurs à une généricité des usages des nouveaux appareils. On voit ainsi apparaître des combinés PDA/lecteur/téléphone ou encore des montres/lecteur MP3.

Les équipements qui nous intéressent sont principalement les *Petits Objets Portables et Sécurisés* (POPS). Ces appareils vont, en taille, des PDAs aux cartes à puce, en passant notamment par les capteurs de terrain. Ce type d'appareil se distingue en premier lieu

par sa configuration matérielle très particulière d'une part, et très hétéroclite d'autre part. En effet l'appareil étant dédié à une ou plusieurs tâches précises, le matériel est choisi de façon à effectuer ces tâches au mieux, donnant ainsi un profil très particulier à l'ensemble. Une configuration typique comprend donc un ensemble de composants plus ou moins dédiés. Le tableau 2.3 montre quelques exemples d'appareils illustrant la classe de matériel visée par nos travaux.

Microprocesseur L'unité de calcul choisie dépend grandement de la fonction de l'appareil. Les critères de choix les plus importants sont la puissance de calcul, la consommation énergétique et le coût. Le processeur est traditionnellement relié à d'autres composants externes à la puce via des bus. Cependant, afin de réduire les coûts, un microcontrôleur peut être utilisé, regroupant sur la même puce un processeur, un ou plusieurs types de mémoire et des ports d'entrées/sorties, reliés par un bus d'adresses et un bus de données. Parmi les processeurs très répandus, on trouve ceux de la famille ARM, en particulier l'*ARM7tdmi*. Du côté des microcontrôleurs très utilisés on peut citer l'AVR d'Atmel ou encore les PIC de la Sociéte Microchip. À noter que dans le choix du processeur, deux approches s'opposent quant à l'augmentation des performances. Soit augmenter la puissance du processeur, soit ajouter des coprocesseurs dédiés (cryptographie, décompression de données) en laissant un processeur minimaliste pour réduire la consommation énergétique et les coûts.

Périphériques Un certain nombre de modules externes peuvent faire partie de l'appareil. On trouvera ainsi éventuellement un écran (appareils photographiques), micro, sortie son (lecteur MP3), clavier (PDA). Cependant, ce type de périphériques n'est pas forcément nécessaire, les cartes à puce, étiquette électronique et certains capteurs, entre autres, fonctionnant de manière non interactive. Cependant, un module de communication est généralement nécessaire : un téléphone portable va relayer les appels grâce à sa connection sans fil, un capteur de terrain va transmettre les données mesurées via un module radio spécifique (utilisant un protocole 802.15 par exemple), une carte à puce va communiquer avec son lecteur par le biais d'une liaison série.

Mémoire Une des spécificités des systèmes embarqués est qu'ils incluent différents types de mémoires. De plus, les types de mémoires utilisés peuvent être très différents d'un matériel à l'autre. La section suivante détaille plus avant les plus courants des différents types de mémoires embarquées.

2.1.2 Mémoires embarquées

Tout d'abord, un grand nombre de mémoires différentes existe. On ne les trouve pas sur du matériel générique comme un ordinateur personnel. Cependant, on a vu que la spécialisation des systèmes embarqués dans une ou plusieurs tâches précises demandait l'utilisation de composants matériels dédiés à ces tâches. Les mémoires utilisées sont concernées de la même façon que les autres composants, puisqu'elles diffèrent grandement dans leurs propriétés :

Propriété de modification La plupart des mémoires sont réinscriptibles. Cependant, certaines ne le sont pas (ROM, PROM), elles ne peuvent être initialisées qu'une

Équipement	Type	Processeur	Mémoire	Périphériques
Carte Platinium	Carte à puce	Microcontrôleur Atmel AT90SC6464C	64 Ko EEPROM, 64 Ko Flash, 3 Ko RAM	Ligne série 625 Kbps
MICAz	Capteur	Microcontrôleur Atmel ATmega128L	128 Ko Flash, 4 Ko EEPROM, 4 Ko SRAM	Émetteur radio à 250 Kbps
Zaurus SL-5500	PDA	Strong ARM 206 MHz	64 Mo RAM, 16 Mo Flash, ROM	Clavier, écran, audio, liaison réseau USB ou 802.11b

TABLE 2.1 – Les systèmes embarqués couvrent une gamme très variée de matériels.

seule fois. D'autres mémoires ne peuvent l'être que d'une certaine manière, en utilisant un matériel spécial ;

Propriété de persistance Certaines mémoires telles l'EEPROM n'ont pas besoin d'être alimentées pour conserver les données. Les données contenues dans une telle mémoire sont rendues persistantes ;

Vitesse d'écriture Les performances en lecture peuvent être extrêmement variables d'une mémoire à l'autre. Il existe ainsi un facteur de plusieurs milliers entre la SRAM (Static RAM) et la mémoire Flash.

Vitesse de lecture De plus, les vitesses en lecture et en écriture peuvent être complètement différentes, comme pour l'EEPROM ;

Latence La mémoire Flash par exemple peut demander un certain temps de chargement de page, rendant ainsi l'accès au premier octet d'une page plus lent que la lecture des suivants ;

Granularité d'accès Certains types de mémoires comme les différentes RAM peuvent accéder aléatoirement à n'importe quel octet, d'autres ne peuvent lire et/ou écrire qu'une page complète ; dans ce dernier cas, la page doit être chargée en mémoire réinscriptible, puis modifiée, et enfin re-écrite ;

Point mémoire L'occupation d'une mémoire en terme de $bits/mm^2$ est très variable d'une mémoire à l'autre. Ce facteur limite grandement l'utilisation de RAM sur les très petits appareils tels que les cartes à puce ;

Consommation électrique Un grand nombre d'appareils ont des contraintes plus ou moins fortes en terme de consommation énergétique, ce qui peut rendre impossible l'utilisation d'une mémoire consommant beaucoup ;

Durée de vie L'écriture abîme certaines mémoires dans une certaine mesure. Elles ont alors une durée de vie limitée, que l'on exprime en nombre d'écriture/effacement ;

Coût Le prix est un facteur important qui empêche certaines mémoires pourtant très efficace d'être utilisées par les industriels.

La table 2.2 indique les caractéristiques des mémoires les plus couramment utilisées dans les systèmes embarqués. On y trouve les principales caractéristiques des mémoires suivantes :

ROM *Read-Only Memory* Mémoire non réinscriptible bon marché utilisée principalement pour stocker l'image initiale du logiciel gérant un appareil ;

PROM *Programmable ROM* La PROM peut être modifiée une fois. Elle a le même type d'utilisation que la ROM ;

EPROM *Erasable PROM* L'EPROM est utilisée de la même manière mais peut, elle, être modifiée après son initialisation, ce qui offre la possibilité par exemple de mettre à jour le système entier ;

EEPROM *Electrically Erasable PROM* L'EEPROM peut elle être lue et écrite par le programme s'exécutant sur l'appareil. Cela ajouté à sa faible empreinte mémoire et sa propriété de persistance des données en fait une mémoire de choix, même si elle a des limitations en écriture (très lente, par page, nécessite un accesseur particulier) ;

RAM *Random Access Memory* La RAM est une mémoire rapide pouvant accéder directement à n'importe quelle donnée. Différentes sortes de RAM existent :

- **DRAM** *Dynamic RAM* Elle nécessite un rafraîchissement périodique (toutes les quelques millisecondes) afin de conserver ses données. Plusieurs sortes de DRAM existent. Les détailler ici ne servirait que peu mon propos ; je me contenterai de les mentionner : FPM DRAM, EDRAM (combinaison de SRAM et DRAM), EDO DRAM (plus rapide que la DRAM standard). La DRAM est utilisée comme mémoire de travail dans la majorité des systèmes informatiques, même si son encombrement fait qu'elle n'est présente qu'en (très) faible quantité dans les appareils contraints,
- **SRAM** *Static RAM* Ce type de RAM est très rapide mais aussi volumineux (environ quatre fois plus que la DRAM) et coûteux. Elle est principalement utilisée en petite quantité pour servir de cache de données ;

Mémoire *Flash* Cette mémoire est non volatile et réinscriptible par blocs. Elle est de plus en plus utilisée comme mémoire de masse. Deux types de Flash peuvent être distinguées :

- **Flash NOR** La lecture dans cette mémoire est similaire à la lecture dans la RAM, ce qui permet entre autres de pouvoir exécuter sans relocation un code stocké dans une telle mémoire,
- **Flash NAND** Au contraire, tous les accès se font par page dans la Flash NAND, ce qui rend beaucoup plus difficile son utilisation comme mémoire de travail. Cependant sa vitesse d'écriture est supérieure à celle de la Flash NOR.

La table 2.3 présente les configurations mémoire de quelques appareils et composants matériels. Elle illustre bien l'hétérogénéité des mémoires que l'on peut trouver, d'un appareil à l'autre d'une part, sur le même matériel d'autre part.

Types de mémoires annoncés De nombreux types de mémoires existent donc, et sont utilisés conjointement. Cette tendance ne semble pas devoir s'inverser. On voit en effet apparaître sans cesse de nouveaux types de mémoires, aux caractéristiques sans cesse améliorées, et toujours variables. Aujourd'hui, certaines technologies émergent peu

Type	Persistente	Réinscriptible	Vitesse d'accès	Granularité	Durée de vie	Coût	Point
ROM	Oui	Non		Octet	Durée de vie	Très faible	*base*
PROM	Oui	Non	n/a	Octet	Illimitée	Faible	x 1-4
EPROM	Oui	Oui (spécial)	n/a	Octet	Millions of write/erase	Faible	x 1-4
EEPROM	Oui	Oui	Lecture : 100ns, écriture : 4 ms	Lecture : octet, Écriture : 1→4 octets	Millions of write/erase	Élevé	x 4
SRAM	Non	Oui	few ns	Octet	Illimitée	Élevé	x 50
DRAM	Non	Oui	10-60 ns	Octet	Illimitée	Moyen	x 20
Flash NOR (x16)	Oui	Oui	Lecture : 50μs/page (103 Mo/s), Écriture : 900ms (0.5 Mo/s)	Lecture : octet, Écriture : page (512→2048 octets)	Hundreds of thousands of Write/Erase	Moyen	x 2-3
Flash NAND (x8)	Oui	Oui	Lecture : 100ns/page (20 Mo/s), écriture : 2ms/page (8 Mo/s)	Page (512→2048 octets)	Hundreds of thousands of Write/Erase	Moyen	x 2-3

TABLE 2.2 – Caractéristiques principales de différentes mémoires

Appareil	Mémoires persistantes	Mémoires non persistantes
Excalibur	3x8 Mo Flash	4x32 Mo SDRAM, 2 Mo SSRAM
MICAz	128 Ko Flash en deux sections, 4 Ko EEPROM, 1 Mo Flash série, mémoire Flash de démarrage	4 Ko SRAM interne, 64 Ko mémoire externe
GameBoy Advance	ROM, cartouches d'EEPROM, SRAM ou Flash	32 Ko IWRAM (Internal Working RAM), 256 Ko EWRAM (External WRAM)
WSN430	1 Mo Flash, 48 Ko Flash	10 Ko RAM
Carte platinium	64 Ko EEPROM, 64 Ko Flash	3 Ko RAM

TABLE 2.3 – Configuration mémoire de différents appareils

à peu, même si elles semblent mettre trop de temps à s'imposer comme des remplacements viables. Ainsi, la Z-RAM (*Zero Capacitor RAM*) est une remplaçante potentielle de la SRAM offrant une meilleure densité. La FeRAM (*Ferroelectric RAM*) semble être prometteuse, en affichant des performances comparables à celles de la DRAM mais en étant persistantes. La MRAM (*Magnetoresistive RAM*) est similaire à la DRAM mais sans nécessiter de cycles de rafraîchissement car stockant les données sur des éléments magnétiques.

De nouveaux types de mémoires sont également annoncés par des sociétés désireuses d'améliorer leurs produits. La RRAM (*Resistive Random Acces Memory*) en ce moment développée par la société Sharp est annoncée comme ayant des vitesses d'accès 1000 fois plus rapides. Samsung a dévoilé un prototype de OneDRAM, censé consommer 30% d'énergie en moins et nécessitant 50% moins d'espace que la SRAM. Toshiba a, elle, annoncé être capable de fabriquer de la mémoire Flash NAND plus dense en organisant les cellules sur 3 dimensions. Ensuite, la PRAM (Phase-Change RAM) est une RAM non volatile qui permettrait une modification des données octet par octet, et non par page, tout en offrant des performances meilleures. Enfin, la société Nantero travaille sur la mémoire NRAM (*Nano-RAM*), un autre type de RAM non volatile.

Tout cela ne préfigure pas la fin de la coexistence de mémoires aux propriétés différentes, chaque nouvelle mémoire annoncée ayant également ses propres avantages et ses propres défauts par rapport aux autres. De nouveaux standards n'émergent pas pour le moment, ces nouvelles mémoires faisant face à des difficultés de mise au point et à des coûts rédhibitoires.

Au vu des spécificités du matériel embarqué, il s'est révélé nécessaire de mettre au point, pour gérer le mieux possible ce matériel, des méthodes adaptées à la conception d'un logiciel embarqué.

2.2 Logiciel embarqué

Les systèmes embarqués sont de plus en plus complexes. Il en résulte un fort besoin d'évolution des méthodes de programmation des logiciels embarqués.

2.2.1 Programmation des logiciels embarqués

Avant l'apparition du concept de système d'exploitation vers 1965, le matériel informatique était géré par des bibliothèques simples, écrites en même temps que le développement des applications et du matériel. Cependant, une telle approche rend très difficile le développement de systèmes complexes car le système embarqué doit être conçu dans son ensemble, d'un seul coup. Les temps de conception sont ainsi très longs.

Cette approche a été grandement améliorée avec l'apparition de méthodes de *codesign* matériel/logiciel. Le principe de base de ce codesign est basé sur la modélisation haut-niveau de spécifications à partir de laquelle à la fois les parties logicielle et matérielle vont être synthétisées.

L'approche opposée est l'utilisation, sur le matériel embarqué, d'une couche logicielle servant de système d'exploitation. Les applications s'exécutent alors au dessus de cette couche dont le rôle est de :

Fournir une abstraction du matériel L'objectif est de fournir au concepteur une vue agréable et fonctionnelle du matériel. Par exemple, les machines virtuelles fournissent une abstraction du processeur.

Gérer le matériel L'utilisation du matériel nécessite généralement des traitements annexes à ce qui lui est demandé. Par exemple, écrire en mémoire Flash n'est pas une opération élémentaire du microprocesseur, et requièrt l'utilisation d'une routine d'écriture spécifique qui va piloter les accès en Flash. Ce type de considérations matérielles est pris en charge par le système de telle sorte qu'elles n'incombent pas aux applications embarquées.

Sécuriser l'accès au matériel Le système empêche que l'activité d'une application nuise au fonctionnement d'une autre de par l'utilisation du même support matériel. Par exemple, les données allouées par une application ne doivent pas écraser les données utilisées par une autre.

Plusieurs modèles de systèmes d'exploitation existent. On citera tout d'abord les noyaux monolithiques, pensés et construits comme un tout. Les micro-noyaux [Gued 91] quant à eux cherchent à avoir des fonctionnalités minimalistes, tout en fournissant des moyens d'étendre facilement ce micro-noyau. Les services tels que le gestionnaire mémoire se retrouvent alors comme des services extérieurs à ce noyau. Enfin, les exo-noyaux [Engl 98] ne fournissent pas d'abstraction au matériel mais sécurisent son accès à grains fins.

Cependant, dans le monde des systèmes embarqués, la frontière entre une bibliothèque dédiée à la gestion du matériel et un système d'exploitation est assez floue. En effet, un ensemble de routines permettant d'accéder au matériel de façon sécurisée peut être considéré comme un système d'exploitation minimaliste.

Aujourd'hui, alors que le matériel et les besoins évoluent constamment, le temps de conception des systèmes embarqués a fortement diminué. Ceci a fortement augmenté les nécessités de vitesse de développement, de portabilité, de maintenabilité, et surtout de

réutilisation des logiciels embarqués. Les concepteurs de systèmes embarqués s'aident d'outils spécialisés afin de faire le pont entre les applications demandées et le matériel disponible. Cependant, les systèmes ne cessent de se complexifier. Une forte évolution des techniques de programmation a permis d'adresser ces problèmes dans les systèmes non embarqués. Une des techniques qui a fait ses preuves, qui est maintenant utilisée partout dans le monde, mais qui pose encore des problèmes en environnement contraint est la programmation orientée objet.

2.2.2 Programmation orientée objet

Apparue après la programmation assembleur et la programmation impérative, la programmation par objet constitue un pas de plus dans la capacité du langage à s'abstraire du langage machine. Ce sont le langage Simula-67 [Dahl 68], et surtout par la suite Smalltalk [Gold 83] qui ont marqué la naissance de ce nouveau paradigme de programmation. La programmation par objet consiste en la définition et l'assemblage de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Chaque objet possède un état défini par un ensemble de valeurs de champs, et a certains comportements. On peut déclencher ceux-ci par envoi de *message* et passage de paramètres. La formalisation des concepts liés à ce paradigme a eu lieu bien après Smalltalk. Ces concepts généraux sont les suivants :

Réification La résolution d'un problème demande en premier lieu une opération d'abstraction. En programmation orientée objet, la réification est l'opération qui consiste à sélectionner les éléments du problème pour leur donner le statut d'objets.

Polymorphisme Chaque objet peut être vu de différentes manières, ce qui permet de lui attacher plusieurs comportements différents, et ainsi de définir très simplement les différents comportements d'un programme. Le polymorphisme est notamment permis par l'**héritage** qui permet de réutiliser le comportement d'une classe d'objets tout en le spécialisant ;

Encapsulation Permet de séparer les interfaces d'un programme — ou une partie d'un programme — de son implémentation ;

Agrégation/composition Une autre façon de faire de la réutilisation est de construire des objets à partir d'autres objets.

Les concepts objets permettent une grande réutilisation. En effet, il est premièrement possible de factoriser des comportements communs (héritage). Deuxièmement, il est possible d'étendre et de sécuriser l'accès à un comportement en agrégeant/composant plusieurs classes d'objets. Et enfin, il est possible de spécialiser un comportement (héritage ou agrégation). Or on a vu que la réutilisation est primordiale dans le domaine de l'embarqué.

Ces avantages, en plus de la lisibilité et de la maintenabilité des langages orientés objet, ont mené à l'apparition d'un nombre impressionnant de langages supportant ce paradigme de programmation :

- langages purement objet, pour lesquels tout est objet : Smalltalk, Eiffel, Ruby ;
- langages orientés objet (tout n'est pas objet à 100%) : Python, Java ;
- langages étendus avec des propriétés objets : C++, Perl.

On notera le succès considérable de Java qui a facilité l'écriture de programmes, en fournissant un langage de classes orienté objet, compilé en bytecodes indépendants du matériel (utilisation d'une machine virtuelle). Plus récemment, le langage C# a été créé ; il a, à peu de choses près, les mêmes caractéristiques que Java et connaît un succès grandissant.

Le développement des technologies en liaison avec le modèle de programmation objet a également amené d'autres avantages, notamment en ce qui concerne la gestion de la mémoire. En effet, l'allocation et la libération explicite des données sont une source d'erreurs non négligeable. Ces erreurs peuvent mener à des références pointant vers une donnée qui a été écrasée car libérée trop tôt, causant parfois des pannes informatiques dont la cause est très difficile à déterminer. Des fuites de mémoire peuvent également être provoquées si certaines données ne sont pas libérées, provoquant éventuellement le remplissage très rapide de la mémoire. Rovner a estimé que 40% du temps de développement du système Mesa [Rovn 85] était passé à la résolution de problèmes liés à la gestion de la mémoire. L'allocation et la libération automatique des objets simplifient grandement les programmes et éliminent une préoccupation de bas niveau, contribuant de ce fait à s'abstraire plus facilement du matériel. Pour cela, un grand nombre de techniques ont été développées pour la gestion de mémoires à objets.

2.2.3 Mémoires à objets

L'objet est l'unité de base, en terme d'utilisation de la mémoire, des programmes écrits dans un langage orienté objet. De plus, cette unité de base n'est pas vue comme un simple espace mémoire que le programmeur peut allouer lui-même puis remplir comme il l'entend (langage C par exemple). Il s'agit au contraire d'une entité ayant une certaine utilité définie en partie par un type (une classe en Java), et dont le programmeur ne se soucie ni de l'allocation, ni de sa libération. Un programme va ainsi créer au fur et à mesure de son exécution un certain nombre d'objets, de taille arbitraire dans le cas général, qui seront alloués dans une mémoire à objet, ou *tas*. Au sein du tas, des références se créent entre les objets, au grès des affectations de champs qu'effectue le programme. A un instant donné de l'exécution d'un programme, la structure du tas peut être représentée sous la forme d'un graphe dont les noeuds sont les objets et les arcs représentent une référence d'un objet à l'autre. La figure 2.1 illustre la composition d'un tas d'objets et la représentation que l'on peut en avoir sous forme de graphe.

Il est à noter que le support du paradigme de programmation orientée objet conduit le système de gestion de la mémoire sous-jacent à ajouter des informations à chaque objet (pointeur vers la classe par exemple), amenant ainsi à la présence d'une certaine proportion de méta-données ; ces méta-données peuvent d'ailleurs être considérées comme des objets ordinaires. Cependant, l'organisation de la mémoire selon un modèle à objets n'est pas réservée aux langages à objets. Ainsi, on citera le langage Lisp qui organise ses données selon ce modèle sans pour autant être un langage à objets.

La gestion automatique d'une mémoire à objets est principalement liée à un composant bien particulier : le ramasse-miettes, chargé de récupérer automatiquement la mémoire. Cette dénomination est la plus courante. Dans la suite, on utilisera également l'abréviation GC [1], pour glaneur de cellules.

1. Historiquement, GC désigne en anglais un *Garbage Collector*

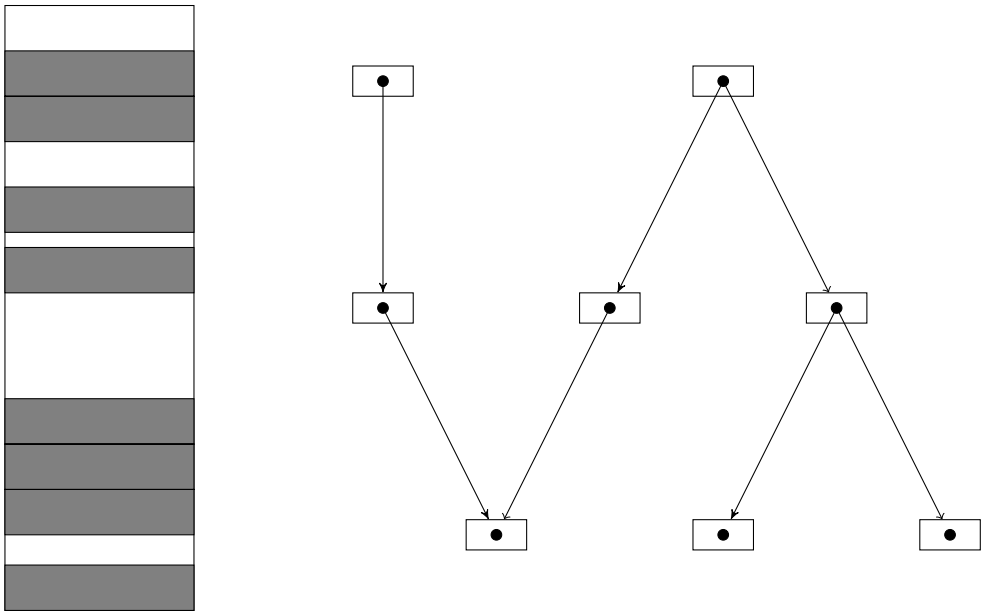


FIGURE 2.1 – Mémoire à objets (tas)

Le problème de la collecte automatique de données est toujours le même : identifier les blocs de mémoire préalablement alloués n'étant plus utilisables. Un bloc de données n'est plus utilisable quand le programme ne possède plus aucune référence vers ce bloc. Dans le graphe d'objets, cela correspond à un objet pointé par aucun arc.

Il existe un certain nombre de *racines* du graphe d'objets ; une racine est une référence vers un objet que le programme sait devoir conserver. Typiquement, en Java par exemple, une référence contenue dans la pile d'appels. Tout objet utile au programme est atteignable depuis ces racines. Un objet est dit atteignable s'il est référencé par une racine ou s'il est référencé par un autre objet atteignable. L'ensemble des objets encore utile au programme est donc la clôture transitive de l'ensemble des racines à travers la relation de référencement d'un objet par un autre. La figure 2.2 illustre ce mécanisme.

Un grand nombre d'algorithmes de ramasse-miettes existe [Wils 92]. Le premier à avoir été imaginé est un algorithme très simple : le compteur de références [Coll 60]. Chaque objet contient un compteur de références indiquant le nombre de références existant vers cet objet. Ce compteur est incrémenté chaque fois qu'une référence vers l'objet est créée quelque part dans le tas, et décrémentée quand une référence vers lui est écrasée par une autre. Si ce compteur tombe à zéro, l'espace occupé par cet objet peut être libéré.

Ce système préserve les objets atteignables. En effet, d'après la définition donnée, tout objet atteignable est un objet référencé, soit par une racine soit par un autre objet ; par conséquent le compteur de références ne peut être nul. Cependant, il préserve aussi certains objets qui ne sont pas atteignables. Par exemple, ceux référencés par un objet qui n'est pas référencé lui-même. Ainsi, cet algorithme est incapable de détecter les cycles dans le graphe d'objets. De plus, il nécessite de mettre à jour les compteurs de références

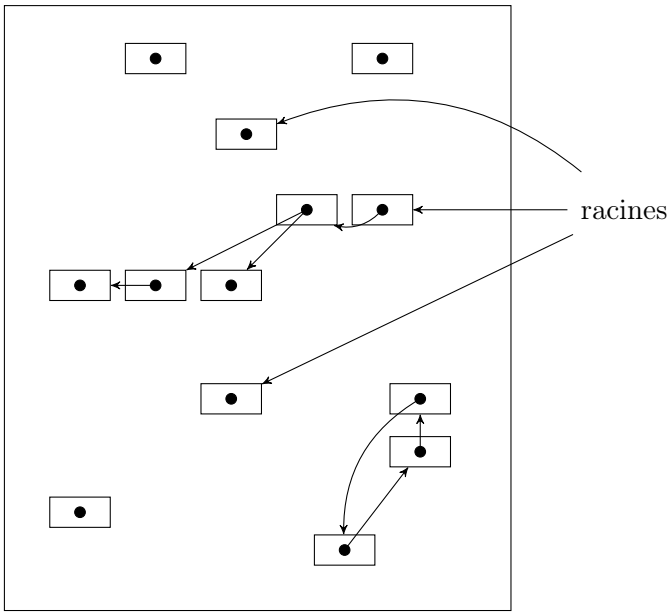


FIGURE 2.2 – Atteignabilité des objets dans un tas

chaque fois qu'une référence est modifiée, ce qui est extrêmement coûteux [Hart 88]. Ces deux problèmes limitent grandement l'utilisation de ce ramasse-miettes. Bien qu'il existe des solutions [Weiz 69, Knut 73, Deut 76, Wise 79] permettant de coupler un tel ramasse-miettes à un autre type capable de détecter les cycles, je n'ai pas considéré le comptage de références dans ces travaux.

Les autres types de ramasse-miettes réalisent une phase de *marquage* afin de détecter les objets survivants. Ils peuvent être classés en quatre familles :

Marque et nettoie Ce type de ramasse-miettes [McCa 60, Edel 92] maintient une liste de blocs libres dans lesquels sont alloués les objets. Les objets ne bougent donc jamais, ce qui a pour principal défaut de générer un partitionnement des données ;

Copiant Le principe [Feni 69, Chen 70] est de diviser la mémoire en deux espaces égaux et d'utiliser successivement l'un puis l'autre espace, une collection consiste à copier, pendant la phase de marquage, les objets survivants vers l'espace non utilisé ;

Compactant Dans cette famille [Cohe 83], les objets survivants sont compactés vers le bas du tas. Chaque nouvelle allocation se fait comme pour le copiant au dessus du dernier objet ;

Générationnel Ce type de ramasse-miettes [Lieb 83, Unga 84] gère plusieurs générations d'objets. Chaque nouvelle allocation se fait dans la génération la plus jeune, appelée la *nurserie*. Elle est donc collectée plus souvent que les autres.

Le nombre de variantes existant dans chaque famille est énorme ; il est bien sûr impossible de les détailler toutes ici. Lorsque cela s'avérait nécessaire, une ou deux variantes de chaque famille ont été considérées dans la suite du document. D'une manière générale, un grand nombre d'aspects ont trait à la collection automatique des données. Ils seront détaillés dans la suite dès que nécessaire.

S'il existe tant d'algorithmes de ramasse-miettes différents utilisés, c'est qu'ils ont tous des propriétés différentes, en termes de :

Consommation mémoire Chaque ramasse-miettes réserve un certain espace pour son fonctionnement propre. Cet espace peut aller jusqu'à 50% de la mémoire ;

Efficacité La complexité varie d'un algorithme à l'autre, en fonction de différents paramètres dépendant à la fois de l'application et du matériel ;

Consommation énergétique De même que les performances varient, la consommation énergétique varie également [Chen 02] ;

Accès mémoire Le nombre d'accès mémoire est un facteur important de la vitesse d'exécution et de la consommation énergétique des ramasse-miettes, mais importe aussi énormément lorsque l'on utilise des mémoires dont la durée de vie se compte en nombre d'accès ;

Exactitude Certains ramasse-miettes comme les compteurs de références ne sont pas exacts. Ils ne collectent pas tous les objets inutiles, ce qui peut induire un surplus de consommation mémoire.

Il n'existe a priori pas de « meilleur » algorithme de gestion de mémoire à objets, chacun ayant ses avantages et ses inconvénients. Ceci a déjà été montré au niveau des performances au-dessus d'une mémoire de type RAM [Atta 01, Fitz 00, Zorn 90, Smit 98]. Une autre illustration de cela est le fait que Sun donne les moyens de modifier la façon dont est gérée la mémoire dans sa machine virtuelle. De plus, en fonction de sa conception, un ramasse-miettes peut être plus ou moins portable, plus ou moins réutilisable, plus ou moins maintenable. Or, ces facteurs sont importants dans le monde de l'embarqué.

On remarque que plusieurs solutions de gestion de mémoire à objets font appel à un partitionnement logiciel. Le principe est d'allouer des objets aux propriétés semblables dans la même région afin de permettre leur collection très rapide. Ainsi, dans le modèle générationnel plusieurs générations sont utilisées afin de ne collecter qu'une petite partie de la mémoire à la fois. Plusieurs travaux s'appuient sur l'analyse d'échappement [Blan 98, Choi 99] pour allouer les objets n'échappant pas à leur contexte dans des espaces différents. Un grand nombre de travaux similaires existent [Gay 01, Gros 02, Chin 04, Sala 07].

Bien que le domaine de la gestion de mémoire à objets ait été très largement étudié, de telles solutions ne sont considérées pour les systèmes embarqués que depuis peu. En effet, ce contexte particulier pose un certain nombre de problèmes à la gestion de la mémoire orientée objet pour système embarqué.

2.3 Gestion de la mémoire orientée objet pour système embarqué

Il existe actuellement différents systèmes embarqués à objets.

2.3.1 Systèmes embarqués à objets

Les principaux systèmes embarqués gérant des mémoires à objets sont des machines virtuelles Java, dont voici les principales :

J2ME (*Java 2 Micro Edition* [Micr 00]) est un dérivé de Java qui cible les équipements dotés d'au moins 128 Ko de mémoire. À l'heure actuelle, deux configurations sont

mises à disposition par Sun sur le marché. Premièrement la ***Connected Device Configuration*** qui vise les équipements connectés et sédentaires. Elle se base généralement sur la machine virtuelle **CVM** (Compact Virtual Machine) de Sun. Deuxièmement, la ***Connected Limited Device Configuration*** qui est destinée à des équipements mobiles, plus restreints tels que les téléphones portables ; elle utilise plutôt la **KVM** (Kilobyte Virtual Machine).

Squawk est un projet [Shay 03] de recherche pour une machine virtuelle fonctionnant sur la nouvelle génération de cartes à puce.

JavaCard est une spécification dégradée de Java conçue pour profiter des avantages de Java sur les cartes à puce.

VM* est un canevas [Kosh 05] dédié à la construction de machines virtuelles Java pour réseaux de capteurs.

JEPES est une plateforme Java embarquée [Schu 03]) visant les équipements très contraints.

Des solutions équivalentes sont en train d'être développées par Microsoft, avec un framework appelé *.NET Micro*. Il est pour le moment difficile d'obtenir beaucoup d'informations sur des solutions basées sur cette architecture mais rien ne laisse supposer qu'elles soient très différentes, du moins du point de vue de la gestion mémoire, des machines virtuelles Java. De même, le langage ADA a été très utilisé dans l'embarqué, bien qu'il soit actuellement très peu considéré. Ces anciens systèmes avaient généralement des propriétés temps réel, ce qui ne laissait pas la place à une gestion automatique de la mémoire.

La conception de n'importe quel système embarqué demande une phase de déploiement du logiciel sur le matériel. Cette phase est d'ailleurs plus importante pour la conception de machine virtuelle embarquée car un certain nombre d'opérations supplémentaires sont nécessaires liées aux différentes parties du logiciel à assembler (notamment les classes à lier entre elles etc.). Quoiqu'il en soit, il n'existe pas véritablement de corps de métier responsable de ces opérations de déploiement. Elles peuvent être réalisées par l'électronicien qui conçoit son circuit, comme par le concepteur des applications destinées à tourner sur le système. Cela dépend vraiment des pratiques industrielles (industrie de la carte à puce, capteur de terrain...). La figure 2.3 illustre ce phénomène, en effectuant une analogie avec la production du café. En effet, on n'a plus, comme sur un ordinateur conventionnel, un producteur et un consommateur de code, mais une étape supplémentaire réalisée hors de la cible : le *torréfacteur de code*. Quelle que soit la personne responsable de cela, cette phase peut être compliquée et, de plus, différente pour chaque système embarqué. C'est pourquoi il est aussi important de simplifier ce processus. Cet aspect sera rappelé à plusieurs reprises dans la suite du document.

Quel que soit le type de couche logicielle utilisée pour gérer le matériel, l'allocation, le placement, la libération, ou encore le partage de données sont pris en charge par un gestionnaire mémoire. La mise au point d'un gestionnaire mémoire pour système embarqué est difficile car il lui faut gérer plusieurs mémoires aux propriétés différentes. Les problématiques que j'adresse dans ce document concernent la gestion de mémoire orientée objet. Ces problématiques sont donc d'autant plus ardues, puisqu'il s'agit de faire le pont entre différentes mémoires, des besoins applicatifs très différents, et les contraintes du modèle de programmation objet, avec notamment l'utilisation de ramasse-miettes aux propriétés très différentes. Tout d'abord, les particularités du matériel doivent être prises en compte dans l'évaluation de l'efficacité des algorithmes de ramasse-miettes.

(a) Schéma traditionnel de production et consommation de code

(b) Schéma de déploiement d'un système embarqué

FIGURE 2.3 – Le torréfacteur de code, responsable du déploiement du logiciel sur la cible

2.3.2 Efficacité des algorithmes de ramasse-miettes

Aujourd'hui, les améliorations des méthodes de programmation telles les langages orientés objets, les machines virtuelles et la collecte automatique des données sont surtout utilisées sur des systèmes informatiques non contraints, non embarqués. Ces systèmes fonctionnent selon le schéma suivant : les programmes s'exécutent dans mémoire de travail (généralement la RAM), et rendent leur données persistantes par l'usage d'une mémoire dont les données ne sont pas effacées lorsqu'elle n'est plus allimentée (par exemple un disque dur). Ce schéma n'est plus le même dans le cadre de systèmes embarqués. Notamment, les différences de temps de lecture et d'écriture doivent être prises en compte. Plus précisément, cela m'a amené à travailler sur deux points :

- La mise au point des complexités théoriques de ramasse-miettes prenant en compte les spécificités du matériel ;
- La validation expérimentale de ces complexités, afin d'observer s'il existe des ramasse-miettes meilleurs que d'autres sur le type de matériel visé, et de connaître les paramètres fins qui influent sur la gestion automatique de la mémoire ;

2.3.3 Problèmes architecturaux

Les spécificités du matériel embarqué, les spécificités des programmes à objets, et celles des différents ramasse-miettes posent un certain nombre de problèmes relatifs à la mise au point d'une architecture de gestionnaire mémoire.

Tout d'abord, le partitionnement à la fois matériel et logiciel doit être supporté. Une des motivations principales de mes travaux étant de supporter l'allocation et la libération automatique de la mémoire, il s'agit de fournir une vue unifiée des divers espaces mémoire

résultant de ce partitionnement. Cette difficulté est renforcée par le fait qu'un grand nombre de systèmes embarqués ne disposent pas d'un adressage virtuel.

Ensuite, la gestion manuelle de la mémoire est — en principe en tous cas — assez simple du point de vue du programmeur. Un segment de mémoire (virtuelle ou non) est alloué pour l'application. L'allocation et la libération dynamique de données sont demandées par l'application, au travers des instructions données explicitement par le programmeur de l'application. Sur un ordinateur conventionnel, un programme à objets va s'exécuter en demandant une certaine quantité de mémoire qui servira pour le tas. Dans ce cas, il existe deux couches de gestion de la mémoire, ce que l'on veut bien sûr éviter afin de favoriser les performances et la petite taille de code. On souhaite véritablement avoir un gestionnaire de mémoire à objets au-dessus même du matériel.

Enfin, si le placement des données est important sur tout système informatique, il l'est encore davantage sur un système gérant plusieurs espaces mémoire aux propriétés différentes. Il s'agit donc de gérer efficacement le placement des données, en prenant en compte le fait que les données sont dans notre cas des objets.

Mes contributions répondant à ces problématiques sont, en résumé, les suivantes :
- La mise au point d'une nouvelle architecture adaptée à la gestion du partitionnement. Cette architecture permet la gestion de chaque espace mémoire par un gestionnaire adapté aux différentes propriétés, soit logicielles, soit matérielles, que peut avoir cet espace ;
- La mise au point d'une solution de placement des données dans les différents espaces basée sur un langage dédié. L'utilisation de ce langage permet une flexibilité de la mémoire par rapport au matériel et aux applications, conformément aux besoins des systèmes embarqués décrits précédemment.

La présence au sein de l'appareil considéré de support de mémoire virtuelle peut éventuellement apporter une aide à la mise au point d'un modèle de mémoire unifié. On ne considère pas dans les travaux présentés ici la présence d'un tel support. Même si les propositions faites ne sont pas incompatibles avec la présence d'un tel support, aucune optimisation ou évaluation n'a été faite, ceci pour trois raisons. La première est simple : le support de l'adressage virtuel est coûteux en surface de silicium et est donc généralement absent des petits systèmes embarqués. Deuxièmement, la gestion du partitionnement matériel et logiciel nécessite de disposer d'informations précises que ce support n'a pas vocation à utiliser ou rechercher. Par exemple, l'utilisation de critères purement logiciels tels que le type d'objets. Enfin, l'utilisation de mémoire virtuel a un coût en terme de performances, qu'il reste à évaluer dans le contexte des systèmes embarqués à objets.

2.3.4 Gestion du déploiement du système

Afin d'embarquer un système sur un appareil donné, une image initiale de ce système est produite grâce notamment à un compilateur croisé. Cette image du système est ensuite embarquée sur la cible au travers d'une mémoire persistante telle que la ROM. C'est une véritable difficulté que de réaliser cette phase de déploiement efficacement, notamment pour la gestion de la mémoire. Au démarrage, une partie des données est copiée de la mémoire persistante dans d'autres mémoires réinscriptibles, le reste est utilisé *en place* afin d'économiser de la mémoire. La présence de données dans une mémoire non réinscriptible peut poser problème puisque ces données ne peuvent être modifiées

et que les données référencées par celles-ci ne peuvent être déplacées. De plus, l'aspect placement de données est encore une fois bien présent puisqu'il est nécessaire de définir la destination de chaque donnée.

En conséquence, mes recherches dans la mise au point d'un déploiement efficace se sont orientées selon les trois axes suivants :

- L'amélioration du placement des données initiales du système ;
- L'optimisation des ramasse-miettes vis-à-vis de la présence de mémoire de type ROM ;
- L'intégration de toutes les solutions proposées au sein d'un véritable système.

2.4 Conclusion

En résumé, le problème général qu'on se propose d'adresser dans ce document est celui de la gestion automatique de la mémoire dans un contexte embarqué. Dans ce but, il est nécessaire de répondre à un certain nombre de sous-problèmes.

Premièrement, les propriétés matérielles très différentes des mémoires remettent en cause un certain nombre de travaux existants. Cela m'a amené à proposer plusieurs études des mécanismes de gestion automatique de la mémoire.

Deuxièmement, les systèmes embarqués incluent plusieurs types de mémoire, ce qui nécessite de savoir gérer plusieurs espaces mémoire. Ce partitionnement matériel associé au besoin de partitionnement logiciel du modèle de mémoire à objets augmente les contraintes en terme de portabilité et de réutilisation des systèmes. Le partitionnement en espaces hétérogènes requiert également la mise au point de solutions intelligentes de placement.

Enfin, le déploiement particulier des systèmes embarqués associé à la présence éventuelle de mémoire non réinscriptible est contraignant mais permet également quelques optimisations.

Troisième Chapitre

Ramasse-miettes pour l'embarqué

« La volonté aboutit à un ajournement, l'utopie ; la science aboutit à un doute, l'hypothèse. »

Victor Hugo,
Post-scriptum de ma vie.

Ce chapitre explore le comportement des algorithmes principaux de collecte automatique de données en prenant en compte certaines spécificités des systèmes embarqués. En particulier, l'étude présentée ici prend en compte des mémoires de différents types sur les architectures embarquées. Notamment, l'existence de temps d'accès différents en lecture et en écriture est adressée.

Ce chapitre débute ainsi par une étude analytique de la complexité des algorithmes. Cette étude se veut en premier lieu théorique. En deuxième lieu, cette étude est validée expérimentalement. L'impact d'autres considérations telles que la présence de caches de données est prise en compte.

3.1 Limitations des études existantes

Les algorithmes de collecte automatique de données ont été largement étudiés au cours des dernières années. Ces travaux considèrent implicitement que l'architecture matérielle sous-jacente n'est composée que d'une seule mémoire de travail (RAM) puisque, jusqu'à récemment, de tels algorithmes n'étaient utilisés que sur des classes de systèmes de l'ordre de l'ordinateur personnel ou supérieur. Cependant, avec l'évolution du matériel informatique, l'utilisation de tels mécanismes est de plus en plus mise en avant pour une utilisation en système embarqué. Or, la plupart des cibles matérielles embarquent plusieurs types de mémoires aux diverses propriétés. L'objectif de ce chapitre est d'étudier certains algorithmes de ramasse-miettes en prenant en compte certaines spécificités propres à l'embarqué.

Premièrement, un certain nombre d'études ont été publiées, concernant l'efficacité des ramasse-miettes, définie comme la quantité de données collectée par unité de temps. Ainsi, [Jone 96] et [Sris 00] regroupent les efficacités en temps de différents algorithmes classiques. Celles-ci sont généralement données en fonction de la survivance (la quantité de données survivant à une collecte), et de constantes non spécifiées, dépendantes de divers paramètres. Ces études sont incomplètes dans notre contexte puisqu'elles ne tiennent pas compte, par exemple, de la différence de coût entre une lecture et une écriture. En premier lieu, une étude analytique prenant en compte ce type de comportement a donc été réalisée.

Ensuite, un certain nombre de travaux ont déjà montré qu'il n'existe pas un algorithme de ramasse-miettes qui soit le meilleur, en terme de performances, pour toutes les applications. Un des objectifs de ce chapitre est d'étudier le comportement des ramasse-miettes au-dessus de divers profils de mémoires afin d'observer l'impact de ces profils sur la collecte. De plus, la granularité des résultats donnés jusqu'à présent est généralement à très gros grains, et ne met pas en évidence les paramètres influant directement sur les coûts des accès mémoire. Les résultats présentés ici ont permis d'identifier ces paramètres.

Dans tout le chapitre, on ne considèrera que quelques algorithmes traditionnels, choisis pour leurs variétés et leurs qualités reconnues. Les ramasse-miettes considérés dans ce chapitre sont ainsi au nombre de quatre : deux ramasse-miettes compactants, un copiant, et un de type « Marque et nettoie ». Comme il était impossible d'étudier tous les types de ramasse-miettes existants, les algorithmes de type générationnel ont été volontairement

N_t : Nombre total d'objets avant collecte
N_l : Nombre d'objets survivants
S_t : Taille moyenne des objets avant collecte
S_l : Taille moyenne des objets survivants
N_r : Nombre total de références dans les objets survivants
N_{wr} : Nombre de références valides

TABLE 3.1 – Notations pour les paramètres communs à plusieurs ramasse-miettes

écartés, pour plusieurs raisons. La première est qu'il n'existe pas aujourd'hui d'évidence (à ma connaissance) dans la littérature, de leurs qualités concernant l'embarqué. La seconde est qu'ils nécessitent, pour être efficaces, un support de la plateforme d'exécution. En particulier, ils nécessitent l'usage de barrières d'accès qui sont très coûteuses.

En premier lieu, nous allons nous intéresser au comportement des ramasse-miettes du point de vue de l'étude analytique.

3.2 Étude analytique

Cette section est organisée comme suit. Premièrement, les complexités des différents ramasse-miettes étudiés sont exposées, en fonction de divers critères facilitant la compréhension. Ensuite, ces complexités sont validées expérimentalement. Enfin, elles sont modifiées de façon à faire apparaître des paramètres significatifs. Les algorithmes de collecte automatique de données, et en particulier ceux considérés dans ce chapitre, possèdent certaines similitudes de fonctionnement qui nous amènent à présenter en premier lieu les complexités de ces opérations communes.

3.2.1 Opérations communes

Tout d'abord, on considère dans l'étude analytique que les objets contiennent un en-tête standard codant, entre autres, la taille de l'objet et l'état de l'objet par rapport au gestionnaire mémoire. Cette implémentation est très courante, et tient la plupart du temps sur 2 mots [Micr 01, Micr 00].

Les paramètres dont dépendent les opérations détaillées ci-après sont regroupés dans la table 3.1.

Parcours de la mémoire Dans le cas où le ramasse-miettes ne laisse aucun bloc libre entre les blocs de données, parcourir tous les objets du tas est relativement simple puisqu'il suffit pour cela de savoir calculer la taille de chaque objet. Bien que cela dépende de l'implémentation de la machine virtuelle considérée, on considère ici que la taille de l'objet est donnée dans l'en-tête. Cette opération est répétée pour les n objets consécutifs. Ce paramètre n peut en pratique être égal à N_t (le nombre d'objets dans le tas) ou à N_l (nombre d'objets survivants). Dans la suite on notera $C_o^{N_t}$ (respectivement $C_o^{N_l}$) le coût en accès du parcours de N_t (respectivement N_l) objets contigus.

Parcours des références d'un objet Tous les ramasse-miettes effectuant des déplacements d'objets doivent à un moment donné parcourir toutes les références afin de les mettre à jour si les objets pointés ont bougé. L'implémentation considérée du parcours des références d'un objet donne la complexité suivante. Premièrement, le pointeur de classe de l'objet est lu. Un entier dans cette classe sert de champ de bits, chaque i-ème bit indiquant si le i-ème mot de l'objet est une référence ou une valeur. Il suffit donc d'une lecture par référence. Cela donne donc un total de $3 + N_r$ lectures objet, où N_r est le nombre de références contenues dans l'objet. On note $C_r^{N_l}$ le coût en accès du parcours de toutes les références des objets survivant à une collecte.

Marquage De façon à atteindre et marquer tous les objets survivants, on considère ici un mécanisme de marquage utilisant une pile , décrit dans [Jone 96]. Cet algorithme est standard et efficace ; il a l'avantage de ne pas être récursif, ce qui est extrêmement important pour les systèmes contraints en mémoire. La figure 3.1 présente cet algorithme. Marquer un objet nécessite en premier lieu uniquement de positionner un bit dans l'entête des objets. Cette hypothèse est reconsidérée ultérieurement en observant l'impact du marquage en champs de bits.

```
while mark_stack ≠ empty
    N = pop(mark_stack)
    for M in Children(N)
        if mark_bit(*M) == unmarked
            mark_bit(*M) = marked
            push(*M, mark_stack)
```

FIGURE 3.1 – L'algorithme de marquage en pile

Étant donné l'algorithme de marquage, voici le coût en accès d'une phase de marquage :

Sous-tâche	Coût	Explication (éventuelle)
Gestion de la pile	$N_l \times R + N_l \times W$	1 empilage et 1 dépilage par objet survivant
Parcours des références	$C_r^{N_l}$	
Accès aux marques	$N_{wr} \times R + N_l \times W$	1 lecture par référence pour tester, mais un positionnement par objet survivant

3.2.2 Ramasse-miettes compactant

Typiquement, les ramasse-miettes compactants ont le fonctionnement suivant. Tout d'abord, les objets survivants sont marqués. Ensuite, ces derniers sont parcourus l'un après l'autre, l'adresse à laquelle ils seront déplacés est calculée et stockée dans un mot supplémentaire de l'en-tête d'objet. Puis, toutes les références sont mises à jour avec l'adresse stockée dans un mot supplémentaire de l'objet initialement pointé. Enfin, les objets peuvent être compactés sans risque.

Marquage :C_m

N_{ur} : Nombre de références devant être mises à jour (car l'objet qu'elles pointent a bougé) ;
S_b : Quantité (en octets) d'objets survivants ne devant pas être compactés ;
N_c : Nombre d'objets survivants ne devant pas être compactés.

Calcul des nouvelles adresses

Sous-tâche	Coût	Explication (éventuelle)
Traversée de la mémoire	$C_o^{N_t}$	
Tests du bit de marquage	$N_l \times R$	
Écriture des nouvelles adresses	$N_l \times W$	Une nouvelle adresse par objet survivant

Mise à jour

Sous-tâche	Coût	Explication (éventuelle)
Parcours de la mémoire	$C_o^{N_t}$	
Tests du bit de marquage	$N_t \times R$	Lecture d'une marque par objet
Parcours des références	$C_r^{N_l}$	
Mise à jour des références	$N_{wr} \times R + N_{ur} \times W$	Une lecture et une écriture par mise à jour

Compactage

Sous-tâche	Coût	Explication (éventuelle)
Parcours de la mémoire	$C_o^{N_t - N_c}$	Parcours de tous les objets excepté la base du tas
Compactage	$(S_l - S_b) \times (R + W)/S_w$	Compactage du haut du tas

3.2.3 Ramasse-miettes compactant (variante)

Plusieurs variantes du ramasse-miettes compactant existent. On va considérer ici un algorithme utilisant une table pour la gestion des références : une fois les objets survivants marqués, l'adresse de chaque objet survivant est insérée dans une table. Puis, toutes les références sont modifiées de façon à pointer sur la case de la table qui la contient. Ensuite, un compactage est effectué, durant lequel toutes les adresses de la table sont mises à jour avec les nouvelles adresses des objets compactés. Enfin, les références du tas sont mises à jour.

Marquage C_m

Initialisation de la table de références

Sous-tâche	Coût	Explication (éventuelle)
Parcours de la mémoire	$C_o^{N_t}$	
Test marquage	$N_t \times R$	Lecture de la marque de chaque objet pour test
Insertion	$N_l \times W$	L'adresse de chaque objet est insérée dans la table

Modification

Sous-tâche	Coût	Explication (éventuelle)
Parcours de la table	$N_l \times R$	
Parcours des références	$C_r^{N_l}$	
Recherche dans la table	$N_{wr} \times log_2(N_l) \times R$	Recherche dichotomique de l'emplacement dans la table de chaque référence
Modification	$N_{wr} \times R + N_{ur} \times W$	Lecture de chaque référence et modification de celles à mettre à jour

Compactage

Sous-tâche	Coût	Explication (éventuelle)
Memory traversal	$C_o^{N_l - N_c}$	
Compactage	$(S_l - S_c) \times (R + W)/S_w$	Les objets survivants sont copiés
Changement dans la table	$(N_l - N_c) \times W$	L'adresse de chaque objet déplacé est modifiée dans la table

Mise à jour

Sous-tâche	Coût	Explication (éventuelle)
Parcours de la table	$N_l \times R$	
Parcours des références	$C_r^{N_l}$	
Mise à jour	$N_{wr} \times R + N_{ur} \times W$	

3.2.4 Ramasse-miettes copiant

L'algorithme copiant divise le tas en deux zones de tailles égales. Il alloue de façon contiguë dans une de ces zones. Quand celle-ci est pleine, l'autre est utilisée pour y copier les objets pendant la phase de marquage. La nouvelle adresse de l'objet est écrite sur le premier mot de l'ancien emplacement. De cette façon, une deuxième phase est ensuite déclenchée afin de mettre à jour les références.

Marquage

Sous-tâche	Coût	Explication (éventuelle)
Stack	$N_l \times R + N_l \times W$	Un empilement et un dépilement par objet survivant
Copy	$S_l/S_w \times (R + W)$	Une lecture et une écriture par mot à copier
Pointer	$N_l \times W$	Nouvelle adresse de chaque objet survivant
Références	$C_r^{N_l}$	
Test des marques	$N_{wr} \times R + N_l \times W$	

Mise à jour

Sous-tâche	Coût	Explication (éventuelle)
Parcours des objets	$C_o^{N_l}$	
Parcours des références	$C_r^{N_l}$	
Mise à jour des références	$N_{wr} \times (R + W)$	Une lecture pour lire la référence, une écriture pour la mise à jour

3.2.5 Marque et nettoie

Ce ramasse-miettes gère des blocs libres dans lesquels les nouvelles instances vont être allouées. Si aucun bloc libre de taille suffisante n'est trouvé, une collecte de données est

déclenchée. L'algorithme marque tous les objets survivants, puis collecte effectivement les données en les groupant en blocs libres. Bien que le fonctionnement général soit très simple à décrire, la complexité de cet algorithme dépend de divers paramètres. L'implémentation considérée est une implémentation standard dans laquelle une liste de blocs libres est utilisée. Un bit est utilisé dans l'en-tête des objets pour indiquer si le bloc est un objet survivant ou un bloc libre (le même que celui utilisé pour marquer un objet comme survivant).

Cet algorithme effectue un certain nombre d'opérations pendant l'allocation d'un objet, afin de trouver un bloc libre d'une taille suffisante. On tient compte de ces opérations dans les coûts engendrés par l'utilisation de ce ramasse-miettes. De plus, à cause de l'utilisation de blocs libres, le tas est partitionné, cela peut mener à déclencher une collecte de données alors qu'il reste de l'espace libre mais partitionné dans différents blocs libres. Toutes ces particularités amènent à introduire des paramètres spécifiques, listés dans la table 3.2.5.

N_{fbs} : Nombre de blocs libres parcourus pendant la phase de constitution de nouveaux blocs libres ;
N_{fbe} : le nombre de blocs libres agrandis pendant cette même phase ;
N_{al} : Nombre d'allocations avant d'effectuer une nouvelle collecte ;
N_{fba} : Nombre de blocs libres parcourus pendant les allocations ;
S_{lm} : La taille de l'espace perdu à cause du partitionnement du tas.

TABLE 3.2 – Paramètres spécifiques au Mark and Sweep

Marquage C_m

Allocations

Sous-tâche	Coût	Explication (éventuelle)
Parcours des blocs libres	$N_{fba} \times R$	Lecture de l'en-tête de chaque bloc libre
Insertion dans bloc libre	$N_{al} \times R + N_{al} \times W$	Mise à jour de la taille du bloc libre trouvé

Nettoyage

Sous-tâche	Coût	Explication (éventuelle)
Parcours de la mémoire	$C_o^{N_t}$	
Parcours des blocs libres	$N_{fbs} \times R$	Lecture de l'en-tête de chaque bloc libre
Tests du bit de marquage	$N_t \times R$	
Union en blocs libres	$N_{fbe} \times W$	Écriture de l'en-tête du bloc constitué

3.3 Validation expérimentale

De façon à valider les complexités présentées précédemment, je vais maintenant comparer ces dernières à des mesures faites sur une véritable machine virtuelle Java [JITS].

3.3.1 Conditions expérimentales

La machine virtuelle en question est écrite en Java. Cela n'a aucun impact sur les mesures puisque ce ne sont pas des temps d'exécution qui sont mesurés. En revanche, cela facilite grandement son instrumentalisation, en vue d'obtenir toutes les informations relatives aux accès mémoire. Plus en détail :

- les allocateurs d'objets ont été surchargés pour fonctionner de manière cohérente avec un ramasse-miettes donné et notifier des accès effectués. L'adresse des objets est stockée ;
- quand l'allocation d'un objet ne peut se faire sous peine de dépasser une limite donnée, une collecte est déclenchée. Tous les types de ramasse-miettes présentés dans la section précédente ont été implémentés de telle sorte que tout accès aux données soit pris en compte ;
- enfin, les fonctions de l'interpréteur de bytecodes effectuant des accès à la mémoire ont été surchargées très simplement de façon à compter ces accès. Par exemple, le bytecode `putfield` (qui change la référence d'un champ de classe) notifie d'une écriture. Cette partie du travail a été assez simple, car l'interpréteur utilisait déjà des accesseurs particuliers.

Les mesures ont été effectuées sur des applications de tests typiques ayant différents types de comportement quant à leur utilisation de la mémoire :

dhrystone est un programme ayant pour but d'évaluer les performances générales du processeur ;

check teste différentes propriétés de machines virtuelles ;

raytracer effectue la partie calculatoire du rendu d'une scène 3D ; il provoque une utilisation intensive de la mémoire et du processeur ;

crypt effectue le cryptage et le décryptage d'un fichier de données ;

moldyn simule les interactions existant entre différentes molécules organiques, effectuant ainsi un grand nombre de calculs ;

peptest teste différentes propriétés de la machine virtuelle.

3.3.2 Résultats

J'ai mesuré les accès effectués par le gestionnaire mémoire pendant l'exécution des applications précédemment citées. Les résultats sont comparés sur la figure 3.2 à ceux donnés par les complexités théoriques. Pour chaque type de ramasse-miettes et chaque application, la figure montre :

- le nombre de lectures donné par les complexités, et l'erreur moyenne constatée par rapport à la véritable exécution de l'application (en pourcentage par rapport à la vraie exécution) ;
- le nombre d'écritures donné par les complexités, et l'erreur moyenne constatée par rapport à la véritable exécution de l'application ;
- en dessous des barres, on trouve les erreurs maximales observées à la fois pour les écritures et pour les lectures.

Ces résultats montrent que l'erreur observée sur les complexités théoriques est faible. Elle est de moins de 3% en moyenne, et atteint un maximum de 6% sur une collecte particulière d'une application particulière, pour un type spécifique de ramasse-miettes.

On en déduit que les complexités données sont représentatives et assez précises pour être interprétées : par exemple pour savoir comment un type de ramasse-miettes va s'en sortir sur un type de mémoire donné.

3.3.3 Simplification des complexités théoriques

L'efficacité d'un ramasse-miettes est définie comme le nombre d'octets collectés par unité de temps [Jone 96]. On définit l'efficacité relative aux accès mémoire $e_m = \frac{Nb\ \ octets\ \ collectés}{coût\ \ des\ \ accès}$.

Le nombre d'octets collectés par chaque type de ramasse-miettes pendant une collecte est donnée par la quantité de mémoire qui survit à la collecte soustraite à la quantité de mémoire présente avant collecte : $S_t - S_l$. Cependant la taille totale des données avant collecte n'est pas la même pour tous les ramasse-miettes, étant donné que chacun réserve un certain espace pour son fonctionnement propre (la moitié du tas pour le copiant par exemple).

La figure 3.3.3 donne les expressions de S_t, taille totale des données avant collecte, pour les différents ramasse-miettes, ainsi que le nombre d'accès que nécessite une collecte. À noter que S_h représente la taille du tas. Dans les formules données, certains paramètres basiques utilisés précédemment par souci de clarté cèdent la place à d'autres paramètres plus pertinents. Par exemple, la *survivance*[1] apparaît. Elle est définie comme la fraction de la mémoire survivant à une collecte, vaut $r = \frac{S_l}{S_t}$, et a été identifiée dans un certain nombre de travaux [Sans 92, Spoo 05] comme un paramètre clef de l'efficacité des ramasse-miettes. La table 3.3.3 recense ces paramètres, auxquels il faut ajouter les paramètres spécifiques au ramasse-miettes *Marque et Nettoie*. Les conversions de paramètres effectués sont listées figure 3.3.3.

r : Survivance (fraction des données qui survit)
S_o : Taille moyenne d'un objet avant collecte
S_l : Taille moyenne d'un objet survivant
N_l : Nombre d'objets survivants
N_t : Nombre d'objets avant collecte
N_r : Nombre moyen de références par objet avant collecte
$N_{wr,o}$: Nombre moyen de références valides par objet survivant
c_b : Pour un ramasse-miettes compactant, proportion de données déjà compactées
S_b : Pour un ramasse-miettes compactant, taille moyenne des objets déjà compactés

TABLE 3.3 – Paramètres significatifs

3.4 Efficacité

On s'intéresse dans cette section aux différents algorithmes du point de vue de l'efficacité en accès mémoire.

1. *residency* en anglais

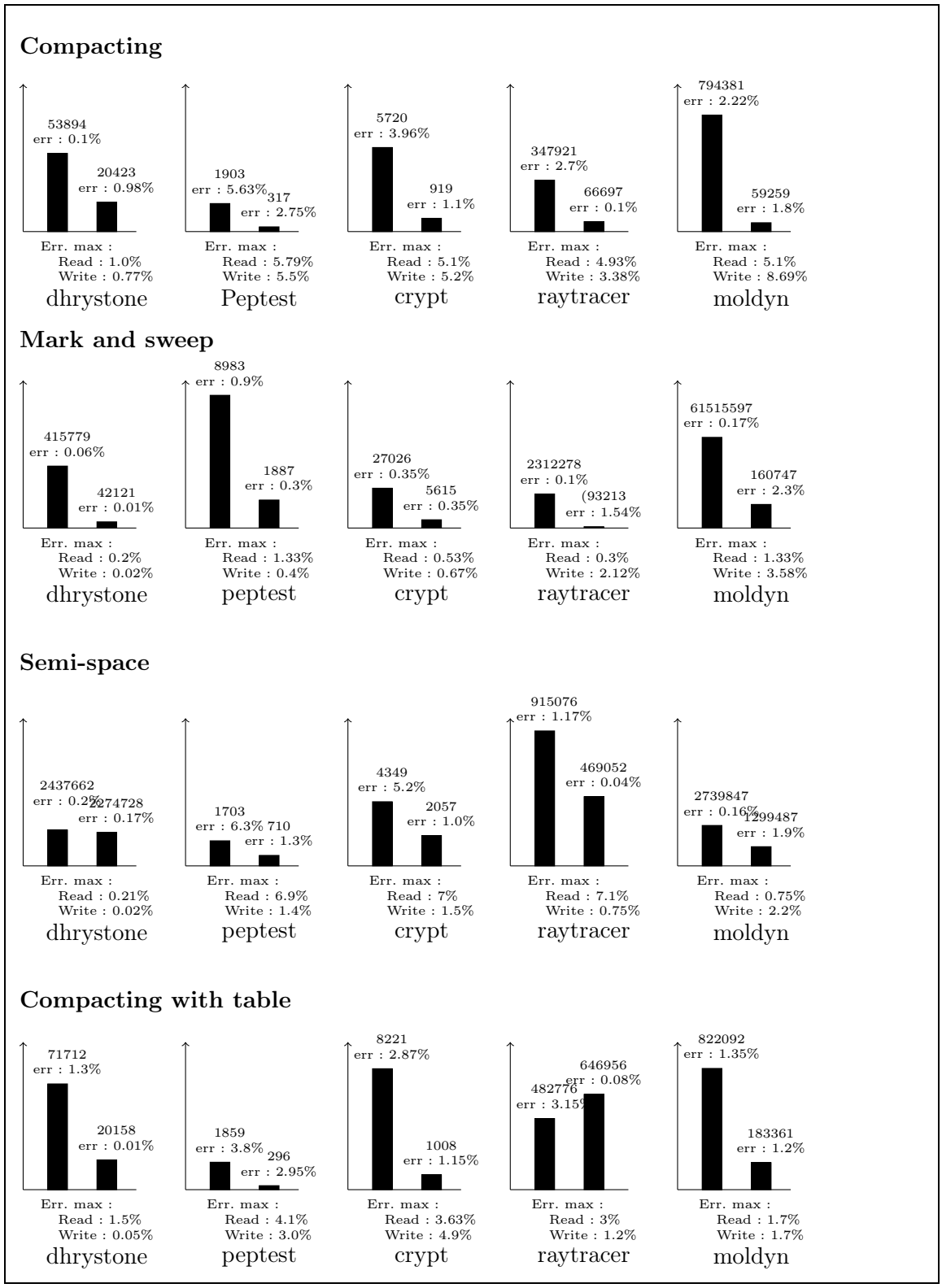


FIGURE 3.2 – Validation expérimentale des complexités

Conversion

$$S_l := rS_t \qquad N_l := \frac{S_l}{S_{o,l}} \qquad N_t := \frac{S_t}{S_o}$$
$$N_{wr} := N_{wr,o}N_l \quad N_r := N_{r,o}N_t \quad S_b := c_bS_l$$

Marquage

$$\begin{aligned} C_m &:= (R+W)N_l + C_r + N_{wr}R + WN_l \\ &:= \frac{(R+W)rS_t}{S_{o,l}} + \left(\frac{N_{r,o}S_t}{S_o} + 3\,\frac{rS_t}{S_{o,l}}\right)R + \frac{N_{wr,o}rS_tR}{S_{o,l}} + \frac{WrS_t}{S_{o,l}} \end{aligned}$$

Parcours des références

$$C_r := (N_r + 3\,N_l)\,R := \left(\frac{N_{r,o}S_t}{S_o} + 3\,\frac{rS_t}{S_{o,l}}\right)R$$

Parcours des objets

$$C_{o,N_t} := N_tR := \frac{S_tR}{S_o}$$

$$C_{o,N_l} := N_lR := \frac{rS_tR}{S_{o,l}}$$
$$C_{o,N_l-N_c} := 2\,(N_l - N_c)\,R := 2\,\left(\frac{rS_t}{S_{o,l}} - N_c\right)R$$

Compactant

$$S_t + \frac{4\times S_t}{S_o} = S_h => S_t \times (1 + \frac{4}{S_o}) = S_h => S_t = \frac{S_h}{(1+\frac{4}{S_o})}$$
$$\text{Readings} : 8\,\frac{rS_t}{S_{o,l}} + 2\,\frac{N_{r,o}S_t}{S_o} + 2\,\frac{N_{wr,o}rS_t}{S_{o,l}} + 5\,\frac{S_t}{S_o} - 2\,N_c + \frac{S_t}{S_w} - \frac{c_brS_t}{S_w}$$
$$\text{Writings} : 3\,\frac{rS_t}{S_{o,l}} + \frac{N_{wr,o}rS_t}{S_{o,l}} + \frac{S_t}{S_w} - \frac{c_brS_t}{S_w}$$

Marque et Nettoie

$$S_t = S_h - S_{lm}$$
$$\text{Readings} : 4\,\frac{rS_t}{S_{o,l}} + \frac{N_{r,o}S_t}{S_o} + \frac{N_{wr,o}rS_t}{S_{o,l}} + N_{fba} + N_{al} + 2\,\frac{S_t}{S_o} + N_{fbs}$$
$$\text{Writings} : 2\,\frac{rS_t}{S_{o,l}} + N_{al} + N_{fbe}$$

Copiant

$$S_t = \frac{S_h}{2}$$
$$\text{Readings} : 8\,\frac{rS_t}{S_{o,l}} + \frac{rS_t}{S_w} + 2\,\frac{N_{r,o}S_t}{S_o} + 2\,\frac{N_{wr,o}rS_t}{S_{o,l}}$$
$$\text{Writings} : 3\,\frac{rS_t}{S_{o,l}} + \frac{rS_t}{S_w} + \frac{N_{wr,o}rS_t}{S_{o,l}}$$

Compactant (version avec table)

$$S_t = \frac{S_h}{(1+\frac{4}{S_o})}$$
$$\text{Readings} : 14\,\frac{rS_t}{S_{o,l}} + 3\,\frac{N_{r,o}S_t}{S_o} + 3\,\frac{N_{wr,o}rS_t}{S_{o,l}} + 2\,\frac{S_t}{S_o} + \frac{N_{wr,o}rS_tlog_2(\frac{rS_t}{S_{o,l}})}{S_{o,l}} - 2\,N_c + \frac{rS_t}{S_w} - \frac{S_c}{S_w}$$
$$\text{Writings} : 4\,\frac{rS_t}{S_{o,l}} + 2\,N_{ur} + \frac{rS_t}{S_w} - \frac{S_c}{S_w} - N_c$$

FIGURE 3.3 – Complexités des différents ramasse-miettes

3.4.1 Efficacité en accès mémoire

A partir de l'étude analytique vérifiée expérimentalement, il est possible de calculer l'efficacité relative aux accès des ramasse-miettes considérés. Plus précisément, à partir des définitions analytiques de l'efficacité des ramasse-miettes, il est possible de tirer des résultats utiles. La figure 3.4 expose un de ces résultats. Elle montre en effet, pour différentes applications, quel ramasse-miettes a la meilleure efficacité, en fonction du coût de lecture et d'écriture de la mémoire sous-jacente. Pour obtenir ces résultats, l'efficacité de chaque ramasse-miettes a été calculée en prenant en entrée les paramètres expérimentaux de chaque application considérée, et ce pour chaque coût d'accès. Ces coûts d'accès sont, comme le montre la figure, différents pour la lecture et l'écriture, et se présentent comme un facteur par rapport à une valeur unitaire de 1. En effet on calcule bien des efficacités relatives les unes aux autres et non une performance exacte.

Ces résultats montrent plusieurs choses :

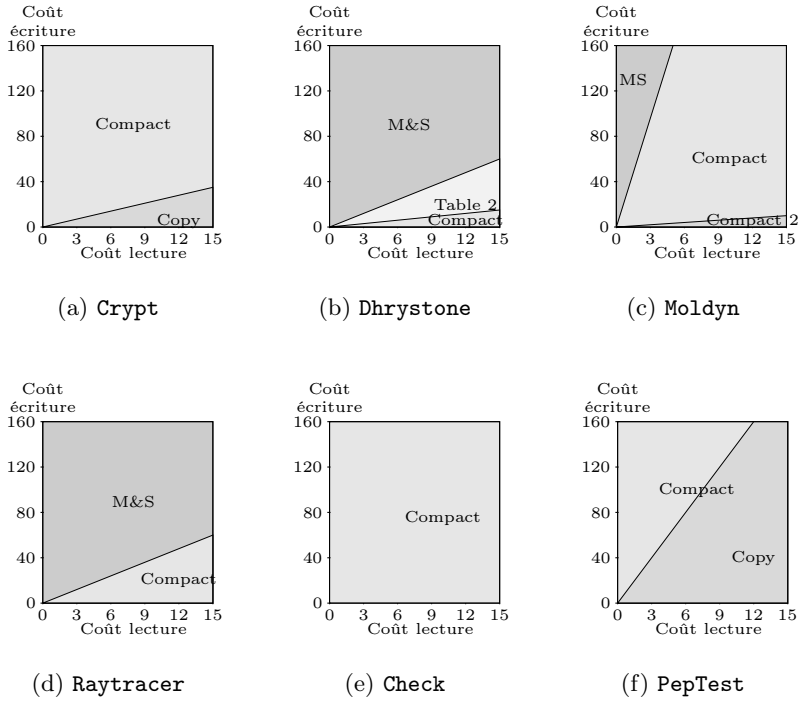
1. Ils sont pleinement cohérents avec les travaux précédents qui ont indiqué que nul algorithme n'est le meilleur pour toutes les applications [Atta 01, Fitz 00, Zorn 90, Smit 98].
2. Ils indiquent clairement que même pour une application donnée, il n'existe pas de ramasse-miettes qui soit le meilleur pour tous les profils de mémoire sous-jacents. En effet, on peut voir que le meilleur ramasse-miettes pour l'application *Moldyn* peut être le *Marque et nettoie*, le compactant simple ou le compactant à base de table de déréférencement selon les coûts de lecture et d'écriture.
3. Même s'il n'existe pas de règle, on peut trouver des tendances :
   - quand le *marque et nettoie* est le plus efficace, c'est pour les petits coûts de lecture et les hauts coûts d'écriture, ce qui est logique étant donné qu'il ne déplace pas les objets ;
   - quand le *copiant* est le meilleur, c'est au contraire pour les petits coûts d'écriture, ce qui est logique puisqu'il déplace *tous* les objets survivants.

Les complexités exposées précédemment sont assez simples en ceci qu'elles ne prennent pas en compte certains phénomènes dépendants du reste du système. En premier lieu, il convient d'évaluer l'impact du cache.

3.4.2 Impact du cache

Les études analytiques sont intéressantes mais ne prennent pas en compte les effets de mécanisme purement liés au support d'exécution, tels que l'impact de la présence de cache de données. Par conséquent, afin de compléter l'étude précédente, l'impact de ce mécanisme est maintenant mesuré.

Exposer les mesures de différents effets de cache pourrait faire l'objet de dizaines de pages de résultats sans pour autant apporter une bien grande contribution. Ici, le but est simplement d'évaluer si la présence de cache remet en cause les observations précédentes, ou si on peut arriver à d'autres conclusions ayant rapport à l'efficacité relative aux accès. Par conséquent, les systèmes de cache présentés ici sont relativement simples. Les taux de succès du cache ont été calculés expérimentalement pour chaque phase de chaque ramasse-miettes, et les efficacités modifiées en fonction, avec le coût suivant : un succès du cache a un coût nul. De plus, on ne considère qu'un seul niveau de cache, ce qui



(a) `Crypt` (b) `Dhrystone` (c) `Moldyn`

(d) `Raytracer` (e) `Check` (f) `PepTest`

FIGURE 3.4 – Meilleur algorithme de ramasse-miettes en fonction du profil de mémoire

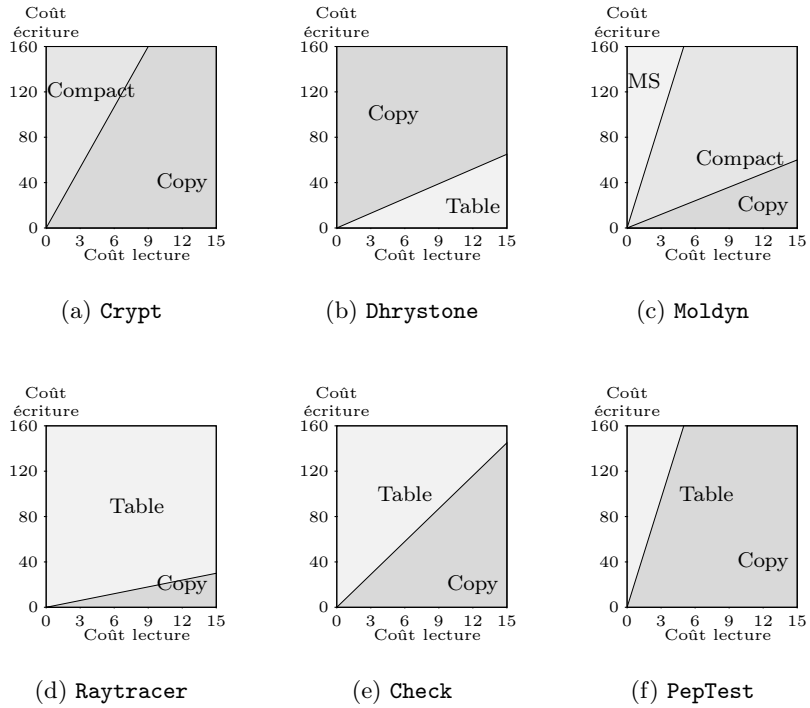
correspond au matériel que l'on trouve dans les petits systèmes embarqués. Enfin, la politique de cache utilisée est *LRU (Least-Recently Used)*, c'est-à-dire que la page de cache retirée en cas d'échec est la moins récemment utilisée [Smit 91].

Les résultats obtenus ne remettent clairement pas en cause les observations précédentes. Ils viennent même renforcer l'idée qu'il n'existe pas de meilleur algorithme pour tous les profils de mémoire puisque les résultats diffèrent selon le système de cache utilisé. On peut observer également que la variante du ramasse-miettes compactant utilisant une table pour la gestion des références présente un intérêt en présence de cache, confirmant ainsi que la mise en cache des références a un impact certain.

Une hypothèse de l'étude précédente est l'utilisation d'un système de marquage dans l'en-tête des objets qui a l'avantage de ne pas nécessiter d'espace supplémentaire pour marquer. Il peut cependant exister un certain impact du marquage en champs de bits.

Impact du marquage en champs de bits À partir du moment où un système de cache est utilisé, le marquage en champs de bits peut avoir une influence sur les résultats puisque les écritures sont alors regroupées. Cependant, ces résultats ne seront pas présentés, pour deux raisons. La première est que les changements sont relativement minimes. Cela est principalement dû au fait que ce changement a un impact sur tous les ramasse-miettes considérés. La seconde est que ces résultats ne changent rien aux observations faites précédemment.

Outre les observations qui ont pu être faites, l'étude analytique a permis d'évaluer l'importance, toujours du point de vue des performances en accès, des différents para-



(a) `Crypt` (b) `Dhrystone` (c) `Moldyn`

(d) `Raytracer` (e) `Check` (f) `PepTest`

FIGURE 3.5 – Meilleur algorithme en fonction des coûts d'accès, en présence d'un cache de 512 blocs de 32 octets

mètres intervenant dans chaque algorithme, et donc de connaître les paramètres significatifs.

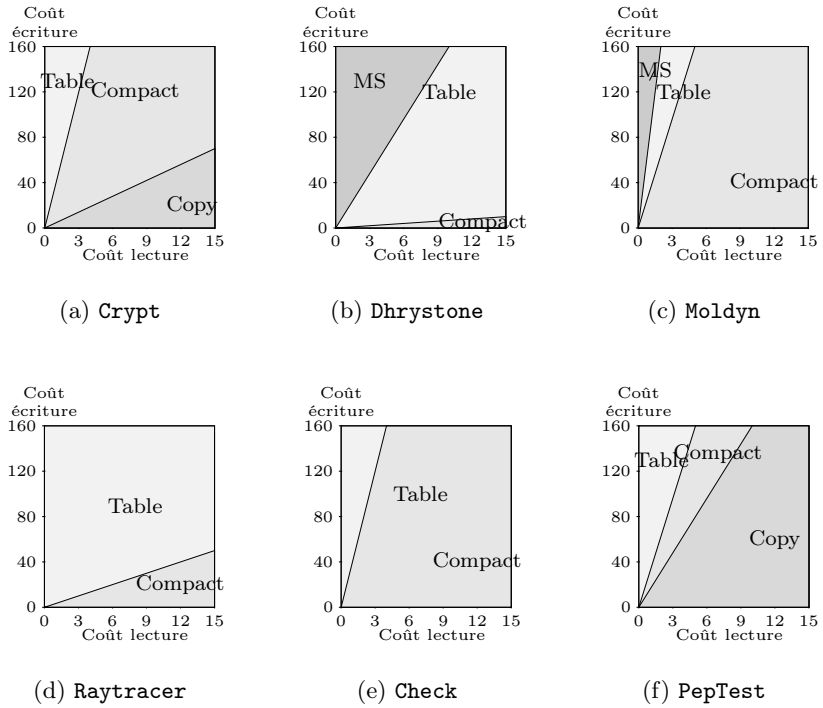
3.4.3 Paramètres significatifs

Puisque les paramètres sont clairement identifiés dans les complexités théoriques, il est possible de savoir lesquels impactent le plus l'efficacité de chaque ramasse-miettes. À cette fin, la table 3.4 présente le changement observé sur le nombre d'accès pour un accroissement de 20% des valeurs moyennes de chaque paramètre. Les résultats confirment clairement les travaux existants indiquant que la survivance est un paramètre clef. On peut observer également un certain impact de l'augmentation de la taille des objets survivants. Effectivement, cela induit une réduction du nombre d'objets survivants et donc une diminution du nombre de lectures d'en-têtes et une réduction du nombre de marquages à effectuer. Au final, les paramètres très spécifiques tels que ceux liés à la gestion des blocs libres dans le ramasse-miettes *Marque et nettoie* ont un impact relativement faibles.

À partir de ces paramètres et du type de mémoire considéré, il serait possible, une fois le profil d'une application effectuée, d'indiquer quel type de ramasse-miettes convient le mieux. En particulier, ces données peuvent servir à l'amélioration de travaux [Soma 04] mettant en place un changement dynamique de ramasse-miettes afin de s'ajuster aux besoins des applications. Ces travaux s'appuient en effet sur le paramètre principal qu'est la survivance ; la prise en compte des paramètres fins présentés permettrait d'obtenir de

	Compactant 1		Compactant 2		Copiant		MS	
	Lect.	Écr.	Lect.	Écr.	Lect.	Écr.	Lect.	Écr.
Taille objet (S_o)	-7%	+3%	-6%	+3%	-8%	0	-10%	0
Taille objet survivant (S_l)	-8.5%	-8%	-11%	-12%	-7%	-7%	-4.5%	-16%
Survivance (r)	+14%	+25%	+15%	+25%	+15%	+25%	+5%	+20%
Références par objet (N_r)	+4%	0	+7%	0	+9%	0	+6%	0
Références valides par objet ($N_{wr,o}$)	+4%	0	+6%	0	+1.5%	0	+1%	0
Proportion compactée (c_b)	-3%	-7%	-2%	-9%	n/a	n/a	n/a	-
Taille objet compacté (S_b)	+0.5%	ϵ	ϵ	+0.5%	n/a	n/a	n/a	-
Blocs libres traversés (N_{fbs})	n/a	n/a	n/a	n/a	n/a	n/a	ϵ	0
Blocs libres formés (N_{fbe})	n/a	n/a	n/a	n/a	n/a	n/a	0	+1%
Blocs libres pendant allocations (N_{fba})	n/a	n/a	n/a	n/a	n/a	n/a	+6%	0
Allocations (N_{al})	n/a	n/a	n/a	n/a	n/a	n/a	+1%	+4%
Mémoire perdue (S_{lm})	n/a	n/a	n/a	n/a	n/a	n/a	-3%	-3%

TABLE 3.4 – Impact d'une augmentation de 20% de la valeur moyenne de chaque paramètre sur le nombre d'accès



(a) `Crypt` (b) `Dhrystone` (c) `Moldyn`

(d) `Raytracer` (e) `Check` (f) `PepTest`

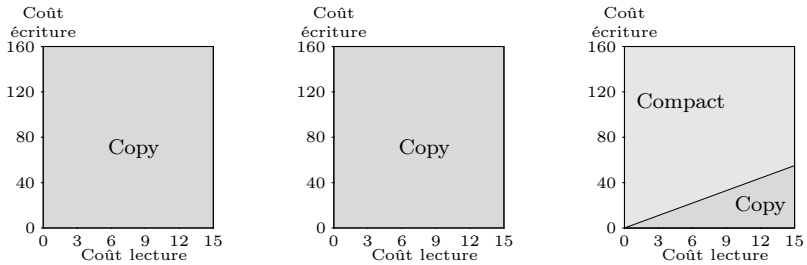
FIGURE 3.6 – Meilleur algorithme en fonction des coûts d'accès, en présence d'un cache de 128 blocs de 4 octets

meilleurs résultats. Néanmoins, cela reste aujourd'hui du domaine des perspectives.

3.5 Conclusion

Les études présentées dans ce chapitre et qui concernent le comportement des ramasse-miettes dans un contexte embarqué sont bien sûr loin d'être complètes, puisqu'elles n'abordent pas un certain nombre de points importants. Ainsi, le nombre de types de ramasse-miettes étudiés est limité aux quelques algorithmes classiques pour lesquels il est possible de trouver un intérêt pour l'embarqué. Les particularités physiques fines de certains types de mémoire telles que la latence d'accès n'ont pas été traitées. Cependant, la précision atteinte dans les études a largement suffi à valider un certain nombre de contributions.

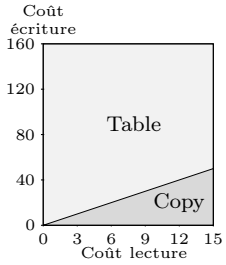
La première contribution de ce chapitre est une étude fine de la complexité des ramasse-miettes relative aux accès en mémoire. Contrairement aux études existantes, celle-ci tient compte du fait que les mémoires n'ont pas les mêmes propriétés, notamment en vitesse de lecture et écriture. Cette étude a été validée expérimentalement et a permis de montrer qu'il n'existe pas de meilleur algorithme de ramasse-miettes, ni pour un type de mémoire donné, ni pour une application donnée. Cette dernière observation confirme des résultats préalablement publiés qui ne différenciaient pas le type de mémoire utilisé. Les résultats exposés montrent que la présence de cache n'infirme pas cette observation. L'importance des paramètres de chaque algorithme a pu être mise en évidence,



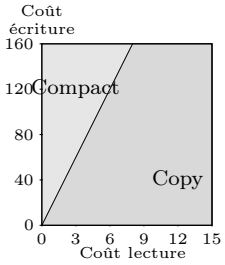
(a) `Crypt`

(b) `Dhrystone`

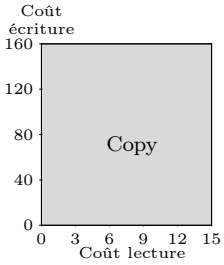
(c) `Moldyn`



(d) `Raytracer`



(e) `Check`



(f) `PepTest`

FIGURE 3.7 – Meilleur algorithme en fonction des coûts d'accès, en présence d'un cache de 16 blocs de 2 Kilo-octets

fournissant ainsi une base sur laquelle choisir un type d'algorithme pour une application donnée. Ces contributions ont fait l'objet d'une première publication [Marq 07c].

Quatrième Chapitre

Proposition d'architecture

« Computers have lots of memory but no imagination. »

Auteur inconnu.

Nous avons vu au chapitre précédent l'illustration que les propriétés des mémoires physiques ainsi que celles des applications ont un impact fort sur la gestion mémoire. L'architecture des gestionnaires mémoire des systèmes embarqués doit être conçue de façon à gérer efficacement les différentes mémoires que l'on peut trouver sur les cibles visées. De plus, un partitionnement purement logiciel est parfois souhaitable. Ce chapitre présente une nouvelle architecture qui a pour but de permettre la gestion de plusieurs espaces mémoire aux propriétés différentes, chacun de la façon qui lui convient le mieux. Tout d'abord, cette solution générale et ses motivations seront présentées. Puis son fonctionnement sera détaillé. Enfin, quelques résultats seront exposés, avant de détailler la dynamicité de l'architecture générale.

4.1 Architecture générale

Le développement des techniques de programmation a mené à l'apparition de techniques permettant de gérer la mémoire de manière transparente pour le concepteur d'applications. On s'intéresse ici aux algorithmes permettant de collecter plusieurs espaces mémoire et à la mise au point de ramasse-miettes pour l'embarqué.

4.1.1 Ramasse-miettes pour l'embarqué

Certains types de ramasse-miettes sont particulièrement adaptés à être utilisés dans le contexte des systèmes embarqués. Les principaux algorithmes ont été détaillés dans le chapitre précédent. Dans cette partie, les avantages de ces algorithmes pour l'embarqué sont mis en avant. Leur fonctionnement est rappelé dans la table 4.1.1. Dans cette table, leur fonctionnement est partitionné en différentes phases. Ce découpage a pour but de rendre la suite de ce chapitre plus claire.

Tout d'abord, l'algorithme *Marque et nettoie* [McCa 60] ne déplace pas les objets. Il effectue donc très peu d'écritures, ce qui le rend particulièrement adapté pour collecter des mémoires très lentes en écriture du type EEPROM.

Les ramasse-miettes copiants [Chen 70] sont généralement très efficaces mais ne sont pas adaptés pour les petites mémoires car la moitié de cet espace est inutilisable pour les applications. De plus, comme la totalité des objets survivants est déplacé, il ne peut être utilisé sur des mémoires ayant de faibles performances en écriture. Cependant, ils peuvent être envisagés pour les mémoires efficaces ayant besoin d'être collectées très souvent.

Les ramasse-miettes compactants [Cohe 83] ont un très faible coût de fonctionnement en mémoire. Ils ont de bonnes propriétés [Baco 04], notamment en termes de fragmentation et de consommation énergétique. Le bénéfice des versions à base de tables de références est d'obtenir de bons effets de cache grâce au groupement des références dans une table. La version 2 est plus efficace (un parcours de mémoire en moins) mais demande plus d'espace pour fonctionner.

L'implémentation CLDC de la machine virtuelle HotSpot de Sun [Micr] utilise un ramasse-miettes 2-générationnel qui effectue des collectes mineures pendant lesquelles les objets survivants sont copiés de la *nurserie* (zone d'allocation) vers le bas du tas.

Application	Fonctionnement	
Marque et Nettoie	Marquage	Depuis les *racines d'atteignabilité*, les objets survivant à la collecte sont atteints via les références entre objets et sont marqués.
	Collecte	Le tas est parcouru, et les espaces inutilisés sont insérés dans la liste de blocs libres.
Copiant	Marquage	Les objets survivants sont parcourus via les références entre objets depuis les racines d'accessibilité. Chaque objet survivant est marqué, déplacé dans la moitié de mémoire non utilisée, et sa nouvelle adresse est stockée à son ancien emplacement.
	Mise à jour	Parcourt tous les objets survivants et met à jour les références qu'ils contiennent.
Compactant	Marquage	Marque tous les objets survivants.
	Préparation	Parcourt le tas et calcule la nouvelle adresse de chaque objet survivant, de telle sorte que ces objets soient placés consécutivement après la collecte. Cette nouvelle adresse est stockée dans un mot supplémentaire de l'en-tête des objets.
	Modification	Parcourt toutes les références de tous les objets survivants et met à jour chacune d'elles.
	Collecte	Déplace chaque objet survivant à sa nouvelle adresse.
Variantes	Marquage	Marque tous les objets survivants.
	Préparation	Parcourt le tas et ajoute la référence de chaque objet survivant à une table de références (dans l'ordre des adresses croissantes). *Version 1* chaque entrée de la table contient un mot ; *Version 2* chaque entrée de la table contient deux mots. L'adresse est stockée dans le premier mot.
	Modification	(*Version 1*) change les références de chaque objet pour qu'elles pointent vers l'entrée de la table de références qui lui correspond (une recherche par dichotomie est nécessaire).
	Collecte	Déplace chaque objet survivant dans le bas du tas, à la suite du précédent objet ainsi compacté : *Version 1* la nouvelle adresse de l'objet écrase l'ancienne dans la table ; *Version 2* la nouvelle adresse est stockée dans le deuxième mot de l'entrée correspondante dans la table.
	Mise à jour	Met à jour chaque référence avec la nouvelle adresse de l'objet pointé : *Version 1* cette nouvelle adresse est contenue à l'endroit pointé par la référence. *Version 2* cette référence est trouvée dans le second mot de l'entrée correspondant à l'ancienne adresse (une recherche par dichotomie est effectuée).
2-générationnel copiant	Marquage	Marque les objets survivants de la nurserie.
	Collecte	Copie les objets survivants de la nurserie vers le bas du tas. La nouvelle adresse est stockée à l'ancien emplacement de l'objet.
	Mise à jour	Met à jour les références.

Ces algorithmes de ramasse-miettes ne sont pas nouveaux. Cependant, ils ne sont pas adaptés à la gestion de mémoires multiples.

4.1.2 Gestion de mémoires multiples

Gérer une mémoire divisée en régions a été le sujet d'un grand nombre de travaux. Certains de ces travaux nécessitent la création et la destruction explicites de régions [Gay 98]. D'autres [Gros 02] spécifient la création et la destruction de régions par des annotations dans le programme. Ces solutions ne répondent donc pas au problème de la collecte automatique de données. L'analyse d'échappement [Choi 99] permet aussi d'allouer les objets dans différentes régions [Qian 02] en fonction de leur propriété d'échappement à un contexte d'exécution. Tous ces travaux supposent que des régions peuvent être créées, et leur taille fixée en fonction de certains paramètres. Ce n'est pas notre cas puisque les différentes zones mémoire présentes sont des mémoires physiques de tailles limitées.

Actuellement, deux modèles se rapprochent du but que l'on souhaite atteindre, c'est à dire : gérer le placement de données dans différentes zones mémoire fixes, ayant des propriétés différentes. Premièrement, les ramasse-miettes *générationnels* [Lieb 83] trient les données selon leur âge dans différentes régions appélées « générations ». Les allocations sont faites dans la plus jeune génération (appelée la *nurserie*), et les objets survivant à une collecte sont promus dans une génération plus ancienne car ils sont supposés avoir encore une longue vie [Unga 84]. En effet, l'argument en faveur des ramasse-miettes générationnels est que les générations plus anciennes seront peu collectées car la majorité des objets meurent jeunes. Cependant, cet argument n'est pas suffisant quand on adresse des zones mémoire ayant différentes propriétés physiques, et notamment des temps d'accès très différents. De très loin, on peut diviser le temps d'exécution d'une application entre le temps d'exécution « utile », et le temps passé à collecter les données. Par conséquent, il ne suffit pas d'accélérer le temps de collecte, car les temps d'accès peuvent pénaliser le reste de l'exécution. Pour illustrer cela, il suffit de considérer une application modifiant un tableau d'entiers pendant toute son exécution. Selon ce modèle générationnel, ce tableau va se retrouver dans une génération ancienne. Si cette génération est placée dans une mémoire physique très lente — ce qui est probable puisque les mémoires efficaces sont présentes en moindre quantité —, les accès au tableau vont être ralentis, et par conséquent l'application également.

Deuxièmement, le modèle JavaCard[Micr 03] utilise un schéma de placement fixe, dans lequel l'allocation par défaut se fait en EEPROM. Le concepteur d'applications peut utiliser une certaine API pour allouer des données en RAM. Cette approche est complètement spécifique au matériel sous-jacent (RAM + EEPROM en l'occurence), et ne permet pas le portage d'applications d'une plateforme à une autre. Ensuite, cela demande au programmeur d'avoir une connaissance poussée du matériel. En effet, il doit non seulement connaître l'existence des différentes mémoires sous-jacentes, mais aussi leurs propriétés s'il veut pouvoir produire du code efficace. De plus, la solution JavaCard ne permet pas le déplacement d'objets d'une zone mémoire à une autre, ce qui est une approche limitante pour différentes cibles tels que les capteurs, ou les systèmes JavaCard de prochaine génération, qui vont introduire l'utilisation de mémoires plus nombreuses, avec des propriétés très différentes. Une autre limitation de cette approche est qu'elle

ne gère pas le partitionnement logique. Or il peut être souhaitable de gérer des régions logiques ayant différentes propriétés (par exemple une nurserie).

Afin d'adresser ces différents problèmes, j'ai mis au point une architecture nouvelle reposant, de très loin, sur une proposition : un gestionnaire par partition.

4.1.3 Proposition : un gestionnaire par partition

Je vais maintenant détailler une solution permettant d'adresser ces différents problèmes. Dans ce modèle, chaque mémoire est vue comme un espace aux propriétés spécifiques, qu'il convient donc de gérer de façon spécifique. De plus, ce modèle supporte le partitionnement logique, permettant de diviser les espaces physiques en espaces logiques. Ceci afin de pouvoir gérer des espaces dont les propriétés logiques sont différentes. Par exemple, une nurserie.

Par comparaison au modèle générationnel, on ne va pas considérer l'âge d'un objet comme étant le seul critère de placement. Plutôt que de fixer le rôle de chaque région en fonction de l'âge des objets qu'elle contient, les régions vont contenir les objets adaptés, au regard de leurs propriétés physiques que ce soit en termes de performances, sécurité des données, ou autre. Les objets vont donc éventuellement évoluer entre les différentes zones mémoires, que l'on appellera dans la suite « partitions », en fonction de leurs propriétés, et des propriétés des mémoires sous-jacentes (figure 4.1.

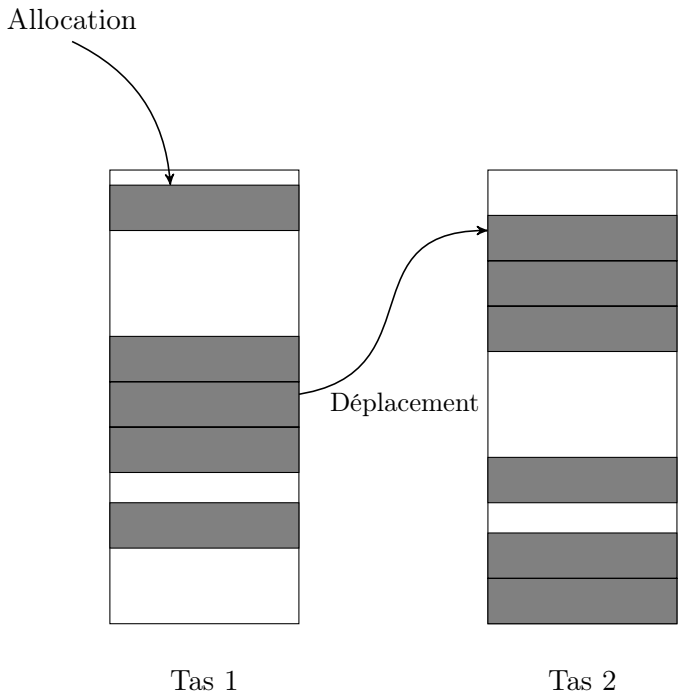


FIGURE 4.1 – Allocation et déplacement d'objets

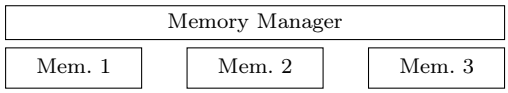
De plus, on a vu dans le chapitre précédent que les propriétés physiques des mémoires avaient un gros impact sur les performances des ramasse-miettes. Chaque type de mémoire ne peut donc pas être géré de la même façon. De plus, même si on ne l'a vérifié quantitativement que pour les performances et l'énergie, cela est vrai également

pour d'autres aspects. Par exemple, chaque octet de l'EEPROM ne pouvant être écrit qu'un nombre limité de fois, il convient de minimiser le nombre d'écritures dans ce type mémoire. Ou encore, il peut-être souhaitable de crypter les données résidant en mémoire persistante.

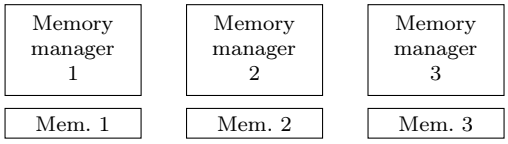
Tout cela m'amène à proposer une solution permettant de spécialiser la gestion mémoire en fonction des applications et en fonction du matériel. Le principe général réside dans ces 2 points :

- *Les mémoires ont des propriétés différentes ; chacune d'elle va donc être gérée par un gestionnaire mémoire spécifique ;*
- *Le placement des données est une préoccupation transversale.*

Ce dernier point est détaillé dans le chapitre 5. L'idée générale est donc d'assigner un système de gestion mémoire (comprenant un allocateur ainsi que l'utilisation d'un ramasse-miettes spécifique) à chaque zone mémoire, dans le but de gérer indépendamment leurs spécificités. Un des objectifs notables est que chaque accès en écriture dans une partition soit effectué par le gestionnaire en charge de celle-ci. Les accès en lecture sont moins problématiques car concernent moins de types de mémoire. Quoiqu'il en soit, je m'efforce dans la suite de traiter ce problème. La figure 4.1.3 illustre, de façon un peu simpliste, la comparaison entre mon approche distribuée et l'approche traditionnelle dans laquelle le gestionnaire mémoire est dédié à une architecture donnée.



(a) Approche des systèmes existants



(b) Mon approche

FIGURE 4.2 – Comparaison de mon approche avec l'approche traditionnelle

Une seule action peut-être demandée au gestionnaire mémoire : allouer un objet. Quand il est nécessaire de libérer de l'espace, une collecte est lancée. Dans mon modèle, les allocations et la collecte sont réalisées par le gestionnaire en charge de la partition concernée. Cependant, si l'on y regarde d'un peu plus près, certaines phases sont transversales et ne peuvent être réalisées de façon distribuée.

- Tout d'abord, étant donné la multiplicité des espaces mémoire, il est nécessaire de choisir la partition dans laquelle placer les objets ;
- Ensuite, la phase de marquage des objets survivants nécessite de suivre toutes les références inter-partitions ;
- Enfin, la mise à jour des références inter-partitions pendant une collecte implique une interaction entre plusieurs gestionnaires.

Deux mécanismes sont mis en place pour gérer ces actions transversales. Tout d'abord

un moyen de décider à l'exécution où placer les objets. Ce moyen est l'utilisation d'un composant particulier appelé *Placeur*. Ensuite, la mise au point d'un algorithme de marquage distribué sur l'ensemble des gestionnaires mémoire. Cet algorithme est détaillé en Section 4.2.1. Enfin, afin de gérer la modification/mise à jour des références, l'idée de base est de découper les phases de collecte en phases permettant aux gestionnaires de coopérer. Un composant particulier que l'on appellera *Collecteur* est chargé de déclencher ces différentes phases. Le marquage étant l'une de ces phases, c'est également lui qui conduira le marquage des objets. À noter que le *Placeur* et le *Collecteur* sont les deux seules parties visibles du reste du système : le *Placeur* peut être appelé pour allouer des objets, le *Collecteur* pour effectuer une collecte demandée explicitement par le programmeur.

La figure 4.3 montre le schéma général de cette architecture. On y trouve un gestionnaire par partition, et les deux composants particuliers qui communiquent avec ceux-ci via une interface dédiée. Les interactions entre ces différents composants sont détaillées dans la suite, qui expose le fonctionnement détaillé de l'architecture.

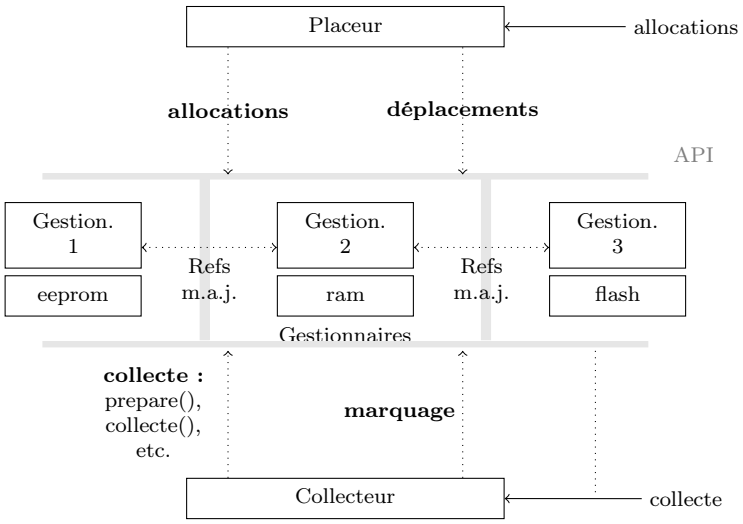


FIGURE 4.3 – Architecture générale

4.2 Fonctionnement détaillé de l'architecture

La solution proposée est maintenant détaillée. Les différentes fonctions du gestionnaire mémoire sont expliquées (allocations, collecte). Une proposition d'implémentation est proposée. Le problème du marquage de différentes partitions est adressé par l'emploi d'un algorithme de marquage distribué.

4.2.1 Un algorithme de marquage distribué

La phase de marquage est problématique dans le cas de la gestion multi-tas. En effet, durant cette phase, des écritures peuvent être effectuées dans les différentes zones mémoire, et les références inter-partitions doivent être suivies. Or, on souhaite que chaque

gestionnaire ait le contrôle des écritures faites dans sa partition. Il est donc nécessaire d'utiliser un algorithme de marquage prenant cette contrainte en compte. Pour ce faire, je me suis inspiré des algorithmes de ramasse-miettes dits « distribués ». Ces algorithmes ont initialement été mis au point afin de répondre au problème de la collecte de données dans un système multi-processeurs.

Comme on peut le lire dans [Jone 96], deux types d'algorithmes distribués existent. Les premiers sont basés sur l'algorithme de collecte de Dijkstra [Dijk 78] et utilisent un marquage incrémental ainsi que des barrières en écriture, ce que l'on se refuse ici. Ensuite, l'algorithme de Ali [Moha 84] adresse les mêmes objectifs (gérer séparément les tas mémoire), mais la consommation mémoire nécessaire rend inapplicable cette solution, puisque le principe est de constituer une liste des références partant de chaque tas afin de servir de racines d'atteignabilité pour les tas suivants. Hughes a imaginé [Hugh 85] une solution basée sur un marquage incrémental et le stockage de dates de création à l'intérieur d'objets. Premièrement on ne souhaite pas se limiter à un marquage incrémental (plus coûteux en ressources). Et deuxièmement le stockage additionnel est impraticable, et n'est pas nécessaire dans notre cas puisque la gestion des tas est distribuée dans l'espace et pas dans le temps. Ce dernier problème est général aux algorithmes distribués. L'algorithme présenté dans la suite tire parti du fait que l'on n'est pas dans un contexte de gestion de la concurrence, donc il ne nécessite aucune synchronisation multi-processus. De plus, les solutions distribuées considèrent généralement que l'organisation des données dans les différents tas est soit la même soit connue (solutions centralisées).

Un algorithme de marquage standard et efficace a été présenté dans le chapitre précédent (figure 4.4(a)). Le principe est simple : les racines d'atteignabilité sont empilées. Puis, tant que la pile de références n'est pas vide, une référence est dépilée, l'objet pointé est marqué et scanné, ses fils sont empilés. De cette manière, tous les objets accessibles se trouvent marqués à la fin de la phase de marquage. La figure 4.4(b) montre le pseudo-code de l'algorithme de marquage distribué entre les gestionnaires mémoire. L'algorithme va utiliser des services fournis par les gestionnaires mémoire (en gras sur la figure) afin de laisser les écritures à la charge de ceux-ci, ainsi que le choix sur la façon de marquer un objet.

Le principe est donc similaire à celui de l'algorithme standard, puisque les références sont suivies depuis les racines d'atteignabilité. Cependant, la boucle globale a changé, puisqu'elle consiste à itérer sur les gestionnaires mémoire tant qu'au moins l'un de ceux-ci a atteint un nouvel objet (devant donc être scanné pour éventuellement marquer ses fils). On a donc :

- à chaque itération, le gestionnaire mémoire courant est notifié (`mark_heap()`) qu'il lui faut s'occuper de ses objets marqués (c'est-à-dire les objets contenus dans la partition qu'il a en charge, qui ont été atteints donc marqués, mais dont les fils n'ont pas encore suivis) ;
- lorsqu'un objet est atteint, le gestionnaire de la partition dans lequel est situé l'objet est notifié qu'il lui faut marquer l'objet (**`mark_object()`**).

La figure 4.4(b) montre un exemple typique d'implémentation des fonctions `mark_heap()` et `mark_object`, où une pile de références classique est utilisée. Dans le cas d'un gestionnaire mémoire de type copiant, l'objet est copié dans le demi-tas inutilisé (et marqué).

À propos des fonctions `mark_heap()`, elle peut en réalité être la même pour tous les gestionnaires à deux conditions :

– tous utilisent une pile de marquage ;
– toutes les mémoires sous-jacentes sont accessibles directement en lecture, sans accesseur spécifique.

Dans l'implémentation réalisée, c'est ce qui a été considéré, étant donné que le marquage par pile est efficace, et que la condition sur l'accessibilité en lecture empêche seulement l'utilisation de mémoire de type Flash NOR, qu'il est difficile aujourd'hui de considérer pour être employée comme tas. À noter que si toutes les mémoires sont accessibles en écriture de la même façon, alors l'utilisation de la fonction `mark_object()` est inutile et son contenu peut être intégré à `mark_heap()`. L'algorithme permet alors juste d'utiliser des méthodes de marquage différentes pour chaque partition. Bien que cela puisse se révéler intéressant le bénéfice d'un tel mécanisme n'a pas été étudié.

```
mark() =
   mark_stack = empty
   for R in Roots
      mark_bit(R) = marked
      push(R, mark_stack)
   mark_heap()

   while mark_stack ≠ empty
      N = pop(mark_stack)
      for M in Children(N)
         if mark_bit(M) == unmarked
            mark_bit(M) = marked
            push(M, mark_stack
```

(a) Algorithme standard

```
mark() =
   done = false
   for MM in Managers
      MM− >prepareToMark()

   for R in Roots
      for O in Children(R)
         MM = getManagerOf(O)
         MM− >mark_object(O)
   while not done
      done = true
      for MM in Managers
         done = not MM− >mark_heap()

boolean mark_heap() =
   while MM− >mark_stack ≠ empty
      N = pop(MM− >mark_stack)
      for O in Children(N)
         OM = getManagerOf(O)
            OM− >mark_object(O)
            marked = true
   return marked

boolean mark_object(Object O) =
   if mark_bit(O) = unmarked
      push(O, MM->mark_stack)
```

(b) Algorithme distribué

FIGURE 4.4 – Comparaison entre l'algorithme de marquage classique et l'algorithme de marquage distribué

Une deuxième partie importante des fonctions d'un gestionaire mémoire est la réalisation des allocations.

4.2.2 Allocations

Le problème, à l'allocation, est de décider du placement de l'objet. Les stratégies de placement à ce niveau sont nombreuses, et varient selon les architectures et les appli-

cations. C'est aussi pour cette raison que le modèle présenté est interéssant : l'aspect placement est complètement isolé dans le *Placeur*. En cas de portage, seul ce composant est à modifier. Le chapitre 5 traite spécifiquement du problème du placement.

Quand un nouvel objet doit être alloué (*i.e.* exécution d'un bytecode `NEW` par l'interpréteur de bytecodes), l'allocateur du *Placeur* est appelé. Cet allocateur détermine dans quelle partition le nouvel objet doit être alloué. Une fois ce choix fait, l'allocateur du gestionnaire mémoire en charge de la partition choisie est appelé. Chaque gestionnaire doit fournir ce service d'allocation.

Gérer le placement des objets dès l'allocation (plutôt que de tout allouer par défaut dans une même partition) permet d'éviter d'inutiles déplacements de données, ainsi que d'éviter le remplissage trop rapide d'une zone mémoire particulière. Enfin, cela permet surtout de placer, dès sa création, un objet dans une mémoire aux propriétés adaptées à l'objet. Aucune contrainte n'est donnée à l'allocateur pour le choix de la partition dans laquelle allouer un nouvel objet. Si l'allocateur ne peut allouer faute de place, il déclenche une phase de collecte de données.

4.2.3 Collecte de données

Après avoir adressé les problèmes de l'allocation et du marquage, il reste à gérer le problème des références croisées lors de la collecte de données. Pour cela, l'approche adoptée est d'essayer de synchroniser les gestionnaires mémoire afin de gérer au mieux les effets de bords liés à la collecte.

A partir des exemples typiques de ramasse-miettes présentés en section 4.1.1, il est possible de trouver des similitudes dans le comportement des ramasse-miettes. Par exemple, ils contiennent presque tous une phase de collecte proprement dite. Pour le *Marque et nettoie*, il s'agit de nettoyer l'espace occupé par les objets non survivants. Pour les algorithmes compactants, les données sont compactées vers la base du tas. L'idée est donc de définir un ensemble de phases ayant une sémantique, un rôle, bien particulier par rapport à la collecte totale. Cette sémantique doit être assez large pour que chaque type de ramasse-miettes puisse s'exécuter en s'inscrivant dans le cadre de travail ainsi défini. Mais elle doit être assez précise pour que le respect de celle-ci implique l'exécution correcte d'une collecte.

Interactions entre gestionnaires mémoire

Quand une collecte est réalisée, en fonction du type de ramasse-miettes utilisé, les objets peuvent potentiellement bouger. Ceci invalide les références d'autres partitions pointant sur ces objets. En conséquence, les références doivent être mises à jour dans toutes les partitions. Deux solutions à ce problème peuvent être envisagées. Premièrement, implémenter des barrières [Zorn 90] en lecture ou écriture (avec un mécanisme de marquage du tas par exemple [Hosk 93]). Deuxièmement, mettre à jour les références quand elles ont été modifiées. Les deux solutions sont compatibles avec la solution proposée, mais le mécanisme de gestion de références par barrière ne sera pas détaillé. En effet, les barrières en écriture et les barrières en lecture sont coûteuses à la fois en espace et en temps.

De plus, on souhaite conserver la propriété de déplacement d'objets entre zones mé-

moire, afin de pouvoir adapter le placement des données après leur allocation, ou pour libérer des zones mémoire. Cette capacité est utilisée par exemple par les ramasse-miettes générationnels. C'est le *Placeur* qui va être responsable de déplacer un objet ou non car les gestionnaires doivent rester indépendants les uns des autres. Par exemple, on ne souhaite pas devoir modifier un gestionnaire mémoire parce qu'un autre est présent sur une architecture ou pour une application donnée.

Le modèle

Si on examine la table 4.1.1 précédente, on peut trouver un ensemble d'étapes dans lequel le fonctionnement de tout ramasse-miettes peut s'inscrire ; chacune de ces étapes est éventuellement vide. Si chaque gestionnaire mémoire respecte cet ensemble d'étapes, alors une collecte fonctionne selon le modèle suivant. Lorsqu'un gestionnaire mémoire M a besoin de collecter les données de sa partition, il le notifie au *Collecteur*. Celui-ci déclenche les différentes phases permettant à la fois au gestionnaire concerné de réaliser sa collecte, et aux autres de gérer les effets de bord de cette collecte. Ces phases sont les suivantes :

Préparation durant cette phase, M effectue toutes les opérations permettant de le préparer aux phases suivantes. Par exemple, certains ramasse-miettes compactants construisent une table de références ;

Modification pour chaque gestionnaire mémoire, les références contenues dans la partition qu'il a en charge sont modifiées (si M nécessite cela ; c'est le cas du compactant). Une condition doit être vérifiée par M s'il nécessite cette étape pour collecter : pour toute référence valide pointant vers un objet de sa partition, M est capable de donner la référence modifiée. Par exemple, les ramasse-miettes compactant font pointer toutes les références vers la table de référence ;

Collecte la collecte des données est effectuée durant cette phase. Pour chaque objet survivant, M demande au *Monitor* si l'objet doit être déplacé dans une autre partition ;

Mise à jour pour chaque gestionnaire mémoire, les références contenues dans la partition qu'il a en charge sont mises à jour (si M_0 a déplacé des objets). Une condition doit être vérifiée par M s'il nécessite cette étape pour collecter : pour toute référence (préalablement modifiée le cas échéant) pointant vers sa partition, M est capable de donner la référence mise à jour. Cette référence est alors valide.

Chaque gestionnaire mémoire doit fournir les quatre fonctions effectuant ces phases. Quand une collecte doit être effectuée, le *Collecteur* appelle successivement ces fonctions afin de réaliser la collecte. Les étapes 1 et 3 ne sont réalisées que par M_0, alors que les étapes 2 et 4 permettent aux gestionnaires de conserver leurs références valides. Cette organisation assure une synchronisation correcte entre les différents gestionnaires. Elle assure en particulier que ce n'est qu'une fois M prêt que l'on va modifier/mettre à jour les références.

Au final, chaque gestionnaire mémoire peut être vu comme un composant possédant un certain nombre de fonctionnalités grâce auxquelles il va pouvoir interagir avec les autres composants, à savoir le *placeur*, le *collecteur*, et bien sûr les autres gestionnaires. Ces composants forment un macro composant, qui fournit 3 services : allocation, collecte

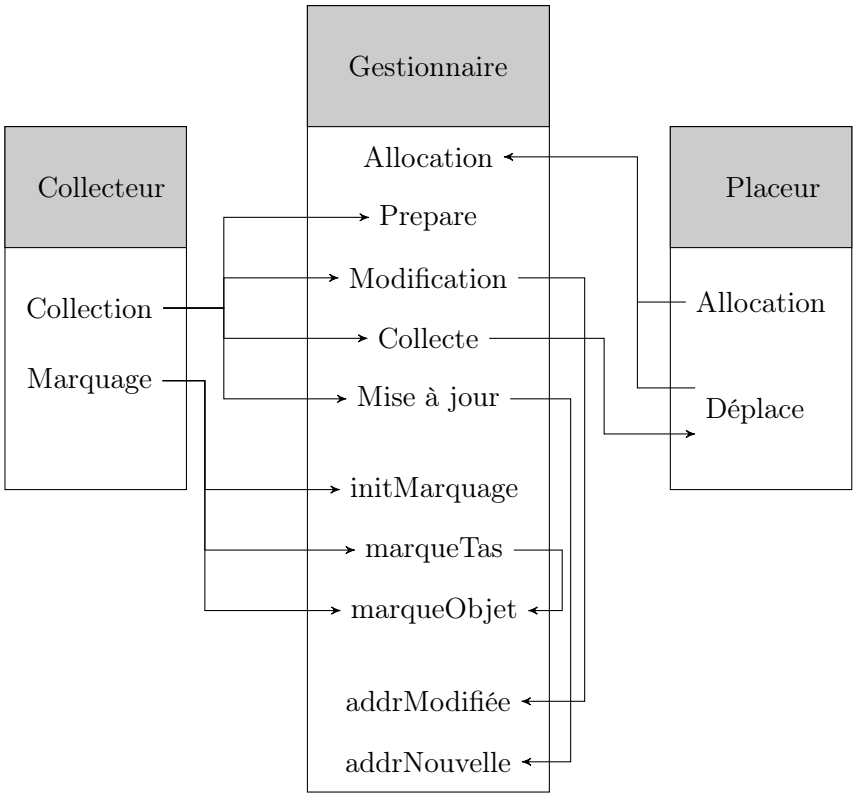


FIGURE 4.5 – Le gestionnaire mémoire vu comme un composant

et marquage. Ce dernier service est utilisé dans le cas où un processus est dédié au marquage incrémental.

Ce modèle ne repose pas du tout sur l'utilisation de barrières d'accès, conformément à ce qui a été annoncé précédemment. Cependant, l'utilisation de telles méthodes n'est pas interdite par la solution proposée. Il n'en a pas du tout été fait usage dans l'implémentation réalisée.

4.2.4 Implémentation réalisée

Le modèle de gestion de mémoires proposé précédemment a été vérifié et testé au moyen d'une implémentation complète dans le système Java JITS [JITS]. Cette implémentation est partiellement présentée ci-après afin de détailler la proposition qui est faite, donner des précisions sur les résultats annoncés en Section 4.3, et donner le cadre des expériences réalisées autour des travaux de cette thèse. Le langage utilisé est le langage C. En effet, bien que la majeure partie du projet JITS soit constitué de code Java, il est nécessaire d'utiliser du code natif C pour plusieurs raisons. Tout d'abord il est impossible de forger un pointeur en Java. De plus, la programmation en C donne accès aux couches basses du système, ce qui permet des optimisations impossibles à réaliser en Java.

L'architecture globale d'un système embarqué JITS est donnée par la figure 4.6. Les couches basses du système (code C) sont composées d'un interpréteur de bytecodes, un ensemble de fonctions natives auxquelles font appel les APIs Java utilisées au-dessus de

ces couches basses, et enfin un gestionnaire mémoire. C'est ce composant qui va être détaillé ici.

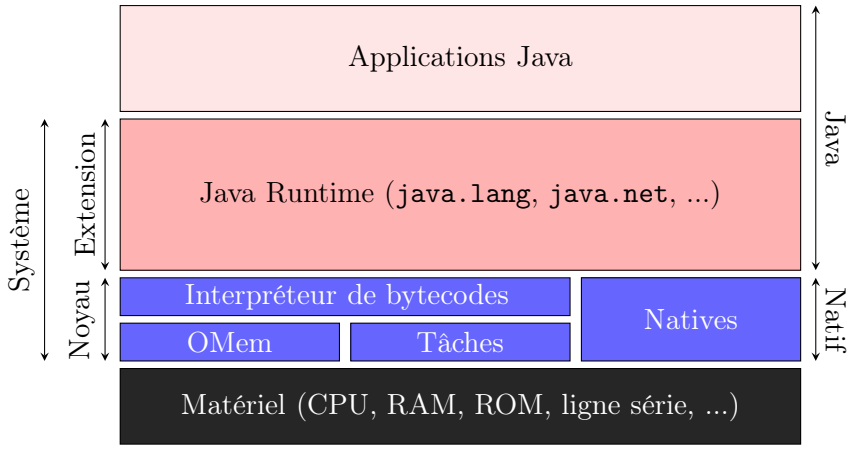


FIGURE 4.6 – Architecture JITS

Tout d'abord, il convient de présenter la structure des tas d'objets manipulés par ce gestionnaire mémoire et utilisés au-dessus de la notion de partition. Quand le système Java démarre, certaines données initiales du système (code et structures de données) sont copiées à partir d'une mémoire persistante vers une mémoire de travail (voir chapitre 6). Au démarrage, chaque partition contient donc un tas d'objets dont la structure est illustrée par la figure 4.7. Cette structure est composée de deux ensembles d'objets. À la base du tas, on trouve la partie « fixe » du tas. Cette partie contient tous les objets qui ne peuvent pas bouger car ils sont référencés par des objets situés en mémoire non-réinscriptible et ne peuvent donc modifier leurs références.

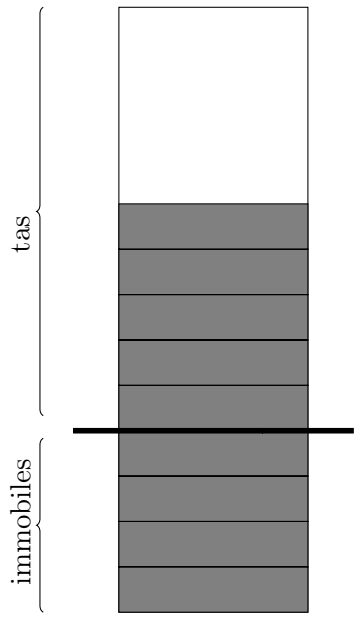


FIGURE 4.7 – Structure d'un tas

Chaque gestionnaire mémoire gère un tas de ce type, et organise les données à l'intérieur à sa manière une fois le système démarré. En cours d'exécution, chaque gestionnaire mémoire va utiliser un certain nombre de variables pour stocker diverses informations telles que le nombre d'objets contenus dans le tas, ou encore l'indicateur de sommet de tas. De plus, chacun des gestionnaires doit fournir un accès aux services présentés précédemment, et nécessaires à la bonne coopération des divers gestionnaires. Dans l'implémentation réalisée, ces services sont proposés sous la forme de fonctions, accessibles via des pointeurs de fonction C. Ces deux types de données spécifiques à chaque gestionnaire sont stockées dans une structure C, présentée en figure 4.8.

Cette structure contient toutes les variables qui ne sont jamais modifiées en cours d'exécution (typiquement les pointeurs de fonctions) ainsi qu'un pointeur vers une structure contenant les variables modifiables en cours d'exécution par le gestionnaire mémoire. Ce procédé permet de distinguer le contenu pouvant être placé en mémoire non-réinscriptible du contenu qui doit se trouver en mémoire modifiable. Dans l'implémentation réalisée, cette deuxième structure contient seulement deux variables, pour la simple et bonne raison que cela suffit à tous les gestionnaires implémentés. Il s'agit premièrement d'un compteur d'objets, utilisé principalement pour savoir s'il reste assez de place dans le tas, en particulier pour les ramasse-miettes réservant un certain nombre de mots par objet (compactant par exemple). Et deuxièmement d'un pointeur indiquant le plus souvent le sommet du tas.

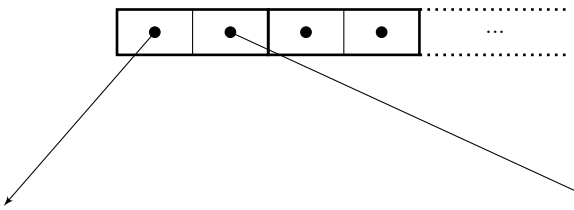
L'implémentation présentée est une implémentation fonctionnelle du modèle exposé préalablement. Cette implémentation supporte également quelques propriétés annexes.

4.2.5 Propriétés annexes

Il est intéressant de noter que l'architecture proposée supporte deux propriétés qui peuvent avoir leurs bénéfices.

Tout d'abord, l'analyse d'échappement est supportée [Grim 07]. Celle-ci permet d'allouer dans une zone mémoire spéciale les objets qui n'échappent pas à un contexte d'exécution donné, typiquemement une fonction. Ces objets sont détruits en même temps que le contexte en question, typiquement donc quand la fonction retourne. Ce mécanisme permet de réduire le nombre de collectes nécessaires si le système de gestion mémoire le supporte, ce qui est le cas ici.

Ensuite, l'architecture est capable de gérer toutes les méta-données — typiquement les objets que sont les classes, méthodes et champs en Java — comme des objets normaux. Il est donc tout à fait possible de déplacer les méta-données entre des partitions. Ainsi, prenons le cas de Java, il est possible de placer des classes d'une manière adaptée (la mettre en mémoire efficace par exemple, pour améliorer les performances d'une application), et même de collecter les classes d'une application terminée. En effet nul besoin de mécanisme spécifique, il suffit de supprimer sa référence dans le chargeur de classe, la collecte fera le reste. Une collecte typique dans l'architecture est maintenant détaillée dans un exemple.



```
struct heap_variables_s {

  /* identifiant de gestionnaire */
  const unsigned int id;

  /* Pointeur de début de tas. */
  unsigned char* heapBase;

  /* Taille du tas */
  const unsigned int memsize;

  /* Phase de préparation */
  void (*prepare) ( void );

  /* Phase de modification */
  void (*modify) (struct heap_variables_s *);

  /* Phase de collecte */
  void (*collect) ( void );

  /* Phase de mise à jour */
  void (*update) (struct heap_variables_s *);

  /* Donne une adresse modifiée */
  struct java_lang_Object*\\
  (*getModifiedAddress) (struct java_lang_Object *aref);

  /* Donne une adresse mise à jour */
  struct java_lang_Object*
  (*getUpdatedAddress) (struct java_lang_Object *aref);

};
```

```
struct modifiable_heap_variables_s {

 /* Marqueur ( de fin de tas généralement)*/
  unsigned char* toh;

  /* Nombre d'objets dans le tas */
  unsigned int nbObjects;
};
```

FIGURE 4.8 – Structures de données associées à un gestionnaire mémoire

4.2.6 Exemple

La figure 4.9 illustre le fonctionnement d'une collecte de données. Dans le scénario joué, trois gestionnaires mémoire existent (mm1, mm2, mm3). mm1 utilise un ramasse-miettes compactant dans sa version avec en-tête étendu, et c'est ce gestionnaire mémoire qui doit collecter ses données. Il le notifie au *Collecteur* qui déclenche alors la série d'étapes nécessaires. Tout d'abord, la phase de marquage décrite précédemment est lancée. Après cela, le *Collecteur* indique à mm1 de se préparer ; pendant cette phase, mm1 calcule les futures adresses des objets de sa partition. Ensuite, la phase de modification des références est déclenchée : cette fonction est appelée pour chaque gestionnaire mémoire. Durant cette phase, mm2 et mm3 qui ne sont pas en train de collecter leurs données, mettent à jour leurs références en demandant, pour chacune d'elles, à mm1 la nouvelle adresse de l'objet pointé. Cela est possible puisque la phase de préparation a permis le calcul de ces nouvelles adresses. Enfin, le *Collecteur* indique à mm1 qu'il peut réaliser sa collecte proprement dite, en l'occurence le compactage des objets.

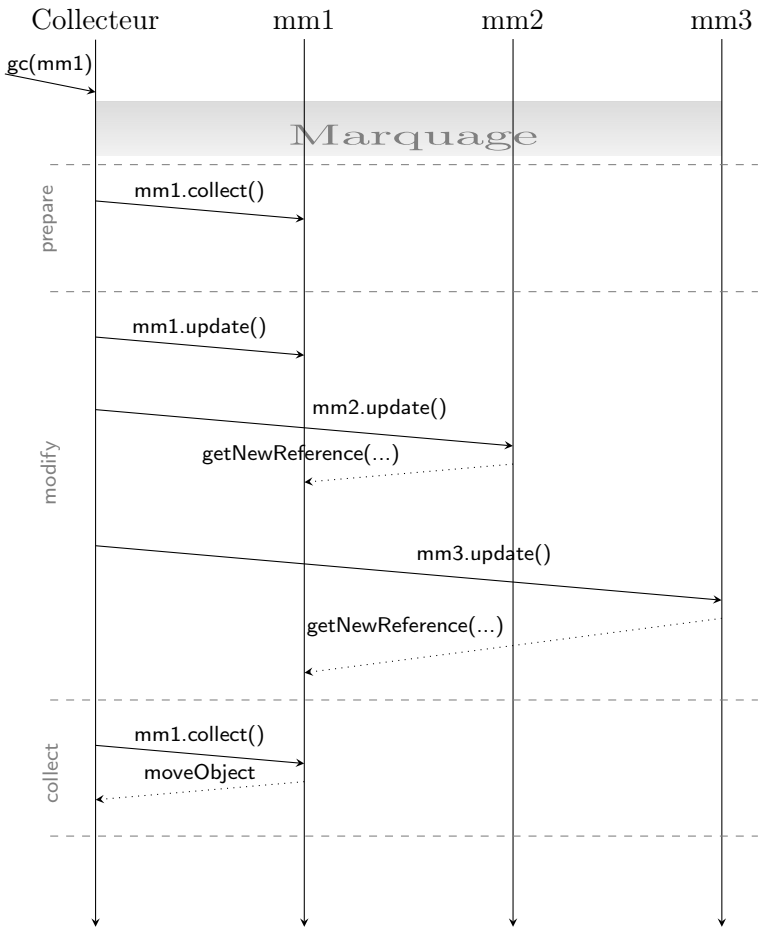


FIGURE 4.9 – Un scénario où mm1 réalise la collecte de ses données

4.3 Résultats expérimentaux

Afin d'évaluer l'architecture proposée, et puisque les systèmes embarqués peuvent être contraints en ressources, il est nécessaire d'évaluer les trois choses suivantes : premièrement son coût à l'exécution, deuxièmement les tailles du code et des données de fonctionnement, et enfin la flexibilité de l'architecture.

4.3.1 Flexibilité de l'architecture

Six systèmes de gestion mémoire (allocateur + ramasse-miettes) ont été implémentés. Il s'agit des six systèmes décrits au début du chapitre en table 4.1.1. Le ramasse-miettes *2-générationnel copiant* n'a malheureusement pas pu être mesuré dans la suite car il n'est pas encore pleinement fonctionnel. La troisième variante du ramasse-miettes compactant n'a pas été mesurée car une troisième variante apportait peu. Cependant, ils ont tous pu s'insérer sans problèmes dans le canevas défini. De plus, Guillaume Salagnac du Laboratoire Vérimag a pu, sans adaptation aucune de l'architecture, implémenter son propre gestionnaire mémoire et l'insérer dans le canevas. En fait, il ne s'est trouvé aucun ramasse-miettes pour mettre en défaut cette organisation. Tout cela montre que le modèle défini pour une collecte est tout à fait adapté et non restrictif. Il nécessite cependant une évaluation des tailles du code et des données de fonctionnement.

4.3.2 Tailles du code et des données de fonctionnement

Dans l'implémentation réalisée, aucun travail poussé n'a été fait pour diminuer la taille du code. On y trouve ainsi parfois, par souci de clarté principalement, des parties non factorisées alors qu'elles le pourraient aisément. Principalement, un certain nombre de fonctions que doivent implémenter tous les ramasse-miettes afin de pouvoir coopérer avec les autres se ressemblent fortement mais sont dupliquées. Cependant, les tailles du code données dans la Table 4.1 permettent de se faire une idée de l'espace nécessaire à l'utilisation de l'architecture proposée. En résumé, on trouve donc environ 10 Ko d'utilitaires auxquels il faut rajouter entre 2.5 et 3.5 Ko par gestionnaire mémoire. Parmi ces 10 Ko, on trouve pour les trois quarts la gestion du marquage. La place importante demandée par cette fonctionnalité est pour une très grande part due au marquage des piles d'exécution qui est complexe à réaliser dans la plateforme considérée.

À ces tailles du code, il faut ajouter l'espace nécessaire pour stocker les structures décrivant les gestionnaires mémoire. L'espace demandé est, pour les données immutables, de douze fois la place de coder un entier ou une adresse, additionnée aux quatre pointeurs nécessaires en mémoire modifiable. Soit en tout 64 octets si l'on considère qu'entiers et adresses sont codés sur quatre octets. Il faut donc compter au minimum treize kilo-octets pour embarquer un gestionnaire mémoire complet implémenté selon le modèle proposé ; la moitié serait tout de même due au marquage et n'a rien à voir avec le modèle proposé.

Ces chiffres montrent que l'architecture proposée est embarquable dans des appareils contraints, du moins si son surcoût n'est pas trop grand en terme d'efficacité.

Fonction	Taille
Marquage	7.4 Ko
Placeur	0.2 Ko
Collecteur	0.2 Ko
Autres utilitaires	2.3 Ko
Ramasse-miettes copiant	2.6 Ko
Ramasse-miettes compactant	2.8 Ko
Ramasse-miettes Marque et nettoie	2.5 Ko
Ramasse-miettes compactant (variante 1)	2.8 Ko
Ramasse-miettes compactant (variante 2)	3.5 Ko
Ramasse-miettes compactant (variante 2)	3.5 Ko

TABLE 4.1 – Tailles du code du gestionnaire de mémoire implémenté

	Copiant	Compactant	Compactant (variante)	Marque et nettoie
Moyenne	+9.1%	+7.7%	+8.7%	+8.8%
Max	+10.2%	+8%	+13.3%	+10.8%
Min	+7%	+7.4%	+6.1%	+7.8%

TABLE 4.2 – Surcoût de l'algorithme de marquage distribué

4.3.3 Efficacité

Afin d'évaluer le surcoût à l'exécution dû à l'architecture dans laquelle chaque gestionnaire doit s'insérer, il a été nécessaire de comparer les temps d'exécution des différents gestionaires dans l'architecture et en dehors. Ainsi, pour chaque type de ramasse-miettes, le surcoût de l'architecture a été mesuré, en pourcentage de temps par rapport à l'exécution de ce ramasse-miettes en dehors de l'architecture. Dans cette exécution de référence, on ne fait plus appel à des pointeurs de fonctions, et les variables sont accédées directement. En résumé, la comparaison se fait par rapport à une exécution dans laquelle il n'y a qu'un seul gestionnaire mémoire, implémenté de manière à supporter son exécution seule. Les applications mesurées sont les mêmes que dans le chapitre précédent.

La Table 4.2 expose le surcoût de l'utilisation de l'algorithme distibué par rapport à l'algorithme de référence. Pour chaque ramasse-miettes sont donnés la moyenne du surcoût observé ainsi que les surcoûts maximum et minimum. Ces chiffres ont été obtenus sur un ordinateur personnel dont les caractéristiques sont les suivantes : processeur Intel Pentium 4 à 3 GHz, 2 Mo de cache, tournant sous Linux.

La Table 4.3 expose les résultats dans la même forme mais pour le temps total passé à faire des collectes de données.

Les mêmes expériences ont été réalisées sur du matériel correspondant aux cibles

	Copiant	Compactant	Compactant (variante)	Marque et nettoie
Moyenne	+3.8%	+5.7%	+1.7%	+5.6%
Max	+4.9%	+5.4%	+3.1%	+9.8%
Min	+2.7%	+4.6%	+0.7%	+3.5%

TABLE 4.3 – Surcoût total de l'architecture concernant le temps de collecte

Surcoût	Copiant	Compactant	Compactant (variante)	Marque et nettoie
marquage	+8.7%	+9.3%	+9.6%	+8.1%
	(max. 9.1)	(max. 9.8)	(max. 10.2)	(max. 11.2)
collecte	+4.8%	+3.3%	+2.4%	+3.9%
	(max. 5)	(max. 4.1)	(max. 2.8)	(max. 4.1)

TABLE 4.4 – Impact de l'architecture sur le marquage et la collecte mesuré sur processeur ARM 9

visées : processeur ARM 9 200 MHz, 8 Ko cache. Les résultats sont donnés dans la table 4.4 et confirment les résultats obtenus à partir des expériences précédentes, ce qui montre que ces résultats illustrent bien les caractéristiques algorithmiques de l'architecture et non celles du matériel sous-jacent.

Globalement, les expériences ont donc montré que l'architecture présentée a un impact sur les performances. Cet impact est d'environ de +5% (maximum 9.8% pour une application spécifique) pour le temps de collecte total. Ce chiffre est non négligeable mais reste faible au regard du temps passé par un système à collecter la mémoire. En effet, le temps passé à collecter les données passe en dessous des 15% pour des besoins mémoire et/ou un choix de ramasse-miettes raisonnables [Appe 87, Tard 00, Blac 02]. Dans ce cas, un accroissement de 5% du temps passé à collecter les données signifie un ralentissement total de moins de 1%, qui est le prix payé pour une architecture portable et flexible. La flexibilité induite permet en outre de gagner en performance en adaptant la gestion de la mémoire aux applications, au matériel, et au partitionnement logiciel. Cependant, ce gain reste à évaluer.

Bien que le coût de l'architecture ait été mesuré, les résultats présentés ici ne comportent malheureusement aucune évaluation concernant une autre possibilité de l'architecture : la modification dynamique.

4.4 Modification dynamique de la gestion mémoire

On s'intéresse dans cette section à la dynamicité de la gestion mémoire. Tel que ce thème est abordé ici, il s'agit plutôt d'une fonctionnalité — décrite dans [Soma 04] — supportée par l'architecture proposée plutôt qu'un nouveau mécanisme.

4.4.1 Mécanisme

Le chapitre précédent, ainsi que des travaux existants [Atta 01, Fitz 00, Zorn 90, Smit 98], ont montré que l'efficacité d'un système de gestion de la mémoire (allocateurs et ramasse-miettes) dépend de l'application et des ressources matérielles. Ainsi, aucun de ces systèmes n'est le meilleur pour toutes les applications et tous les types de ressources. Soman et *al.* [Soma 04] ont montré qu'il est possible de s'affranchir de cette limitation en changeant dynamiquement de système de gestion, afin de l'accorder, au cours de l'exécution de la machine virtuelle, aux besoins des applications. L'architecture proposée permet la modification dynamique du type de gestionnaire mémoire choisi pour une partition donnée. En effet, les gestionnaires mémoire étant pilotés par le *Collecteur* à travers une interface, il suffit de changer l'implémentation, pour une partition donnée,

qui est faite des fonctions respectant cette interface pour que la gestion de la partition change. Et justement, l'implémentation du modèle proposé dans ce chapitre utilise un certain nombre de pointeurs de fonctions qui décrivent complètement l'algorithme de collecte utilisé. Il suffit donc, à l'exécution, de changer ces pointeurs de fonctions pour qu'ils pointent vers les fonctions d'un autre type de gestionnaire mémoire pour changer dynamiquement la gestion d'une partition. Cette opération n'est donc pas coûteuse en temps. Dans cette éventualité, la structure dans laquelle sont stockés les pointeurs de fonctions pour une partition devrait être placée en mémoire réinscriptible.

Cependant, l'organisation des données dans les partitions n'est pas faite de la même manière pour les différents types de gestionnaires mémoire. Une solution à ce problème pourrait être que chaque gestionnaire fournisse une fonction de transition dont le rôle serait de mettre la mémoire dans un état standard (par exemple, placer tous les objets du tas consécutivement). Un gestionnaire mémoire remplaçant un autre initialiserait alors sa mémoire à partir de cet état. Cependant, cela reste pour le moment du domaine des perspectives.

4.5 Conclusion

Dans ce chapitre, j'ai présenté une nouvelle architecture de gestion de mémoire orientée objet. L'objectif principal de cette architecture était de proposer une solution flexible et portable répondant à quatre contraintes :

- la gestion de la mémoire doit être automatique ;
- il existe un partitionnement physique dans les systèmes embarqués ;
- chacune des partitions peut avoir des propriétés très différentes des autres et nécessite donc une gestion adaptée ;
- le partitionnement logiciel est souvent utile dans les mémoires à objets.

La solution proposée [Marq 07d] est basée sur un principe de base simple : chaque partition est gérée par un gestionnaire dédié à ses caractéristiques ; l'aspect placement de données est transversal et réservé à un seul composant dédié. Les interactions entre les différents gestionnaires sont permises grâce à la définition que l'on a faite des services devant être fournis par chacun. Le canevas de gestionnaires obtenu fait fonctionner chacun de ces gestionnaires en coopération avec les autres.

Cette architecture a pu être validée expérimentalement. Elle a notamment été portée sur plusieurs cibles dont une plateforme d'expérimentation Excalibur et les capteurs MICAz. La facilité de portage du gestionnaire mémoire est encourageante. Les performances affichées sont convenables malgré le manque d'optimisations du code.

Cinquième Chapitre

Le placement des données en mémoire : une solution orientée langage

« Il est dit que les programmeurs Lisp savent que la gestion de la mémoire est si importante qu'elle ne peut être laissée aux programmeurs, et que les programmeurs C savent que la gestion de la mémoire est si importante qu'elle ne peut être laissée au système. »

Bjarne Stroustrup,
Concepteur du C++.

« C'est le langage qui noue les choses. »

A. de Saint-Exupéry,
Pilote de guerre.

Ce chapitre a pour objet la présentation d'une solution nouvelle au problème du placement des données qui se pose lors de la gestion de différentes mémoires. Cette solution est basée sur l'utilisation d'un langage dédié au placement des données. La proposition sera tout d'abord présentée dans son ensemble, puis le langage et son processus de compilation seront détaillés. Le tout repose sur une proposition : isoler l'aspect de placement des données.

5.1 Proposition : isoler l'aspect de placement des données

La proposition sera présentée dans son ensemble, après avoir détaillé, pour les systèmes embarqués, le problème du placement des données.

5.1.1 Le problème du placement des données

Comme on l'a vu précédemment, la gestion mémoire des systèmes embarqués est souvent particulière car ils ne fonctionnent pas avec une seule mémoire de travail mais plusieurs mémoires de différents types. De plus, un grand nombre de travaux de recherche ont montré qu'il peut être intéressant de partitionner une mémoire à objets. Un système gérant une mémoire à objets en cours d'exécution sur ce type de matériel va donc devoir à certains moments faire un choix dans l'allocation de ses données : dans quel espace mémoire placer telle ou telle donnée ? Ce choix est crucial pour deux raisons.

La première est que les mémoires ont des propriétés physiques très différentes. On ne peut donc pas mettre n'importe quelle donnée n'importe où. Des exemples simples illustrent cela. En premier exemple, prenons le cas d'une donnée souvent modifiée (écrite). Il ne sera pas efficace de la placer dans une mémoire lente. En deuxième exemple, prenons le cas d'une donnée devant rester secrète (une clef de cryptographie par exemple). Il n'est pas question pour des raisons de sécurité de la placer en mémoire persistante. Prenons un troisième exemple, dans lequel une mémoire a une durée de vie limitée en fonction du nombre de réécritures faites dans celle-ci. Dans ce cas, il convient de ne pas l'utiliser comme mémoire de travail.

La deuxième raison est que les besoins en mémoire de chaque application peuvent être très différents. Chaque application ne va pas accéder aux mêmes données, et pas de la même façon. Il convient donc de pouvoir adapter le placement des données en fonction des besoins applicatifs.

Ensuite, la configuration mémoire peut être très différente d'un appareil à l'autre. Si l'on souhaite garder le système de gestion de la mémoire portable, cela interdit l'utilisation d'un schéma de placement fixe. De plus, on souhaite au maximum réutiliser le logiciel et conserver une certaine portabilité des applications d'un appareil à l'autre. Ces contraintes nécessitent de pouvoir modifier facilement les choix faits par le système au niveau de la gestion mémoire d'un appareil à l'autre. Jusqu'à aujourd'hui, ce problème n'a pas été adressé car l'aspect placement de données est au cœur de la fonction du gestionnaire mémoire. Cela introduit de gros problèmes de portabilité et de réutilisation puisque le code responsable du choix de placement ne peut être pertinent pour deux configurations mémoire différentes (ne serait-ce que si la taille des mémoires présentes n'est pas la même). Les difficultés dans la mise au point de solutions génériques ont mené

jusqu'à aujourd'hui au développement de solutions dédiées à une plateforme et une configuration matérielle spécifique. Cela m'a amené à considérer d'un œil favorable l'isolation des aspects grâce aux avantages des langages dédiés.

5.1.2 Avantages des langages dédiés

L'approche proposée est d'isoler complètement l'aspect de placement des données du reste du gestionnaire mémoire, et plus généralement des autres fonctions du système. À cette fin, je propose d'utiliser un langage dédié (*DSL — Domain Specific Language*). Ce langage devra permettre d'exprimer complètement le placement des données en fonction à la fois du type de matériel présent, et des applications s'exécutant sur le système embarqué.

D'une façon générale, l'utilisation d'un langage dédié a de gros avantages, principalement en termes de :

Séparation des préoccupations C'est le premier avantage d'un langage dédié. Cette séparation des préoccupations permet de mettre au point des applications sans que leur conception soit dictée par un aspect trop particulier ;

Facilité de programmation Un langage dédié permet de manipuler directement les abstractions relatives au domaine concerné par le langage. En conséquence, les notations utilisées sont plus concises et plus lisibles qu'un code équivalent écrit en langage générique tel que le C. Le temps de développement sera ainsi diminué ;

Réutilisation Comme le code du langage dédié ne dépend pas du langage ou des structures internes du reste du système, la réutilisation est grandement facilitée. De plus, un code écrit en langage dédié va implicitement capturer une expertise du domaine, qui doit être faite explicitement par l'utilisateur d'un langage générique ;

Vérification de propriétés Un des avantages les plus importants des langages dédiés est qu'ils autorisent généralement la vérification de propriétés avancées. En effet, la sémantique des opérations décrites par le langage dédié peut rendre décidable des propriétés qui ne le seraient pas dans un langage générique.

Plus spécifiquement, l'usage de langages dédiés a montré ses preuves dans des domaines clefs des systèmes d'exploitation. Ainsi, [Meri 00] décrit une approche orientée langage pour la programmation de pilotes. De la même manière, Bossa [Lawa 02] est un langage dédié à l'écriture des composants-clefs que sont les ordonnanceurs de processus. Dans tous les cas le principe général d'utilisation d'un langage dédié est le suivant : le code écrit dans le langage dédié va être lu et compilé par un compilateur qui effectuera toutes les vérifications que le langage et le domaine concerné permettront de produire du code natif qui pourra être soit lié avec le code restant soit injecté dans celui-ci, ainsi que le montre le schéma 5.1.

Concernant le problème du placement évoqué ci-avant, l'avantage sur un langage générique est accrû puisque les aspects bas-niveau nécessitent le plus souvent l'écriture de code difficile à comprendre et à maintenir. Dans notre cas, cet avantage est clairement d'importance, car il peut être très difficile d'écrire du code bas-niveau (langage C par exemple) manipulant :

Les instances de classes Les données placées sont, dans le cas des machines virtuelles,

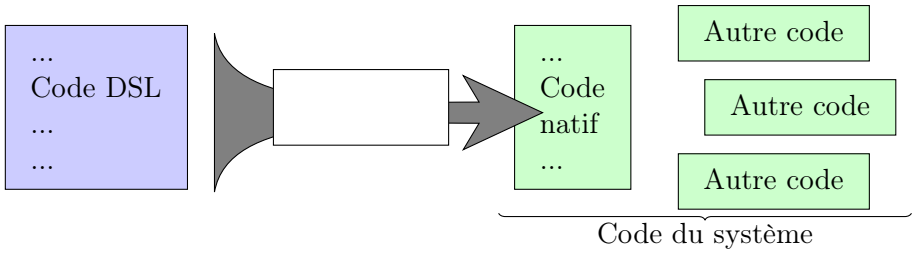


FIGURE 5.1 – Principe d'utilisation d'un langage dédié

des objets devant donc être examinés et manipulés dans leur représentation bas-niveau ;

Les composants du système En effet le code écrit est en partie dépendant de l'implémentation du gestionnaire mémoire ou encore des méthodes natives ;

Les mémoires physiques En fonction des mémoires, l'allocation de données peut être plus ou moins pénible à réaliser.

Les qualités des langages dédiés ainsi que ces difficultés nous amènent à détailler dans la suite une proposition orientée langage.

5.1.3 Proposition orientée langage

Nous allons maintenant voir le fonctionnement général d'une solution mettant en œuvre l'usage d'un langage dédié pour le placement des données. Le cœur de cette solution est l'emploi d'une *politique de placement* écrite dans le langage dédié. Cette politique de placement décrit complètement le placement des objets tout au long de l'exécution du système. Cette politique est compilée et le code produit est injecté dans le gestionnaire mémoire afin de lui permettre de gérer le placement d'une façon adéquate au vu des applications et des mémoires disponibles. Le schéma 5.2 illustre le fonctionnement général de la solution proposée dans l'architecture de gestionnaire mémoire présentée précédemment. Cependant, il est important de noter que ce schéma de fonctionnement n'est pas spécifique à cette architecture. En effet le code produit sert uniquement à faire le choix du placement des données, et pourrait donc être injecté dans un autre système, pour peu que le générateur de code du compilateur soit modifié de façon à ce que le code produit soit compatible.

La fonction de base de notre langage dédié va être d'identifier certains ensembles de données (objets dans notre contexte) dans le but de pouvoir spécifier leur placement. Afin de pouvoir les identifier de façon assez fine, un certain nombre de critères peuvent être utilisés. À l'exécution, le code compilé à partir de ces spécifications va évaluer ces critères et placer les données de la façon indiquée par les spécifications données. Les critères que nous avons identifiés sont présentés ci-après. On distinguera les critères avancés des critères de base.

Critères de base

Parmi les critères nécessaires à l'identification de certaines catégories d'objets, on trouve des critères simples relatifs aux objets eux-mêmes :

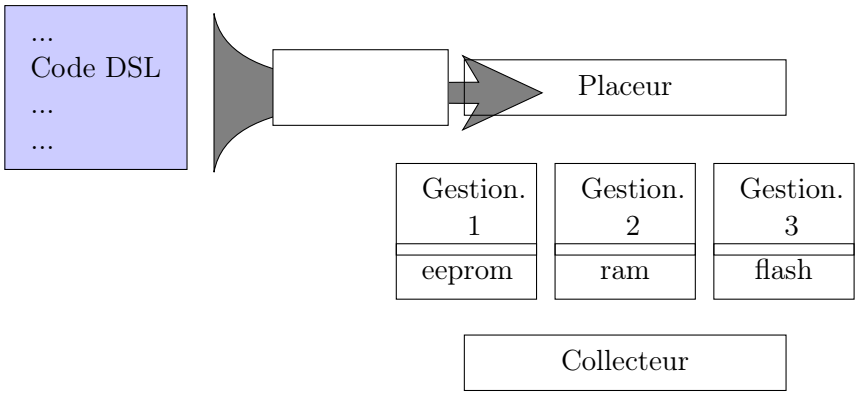


FIGURE 5.2 – Utilisation d'une politique de placement dans l'architecture proposée

Le type est un premier bon indicateur de la fonction d'un objet. Par exemple un objet de type `String` n'étant jamais modifié après son allocation, il peut être placé dans une mémoire lente en écriture.

La taille d'un objet est également un critère significatif. Par exemple, on peut choisir de ne pas remplir d'un coup une mémoire efficace mais présente en faible quantité.

En plus des critères en relation directe avec les objets eux-mêmes, on distingue les critères dépendant de l'état du système :

Le site d'allocation est typiquement identifié grâce à un nom de classe et un nom de méthode. C'est une information précieuse permettant une identification très précise d'un objet. Par exemple, des objets alloués dans une méthode donnée peuvent être très peu écrits : typiquement, des instances de la classe `Integer` créées à seule fin de servir de clefs dans une table de hachage correspondent à un tel cas d'utilisation ; elles peuvent donc être placées en mémoire peu efficace en écriture ;

L'espace disponible dans une ou plusieurs mémoires peut aider à la décision, si on ne souhaite pas remplir une mémoire efficace avec des objets trop volumineux par exemple ;

La location actuelle de l'objet peut également aider à le placer précisément par la suite. Par exemple, dans le modèle générationnel, ce sont les objets de la nurserie qui sont déplacés ;

L'application allouant un objet est un critère intéressant permettant d'accélérer une application en mettant tous les objets qu'elle alloue dans une région donnée. La distinction avec le site d'allocation est nécessaire puisque deux applications différentes peuvent utiliser les mêmes APIs.

Cependant ces propriétés simples ne suffisent pas à identifier précisément tous les objets ; sont donc présentés à présent les critères avancés.

Critères avancés

Le contenu d'un objet (références et valeurs) peut être révélateur de certaines propriétés. Les valeurs, parce qu'elles témoignent de l'état d'un objet ; les références parce qu'elles traduisent les relations d'un objet avec les autres, ce qui est aussi une information précieuse. Par exemple, un tableau de caractères peut être rempli et modifié au fur

et à mesure de l'exécution d'une application mais peut aussi être associé à une instance de type `String`, auquel cas ses valeurs initiales ne seront jamais modifiées. Les critères permettant d'identifier un objet en fonction de ses références vers d'autres objets et de ses valeurs sont listés ci-après :

Les valeurs des champs d'un objet sont également des propriétés d'un objet car ils permettent de l'identifier précisément. Par exemple, le `Thread` de nom « Serveur » doit être placé dans une mémoire efficace.

Référents Le critère de lien entre deux objets peut s'exprimer sous la forme d'objets qui référencent un objet donné ;

Référencés Au contraire, on peut également identifier un objet selon les objets que lui-même référence.

Les critères de base et critères avancés tirent partie de la définition même du modèle de mémoire à objets. Cependant, ils ne suffisent pas à gérer le placement de la manière la plus complète car un certain nombre de travaux ont défini des propriétés annexes.

Propriétés annexes

Un certain nombre de propriétés peuvent être définies, sur la base d'une analyse statique par exemple. Au nombre de ces propriétés, on peut citer :

- Appartenance à un contexte donné. L'analyse d'échappement [Blan 98, Choi 99] permet de définir les objets n'échappant pas à un contexte donné. À l'exécution, de tels objets sont placés dans une zone spécifique. D'une manière plus générale, un grand nombre de travaux concernent la gestion de la mémoire en régions pour améliorer les performances, soit de façon explicite dans le langage [Gay 98, Gay 01], soit automatisée [Cher 04, Chin 04], ou encore afin de conserver des propriétés temps-réel [Dete 02] ;
- L'âge d'un objet. C'est la caractéristique principale exploitée par les ramasse-miettes générationnels ;
- Connectivité spécifique. Afin de ne collecter que partiellement la mémoire, les objets peuvent être discriminés selon leur connectivité [Hirz 03].

Bien sûr, l'utilisation de telles propriétés demande un support à la fois du système embarqué, mais aussi du compilateur, en plus de l'utilisation éventuelle d'un outil d'analyse. Cependant, elles apportent une réelle profondeur à une politique de placement, et améliorent l'utilité des critères.

Utilité des critères

L'ensemble des critères de placement présentés sont intéressants car ils permettent d'identifier finement des catégories d'objets. Cependant ils ne sont pas tous évaluables à n'importe quel moment.

Premièrement, le placement des objets a besoin d'être choisi lorsque très peu d'informations sont disponibles les concernant. Typiquement, la gestion du placement à l'allocation est intéressante puisqu'elle permet de faire un premier choix sur le placement très tôt plutôt que de tout allouer dans une même zone mémoire (qui risquerait d'être pleine très vite ou de ralentir le système si un objet souvent accédé en écriture est alloué en

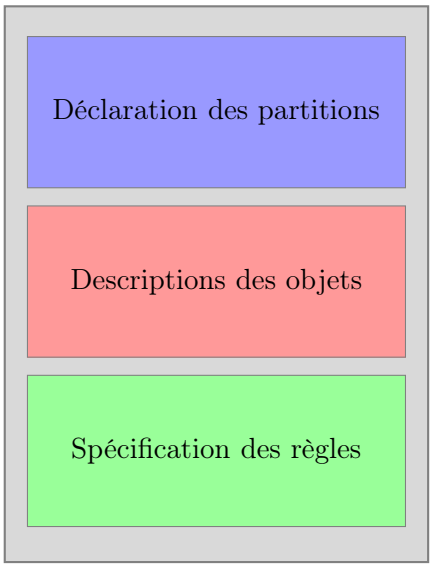


FIGURE 5.3 – Composition d'une politique de placement

EEPROM par exemple). Cependant, à cet instant de la vie d'un objet, peu d'informations sont connues concernant ce nouvel objet. En fait, l'ensemble des critères que l'on va pouvoir évaluer à ce moment est constitué des critères de base décrits en section 5.1.3.

De plus, évaluer les autres critères va demander un certain temps car ils impliquent de scanner la mémoire complètement en examinant notamment les références entre objets. Cela peut être fait lorsque la rapidité d'exécution n'est pas primordiale (extinction de l'appareil par exemple) et/ou à un moment où le contenu des objets, en particulier les références entre objets, est facilement évaluable. La phase de marquage préalable à toute collecte est l'un de ces moments. On va donc gérer le placement des objets également pendant les phases de collecte, ce qui permettra d'ajuster le placement qui ne serait fait qu'à gros grains si l'on se contentait de l'allocation dans l'écriture d'une politique de placement.

5.2 Écriture d'une politique de placement

L'objet de cette section est de détailler la forme des politiques de placement. Dans la suite de ce chapitre, un certain nombre d'exemples sont donnés afin d'illustrer les concepts présentés. Ces exemples sont souvent en rapport avec l'implémentation réalisée, qui est utilisée au sein d'un système Java. Les modifications aux solutions proposées nécessaires pour supporter d'autres types de systèmes sont évoqués en section 5.6.

La forme d'une politique de placement est donnée par la figure 5.3). Elle exprime trois éléments :

- Expression des différentes zones mémoire ;
- Identification de certains objets en fonction de critères de placement ;
- Règles de placement : association d'un type d'objet avec une zone mémoire.

Pour toute précision relative au langage, on pourra se référer à sa grammaire, donnée par la figure 5.4. En premier lieu est détaillée la déclaration de partitions.

policy	: :=	partitions-list descriptions-list allocation-rules collection-rules
partitions-list	: :=	(desc-partition)+
desc-partition	: :=	**Partition** partition-identifier **{** (parameters-partition)+ **}**
parameters-partition	: :=	partition-size partition-memory
partition-size	: :=	**Size is** (**digit**)+ (Bytes | KBytes | MBytes) **;**
partition-memory	: :=	**Memory is** memory-identifier **;**
property	: :=	**Property :** identifier
descriptions-list	: :=	(object-description)*
object-description	: :=	[**KindOf**] type description-identifier [criteria] [+ description-identifier] **;**
criteria	: :=	**{** (criterion)+ **}**
criterion	: :=	(object-criterion | space-size) **;**
space-size	: :=	**PartitionSize** partition-identifier comparaison-op number
object-criterion	: :=	size | site | location | application | referent | value | property
size	: :=	**Size** comparaison-op number
site	: :=	**Site is** type **.** ((method-identifier **(** (type)* **)** [type]) | *****)
location	: :=	**Location is** partition-identifier
application	: :=	**Application is** type
referent	: :=	**ReferredBy** type description-identifier [criteria-referent]
criteria-referent	: :=	**{** (criterion-referent)+ **}**
criterion-referent	: :=	(object-criterion | fieldname | index) **;**
fieldname	: :=	**Field is** string
index	: :=	**Index is** number
value	: :=	field-identifier (**is** | comparaison-op) (string | number | field-identifier)
field-identifier	: :=	description-identifier **.** field-name
allocation-rules	: :=	**Allocation {** (rule)* memory-identifier **}**
collection-rules	: :=	**Collection {** (rule)* **}**
rule	: :=	description => partition-identifier **;**
number	: :=	(**digit**)+ | identifier **.size**
type	: :=	identifier (**.** identifier)*
string	: :=	**quote** (**any character**)* **quote**
comparaison-op	: :=	**is** | < | > | ! = | <= | >=
identifier	: :=	**letter** (**letter** | **digit** | **underline**)*
method-identifier	: :=	partition-identifier : := description-identifier
	: :=	field-name : := identifier

FIGURE 5.4 – Grammaire du langage dédié

5.2.1 Déclaration de partitions

Le langage dédié mis au point permet de manipuler des *partitions*, afin de pouvoir gérer le partitionnement à la fois matériel et logiciel des systèmes embarqués. Une partition désigne ici un espace mémoire logique, par opposition aux mémoires *physiques*. Plusieurs partitions logiques peuvent être créées par mémoire physique. Cette capacité peut être utile afin de créer plusieurs tas ayant certaines propriétés. Par exemple, un tas ne contenant que des objets sans références peut être utile au côté d'un autre dans une même RAM.

La création de partitions est permise très simplement dans le langage dédié, en utilisant le type `Partition` de la façon décrite par la figure 5.5. Un nom de variable doit être spécifié, suivi de la description de la partition après une accolade ouvrante. La déclaration de la partition est terminée lorsque une accolade fermante est atteinte. Au sein de la description, on trouve la spécification de la taille de la partition et le nom de la mémoire physique.

De plus, on trouve également le type de ramasse-miettes en charge de cette partition. Cette indication est purement liée à l'architecture proposée dans le chapitre précédent puisqu'elle concerne l'attribution d'un gestionnaire spécifique à chaque zone mémoire.

Enfin, il est possible de spécifier certaines propriétés sur la partition. Par exemple, la propriété de non-réécriture de cette partition (cette partition n'est pas accessible en écriture).

```
Partition nursery {
    Size is 5 KBytes;
    Memory is ewram;
    Collector is compacting;
};
```

FIGURE *5.5 – Définition d'une partition*

Les noms identifiant d'une part les mémoires physiques présentes sur une cible donnée, et d'autre part les ramasse-miettes disponibles sont donnés par deux fichiers de configuration ; le premier, décrivant les mémoires physiques, est bien sûr spécifique à chaque architecture.

5.2.2 Critères de placement

Afin de spécifier leur placement, les objets sont tout d'abord décrits grâce à une série de critères de placement. La description d'un ensemble d'objets est faite de la manière suivante.

Tout d'abord le type de l'objet doit être indiqué. Le fait de spécifier ce critère en premier lieu est justifée pour deux raisons :

1. Ce critère est le plus important, il permet d'identifier un ensemble d'objets très simplement ;
2. Le mettre en premier lieu facilite la lecture. En effet cela concorde avec la déclaration de type d'un objet.

Mot clef	Description
KindOf	En début de description, indique que le type donné n'est pas forcément exact
PartitionSize	Indique une condition sur l'espace libre restant
Size	Spécifie une condition sur la taille de l'objet
Site is	Décrit le site d'allocation de l'objet
Location is	Décrit la partition dans laquelle doit être l'objet
Application is	Spécifie l'application utilisant l'objet par son type
Referred By	Décrit une condition sur les références que possède un objet
Field is	Permet de spécifier le nom du champ par lequel un objet est référencé
Index is	Permet de spécifier l'index dans un tableau auquel est rattaché un certain objet
Property is	Indique que l'objet doit avoir une certaine propriété pour être concerné

TABLE 5.1 – Syntaxe des différents critères utilisables dans le langage dédié

Ensuite, le nom de cette description doit être spécifié et la description doit soit être interrompue au moyen d'un ';' soit être suivie du corps de la description. La figure 5.6 illustre les différents moyens de déclarer une description d'objet. On notera la notation + permettant de grouper plusieurs descriptions pour en obtenir une autre.

```
Type firstDescription;

Type secondDescription {
    ...
};

Type lastDescription {
    ...
} + firstDescription;
```

FIGURE *5.6 – Déclaration d'une description*

Le corps d'une description d'objets est composé d'une simple liste de critères. Ces critères sont bien sûr ceux détaillés dans la Section 5.1.3. Le tableau 5.1 centralise toutes les notations correspondant à ces critères. On remarquera l'utilisation du mot-clef *Property* suivi par une chaîne de caractères identifiant une propriété spécifique dont le support doit exister dans le compilateur.

La figure 5.2.2 montre quatre exemples de descriptions d'objets. La première illustre certains critères de base. Les deux suivantes illustrent respectivement un critère de valeur de champ et un critère de référence. La dernière illustre l'utilisation de la propriété de non-échappement.

Les descriptions peuvent ensuite, à travers leur nom, être utilisées dans les règles de placement.

5.2.3 Règles de placement

Cette dernière partie d'une politique de placement fait le lien entre partitions et descriptions en indiquant où doit être placé chaque objet répondant aux critères spécifiés par une description. Ces indications sont données sous la forme d'une liste d'items de la forme indiquée par la figure 5.8. Il existe une liste de règles par moment, dans la vie du système, où le placement doit être géré. Le membre gauche de chaque règle est une

```
myserver.Client client {
   Site is myserver.HTTPServer.handleConnection() ;
} ;
```

(a) Critères de base

```
java.lang.Thread serverThread {
     serverThread.name is "HttpServer" ;
} ;
```

(b) Critère de valeur de champs

```
java.lang.Class[] classloaderArrays {
     ReferredBy java.lang.Classloader cl {
          Field is cl.classes ;
     } ;
} ;
```

(c) Critère de référence

```
Object nonescaped {
     Property is "EANew" ;      } ;
} ;
```

(d) Critère de non-échappement

FIGURE 5.7 – Exemples de descriptions d'objets

```
placement_instant {
     desc_objet1 => partition [,partition*] ;
     desc_objet2 => partition [,partition*] ;
     ...
} ;
```

FIGURE *5.8 – Règles de placement*

description d'objets ; en membre droit on trouve une suite de noms de partitions indiquant l'ordre de priorité dans laquelle les partitions doivent être évaluées pour le placement d'un objet respectant les critères de la description concernée. Donc si la première partition indiquée est pleine au moment de réaliser le placement, la deuxième est envisagée et ainsi de suite.

Dans l'implémentation que j'ai réalisée, seuls deux *placement_instant* sont possibles puisque comme mentionné précédemment, le placement est géré uniquement à l'allocation et pendant les collectes. On trouve donc une section *Allocation {....}* et une autre *Collection {...}*. Toutes les variables utilisées dans une règle — donc faisant référence soit à une partition soit à une description — doivent avoir été déclarées préalablement.

Deux objets pouvant correspondre à deux descriptions différentes, ce fonctionnement par liste permet d'exprimer très simplement les priorités entre les règles. En effet pour chaque objet examiné, c'est la première règle dont la description correspond à cet objet qui s'applique. Ainsi, lorsqu'un objet est alloué, les règles de la section « Allocation » sont examinées une par une. Si les critères de la description d'une règle correspondent à l'objet, celui-ci est alloué dans la partition indiquée par la règle.

5.2.4 Exemple

L'exemple présenté ici considère un petit appareil à l'architecture standard, équipé d'un microprocesseur ARM7, incluant 512 Ko de ROM, 256 Ko d'EEPROM, 256 Ko d'EWRAM (External Writable RAM) et 32 Ko d'IWRAM (Internal Writable RAM — située sur le chip lui-même). La figure 5.9 montre une petite politique qui indique le placement suivant :

- Les classes et les objets représentant une chaîne de caractères sont alloués en EEPROM ;
- Les tableaux de caractères sont alloués en EEPROM car ils sont généralement peu modifiés ;
- Les autres objets sont alloués par défaut en IWRAM, qui sert alors de mémoire de travail principale. Pendant une collecte de données, ces objets sont déplacés en EWRAM puis en EEPROM s'ils survivent à une collecte supplémentaire, avec deux exceptions ;
- Les piles d'exécution restent en IWRAM ;
- Les objets manipulés par le thread dont le nom est « HttpServer » restent en IWRAM également de façon à augmenter la vitesse d'exécution de l'application par laquelle ils sont utilisés.

```
import java.lang.*;

Class class;

String string;

char[] charArray;

Stack stack;

Object serverObject {
    Site HttpServer.*;
};
char[] stringArray {
    Referred by String;
};
Object inIwram {
    Location is part_iwram;
};
Object inEwram {
    Location is part_ewram;
};

Allocation {
    stack => part_iwram, part_ewram;
    serverObject => part_iwram, part_ewram;
    charArray => part_iwram, part_ewram;
    class => part_eeprom, part_ewram, part_iwram;
    string => part_eeprom, part_ewram, part_iwram;
    part_iwram;
};

Collection {
    stringArrays => part_eeprom, part_ewram;
    serverObject => part_iwram, part_ewram;
    stacks => part_iwram;
    inIwram => part_ewram;
    inEwram => part_eeprom;
};
```

FIGURE 5.9 – Un exemple de politique de placement très simple

Une fois mise au point, une telle politique de placement est transformée au travers du processus de compilation.

5.3 Processus de compilation

À partir d'une politique de placement, le compilateur du langage dédié va produire le code natif correspondant. Ce code est intégré au sein du gestionnaire mémoire. Le compilateur de politique de placement, responsable de cette transition, est détaillé ci-après. L'architecture générale du compilateur est décrite, et deux parties de ce compilateur sont détaillées : les vérifications effectuées par ce compilateur et les optimisations faites avant de produire le code natif. Mais tout d'abord, afin que le lecteur ait une bonne compréhension de ce que produit le compilateur, il trouvera en premier lieu l'explication du fonctionnement du code généré.

5.3.1 Fonctionnement du code généré

Le code exécuté pour choisir le placement d'une instance au moment de l'allocation est relativement simple puisqu'il s'agit d'une fonction évaluant des critères simples concernant le nouvel objet. Cependant, l'une des propriétés que l'on souhaite conserver est la capacité de déplacer les objets d'une partition à une autre tout au long de l'exécution du système. Cela signifie qu'il faut évaluer potentiellement toutes les propriétés d'un objet au moment où son placement est à nouveau évalué. Cela peut être extrêmement coûteux. On va donc considérer le fonctionnement général suivant :

- Quand un objet est alloué, un numéro d'*état* lui est attribué. Ce numéro d'état traduit toutes les propriétés simples que l'objet possède à sa création (exemple : tous les objets de type `String` reçoivent l'état 5).
- Ce numéro d'état est mis à jour tout au long de sa vie aux moments où le placement est évalué.

Ce fonctionnement permet tout d'abord de connaître directement les propriétés simples sans devoir les évaluer une seconde fois. Ensuite, cela permet de conserver l'information concernant le site d'allocation qui ne peut être retrouvée après l'allocation. Lorsque le placement sera évalué pendant une collecte par exemple, si tous les objets de type `String` alloués par la méthode m de la classe C doivent être placés dans une certaine partition, il suffit de tester l'état de l'objet. En bref, chaque objet possède un entier qui code son état au vu de la politique de placement. Un objet n'ayant aucune signification au vu de la politique aura un état 0.

Comme on l'a vu précédemment, la phase de marquage présente l'avantage de parcourir toutes les références des objets ; on souhaite donc profiter de cette phase pour évaluer les critères avancés tels que ceux prenant en compte les références entre instances. L'algorithme de marquage a donc été modifié de façon à permettre l'évaluation de ces critères avancés. Le principe, illustré par la figure 5.10, est donc le suivant.

Chaque objet (non marqué) atteint pour la première fois par un lien depuis un autre objet survivant à la collecte est scanné. Les valeurs qu'il contient sont examinées et évaluées afin d'indiquer si l'objet est concerné par une règle de la politique de placement. Cela est fait *si et seulement si* son état à ce moment rend nécessaire l'évaluation d'un critère de valeurs de champs au vu de la politique de placement. Exemple : si la seule description dans la politique contenant un critère de valeur de champ concerne uniquement les instances de `Thread`, alors seuls les objets ayant un état indiquant qu'ils sont de ce type seront scannés.

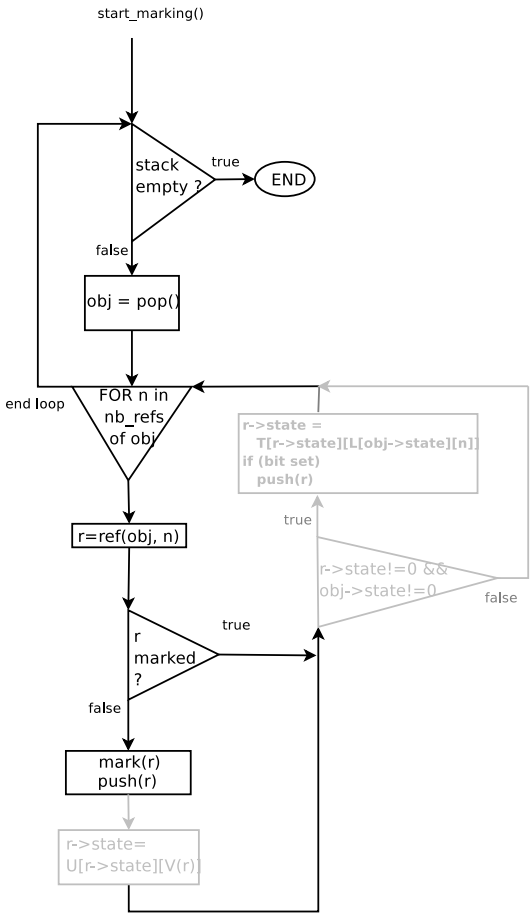


FIGURE 5.10 – Modification de l'algorithme de marquage

De plus, à chaque fois qu'une référence r est suivie d'un objet A vers un objet B, alors l'état de B doit changer si dans la politique de placement une description contient un critère de référence (`referredBy`) s'appliquant à A et vers un critère à B.

Durant la phase de marquage, l'état d'un objet est donc susceptible d'évoluer plusieurs fois, suivant les liens qu'il a avec d'autres objets. Le schéma 5.11 illustre cette évolution d'état : ce petit exemple concerne une description d'objets spécifiant un critère de référence et un critère de valeur de champ. Ce schéma représente l'évolution du numéro d'état d'un objet vérifiant tous les critères exprimés, sous forme d'un graphe dont les noeuds sont les états et les transitions les vérifications de critères. Sur ce schéma, l'état initial est l'état attribué à l'objet à son allocation, et l'état final est l'état codant directement son placement dans une partition donnée. On remarque que deux chemins sont possibles dans ce graphe car on ne peut savoir à l'avance si le critère de référence sera vérifié avant celui de valeur de champ pendant la phase de marquage. Au cours de l'évolution de l'état d'un objet, un seul changement d'état (une transition dans le graphe) sera dû à l'évaluation des critères de valeur puisque les valeurs ne sont évaluées qu'une seule fois.

De manière un peu plus formelle, notons $s_i^{O_x}$ l'$i$ème état atteint par un objet O_x et donc $s_0^{O_x}$ l'état donné par l'allocateur à celui-ci. La fonction utilisée pour attribuer

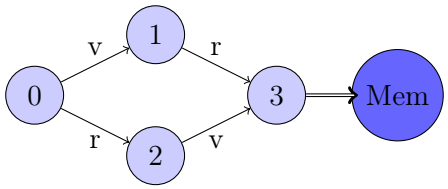


FIGURE 5.11 – Évolution de l'état d'un objet pendant la phase de marquage, et affectation de sa mémoire de destination en fonction de son état final

ce numéro d'état est notée A. Au cours du marquage, pour chaque lien l_j (= j-ième référence contenue dans l'objet O_r) d'un objet O_r vers O_x exprimé dans la politique de placement, le nouvel état de O_x dépend : de la nature du lien l_j, de $s_i^{O_r}$, et de $s_i^{O_x}$. Notons G la fonction qui attribue le nouvel état en fonction de ces trois paramètres. L'évolution de l'état de O_x s'exprime donc de la façon suivante :

$$s_0(O_x) = A(O_x) \tag{5.1}$$

$$M_{O_x} \Rightarrow\ s_i(O_x) = G(S_i^{O_r}, l_j^{O_r}, S_{i-1}^{O_x} \tag{5.2}$$

$$)\overline{M_{O_x}} \Rightarrow\ s_i(O_x) = G\prime(S_i^{O_r}, l_j^{O_r}, V(O_x)) \tag{5.3}$$

La notation M_{O_x} indique que l'objet O_x est marqué quand il est atteint via une référence ; la notation $\overline{M_{O_x}}$ indique le contraire.

Lorsqu'un objet atteint est déjà marqué, la fonction G est utilisée. On va diviser cette fonction en deux sous-fonctions T et L de telle sorte que l'on ait :

$$G(S_i^{O_r}, l_j^{O_r}, S_{i-1}^{O_x}) = T(s_{i-1}^{O_x}, L(s_{i-1}^{O_x}, l_j^{O_r})) \tag{5.4}$$

Cependant, il est possible de caractériser $l_j^{O_r}$ par un entier égal à j (le numéro de la référence dans O_r). T et L sont donc deux fonctions prenant en paramètres deux entiers et retournant un entier. En conséquence, on va pouvoir implémenter ces fonctions sous formes de tableaux à doubles entrées : L donnant un entier à partir d'un état et d'un numéro de lien, et T fournissant le nouvel état à partir de cet entier et d'un numéro d'état.

Lorsqu'un objet atteint n'est pas marqué (il est donc atteint pour la première fois), il est marqué, puis son état est mis à jour en fonction de ses valeurs de champs. À cette fin, une fonction V calcule un nombre caractérisant ces valeurs de champs au vu de la politique spécifiée). Puis, un tableau U donne le nouvel état de l'objet en fonction des deux indices suivants : l'ancien état de l'objet atteint $S_{i-1}^{O_x}$ et l'entier donné par V. Enfin, l'état est éventuellement mis à jour en fonction des critères de référence, en utilisant le même principe que celui expliqué ci-dessus.

Dans le cas où plusieurs critères de référence sont imbriqués dans une description d, si l'état d'un objet O est mis à jour, il est possible que les objets référencés par O doivent alors changer d'état également. On note R la fonction indiquant si O est dans un tel état, c'est-à-dire si la modification de son état va potentiellement mener à un changement d'état des objets qu'il référence. R est implémentable par un simple bit que l'on associe à chaque état. On code simplement cette information sur un bit dans les

entrées du tableau U. Quand un objet pour lequel ce bit est positionné est atteint, il est remis sur le sommet de la pile, de façon à ce que les références qu'il contient soient à nouveau suivies plus tard.

Le rôle du compilateur est donc de :
- Générer la fonction A qui calcule l'état initial d'un objet ;
- Générer la fonction V qui met à jour, si nécessaire, l'état d'un objet en fonction de la valeur de ses champs ;
- Générer 4 tableaux :
  - les deux tableaux correspondant à L et T,
  - le tableau P associant à un état final un numéro de partition,
  - le tableau U utilisé pour l'évaluation des critères de valeur de champ.

Une des qualités de ce mécanisme est qu'il profite grandement de l'algorithme de marquage standard. Les parties grisées de la figure 5.10 montrent les modifications apportées à cet algorithme. On peut voir qu'elles ne concernent que les objets survivant à la collecte *et* dont l'état indique qu'ils sont concernés par la politique de placement. Seuls deux accès dans des tables sont nécessaires pour mettre à jour l'état d'un objet concerné par un critère de référence. Si on considère que seulement une fraction des objets survivent à une collecte de données [1], ces opérations concernent en plus très peu de données.

La génération de ce code est établie au terme du processus de compilation que permet l'architecture du compilateur.

5.3.2 Architecture du compilateur

L'architecture du compilateur est illustrée par la figure 5.12. On y trouve en partie gauche les différentes partie du compilateur, et en partie droite la représentation d'une politique de placement à chaque étape. Une compilation met en jeu les phases suivantes :

1. Analyses lexicale et syntaxique. La grammaire du langage est décrite en utilisant JFlex[Klei 98]. Au terme de ces deux étapes, on obtient :
   - une liste de partitions ;
   - une liste de descriptions ;
   - un ensemble de règles pour chaque endroit où le placement est géré (donc deux pour l'instant).

   Au terme de cette phase, le compilateur obtient une forme intermédiaire de la politique de placement.
2. Construction de la représentation interne d'une politique de placement, sous forme de graphes.
3. Analyse sémantique. Durant cette étape, certaines vérifications sont effectuées. Cette partie est détaillée dans la section suivante.
4. Optimisations. Cette phase a pour but de réaliser des optimisations sur la forme intermédiaire d'une politique. Les optimisations réalisées par le compilateur sont détaillées en section 5.3.4.

1. (Stefanovic et Moss ont rapporté que seulement 2-8% survivent à 100KB d'allocation dans les applications 4SML/NJ [Stef 94]. Ungar a également rapporté qu'environ 7% des objets smalltalk survivent à 140KB d'allocation [Unga 87]. Dieckmann and Hölzle ont évalué à 1-27%, le nombre d'octets survivants des applications Java (pour 400KB d'allocation) [Diec 99]

5. Génération de code. La section 5.3.5 détaille la fonction du code généré et son fonctionnement à l'exécution du système. Au niveau architectural, cette phase est réalisée, pour chaque ensemble de règles de la politique, par un *terminal* qui transforme la représentation interne d'une politique en code spécifique. Le *terminal* principal dans notre implémentation est bien sûr celui générant du C compatible avec le gestionnaire mémoire en place dans le système Java. **C'est la seule partie, avec la création d'une politique adéquate, à réécrire dans le cas d'un portage, au niveau de la gestion mémoire.**

Le compilateur est écrit en Java et utilise principalement les classes suivantes pour représenter les diverses abstractions du langage, et donc la représentation interne d'une politique :

partitions Elles sont représentées tout simplement par des instances de la classe `Partition` et contiennent les informations nécessaires à toute description de partition. On peut noter l'utilisation d'un champ de type `PhysicalMemory` ; cette classe décrit les propriétés de toute mémoire physique. Les instances de cette classe sont chargées à partir de fichiers de configuration concernant le matériel de la cible ;

critères On utilise des instances de la super-classe `Criteria` dont les sous-classes représentent tous les critères pouvant être spécifiés dans le langage. Les critères sont regroupés au sein d'instances de la classe `ObjectDescription` ;

règles de placement Les instances de type *Rule* associent une instance de la classe *ObjectDescription* à une liste de `Partition`. Il existe en réalité une sous-classe pour chaque moment durant lequel le placement est géré ; elles sont nommées <mot-clef>-*Rule*. Le mot-clef étant celui utilisé dans le langage, on trouve donc dans notre implémentation les classes `AllocationRule` et les classes `CollectionRule` ;

noeuds du graphe Les instances de la classe *Node* sont chainées entre elles, constituant ainsi un graphe pour chaque règle. Deux type de noeuds existent : les noeuds finaux (représentant une partition) et les autres. Ces derniers possèdent une liste de *Criteria* permettant de connaître les critères associés à ce noeud ainsi qu'une liste de transition possibles à partir de ce noeud ;

transitions Les instances de `Transition` possèdent un `Criteria` qui est l'étiquette de la transition ainsi représentée.

Les analyses sémantique et lexicale, ainsi que la construction de la représentation interne sous forme de graphe étant relativement simples et peu originales par rapport à n'importe quel processus de compilation, elles ne sont pas détaillées dans ce document. On trouve maintenant le détail des vérifications du compilateur.

5.3.3 Vérifications du compilateur

Comme vu précédemment, la possibilité offerte par le langage dédié de vérifier les propriétés avancées du domaine visé constitue l'un des avantages d'utiliser un tel langage. Grâce à la nature déclarative du langage proposé, il est possible de vérifier les propriétés décrites ci-après, à commencer par les vérifications de type.

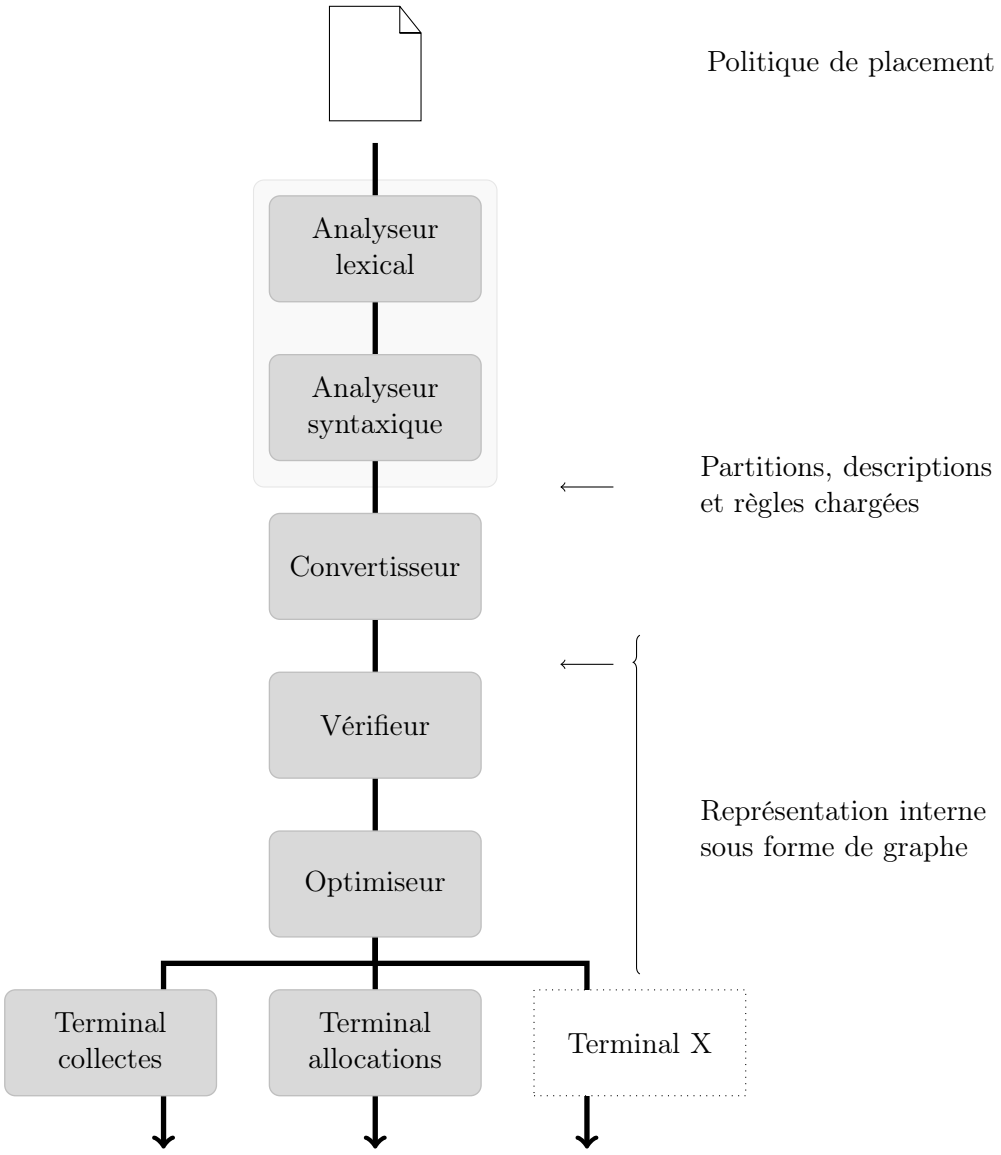


FIGURE 5.12 – Compilation d'une politique de placement : vue détaillée

Vérification de type

Les abstractions du langage (par exemple les partitions, les descriptions) sont typées. Chaque utilisation d'une variable doit être validée et doit correspondre au type de déclaration de cette variable.
N.B. : la lisibilité et la taille d'une politique sont améliorées grâce à la possibilité de réutilisation des descriptions. Le typage fort des variables permet notamment de vérifier cet usage.

Vérification des tailles spécifiées

La détection manuelle des écrasements mémoire peut être pénible à réaliser. Le compilateur vérifie donc que toutes les partitions rentrent bien dans la mémoire physique dans laquelle elles sont déclarées. Il est notamment possible de ne pas spécifier de taille pour une partition. Dans ce cas, la taille est fixée de manière à maximiser l'usage de la mémoire physique correspondante.

Cohérence des descriptions d'objets

Comme le type, la taille, les champs d'un objet et ses valeurs, ainsi que les références d'un objet sont manipulés directement en tant que tels par le langage, le compilateur est capable de vérifier que l'utilisation commune de tels critères est correcte. Il est donc possible de détecter l'usage erroné d'un nom de champ par rapport au type de l'objet (exemple : il n'existe pas de champ *name* dans la classe `Integer`). Ou encore, comme la taille des instances de certains types est connue (et fixe), le critère de taille exprimé peut être vérifié (exemple : spécifier un critère de taille supérieur à 512 octets pour un type `Integer` n'a pas de sens alors qu'il en a pour un tableau). De la même manière, les relations entre objets peuvent être déterminées possibles ou non (exemple : un objet de type `String` ne peut référencer un objet de type `Integer`). Le type d'un objet est vérifié selon l'existence ou non d'une classe de même nom (dans le classpath).

Utilité de tous les critères

Plus le nombre de critères vérifiés à l'exécution est élevé, plus l'exécution du système embarqué faisant ces vérifications est ralentie. Ainsi, dans une certaine mesure, le compilateur détecte les cas dans lesquels certains critères sont inutiles car n'apportent rien au niveau de l'identification d'un objet. Ainsi, spécifier que la taille d'un objet doit être supérieure à 1 octet n'apporte aucune information puisque toutes les instances ont une taille supérieure à 1 octet. De manière plus fine, il est inutile de spécifier à la fois le nom du champ *et* le type de l'objet référencé si un seul champ de ce type peut être référencé par un objet du type du référent.

Utilité de toutes les descriptions

Toutes les descriptions et partitions doivent être utilisées dans au moins une règle de placement. De plus, deux variables ne peuvent avoir le même contenu. Il est donc impos-

sible de déclarer deux descriptions identiques. Cela garantit que toutes les descriptions et partitions définies seront utiles au vu de leur utilisation dans des règles de placement.

Cohérence des priorités données entre les règles

Le système de priorités entre les règles est simple mais explicite. Ainsi, la cohérence de l'ordre de priorité donné peut être vérifiée de la manière suivante : si une description $d1$ est plus générale qu'une autre $d2$, elle ne peut être spécifiée dans une règle mise *avant* celle spécifiant $d2$, sous peine de rendre cette dernière inutile. Par exemple, si une règle indiquant de mettre tous les tableaux d'une partition donnée prend le pas sur une autre concernant tous les tableaux de cette partition d'une taille supèrieure à 200 octets, cette dernière règle est inutile.

Cohérence des règles avec les propriétés

Comme on l'a vu précédemment, des propriétés peuvent être spécifiées sur les partitions, par exemple celles indiquant qu'une partition ne peut être modifiée. Le compilateur vérifie si une règle entre en contradiction avec les propriétés des partitions concernées par la règle. Par exemple, une règle indiquant de placer certains objets dans une partition immutable n'est pas valide.

Retours à l'utilisateur

En plus des vérifications de ces propriétés, le compilateur prend en charge un certain nombre d'options permettant à l'utilisateur d'en savoir plus sur une politique donnée. Par exemple, il peut obtenir toutes les règles demandant un certain critère.

Une fois ces vérifications sémantiques effectuées, le processus se poursuit par la réalisation des optimisations sur la représentation interne.

5.3.4 Optimisations sur la représentation interne

L'évolution de l'état d'un objet pendant la phase de marquage peut, on l'a vu, être modélisée sous forme d'un graphe. On peut en réalité représenter toute règle de placement par un tel graphe. Les règles « simples » telles que celles concernant l'allocation des objets, ne contiennent donc que deux états : un état initial donné par l'allocation et un état final codant la mémoire dans laquelle doit être placé le nouvel objet.

Étant donnée cette représentation d'une politique de placement, le compilateur va effectuer un certain nombre de vérifications et d'optimisations. En premier lieu, tous les états initiaux sont vérifiés. Pour ce faire, une itération est faite sur tous les critères des nœuds concernés. Chaque `Criteria` vérifie s'il est compatible avec les autres via une surcharge de la méthode `verify()`. Les critères des autres noeuds représentant un état sont constitués des critères du noeud précédent et du critère étiquetant la seule transition menant à ce noeud. Chacun de ces `Node` est vérifié selon le même principe que celui des noeuds initiaux.

Une fois les graphes correspondant à chaque règle vérifiés, les parties de graphes qui le permettent sont fusionnées, afin de diminuer le nombre d'états nécessaires. Ce mécanisme

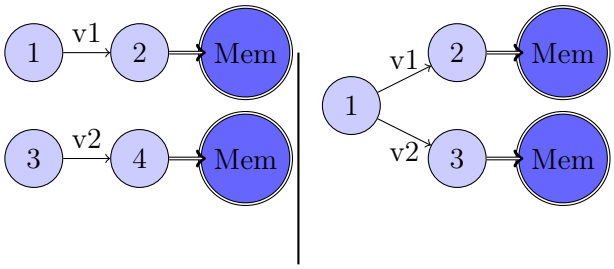


FIGURE 5.13 – Un exemple de l'unification de deux graphes

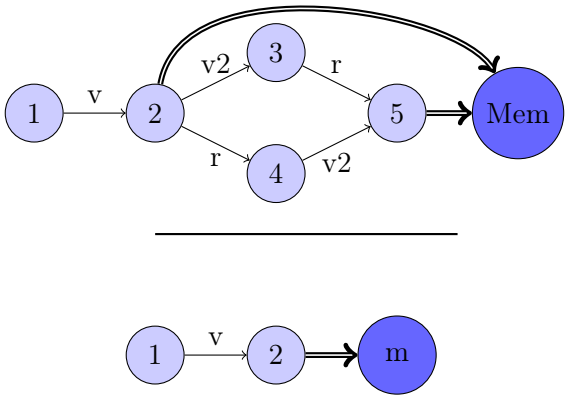


FIGURE 5.14 – Réduction de graphe

est illustré par la figure 5.13.

Une autre optimisation est réalisée sur les graphes. Une fois ceux-ci réunis, ils sont réduits de façon à supprimer les états inutiles. La figure 5.14 montre un exemple d'une telle optimisation. Sur cet exemple, le graphe est simplifié en supprimant une partie du graphe qui ne fait que spécialiser un état déjà atteint sans mener à des mémoires différentes.

Ces optimisations ont pour effet de simplifier les graphes, ce qui induit un code généré plus rapide. Un autre avantage très important de cette phase est de réduire dans le graphe le nombre d'états nécessaires.

Nombre d'états nécessaires Le nombre d'états est important car l'état d'un objet est codé, à l'exécution du système, sur un certain nombre d'octets contenus dans l'en-tête de chaque objet. Si le nombre d'états est trop grand, la taille de cette en-tête va être trop grande pour seulement envisager la mise en place de la solution proposée. On va donc chercher à évaluer le nombre maximum d'états nécessaires que requiert une politique donnée.

Chaque description d'objet différente demande au moins le codage d'un état. Ensuite, à chaque fois qu'un critère doit être vérifié, cela demande au moins un état supplémentaire. En fait, dès qu'il y en a plus d'un, des chemins différents apparaissent dans le graphe puisqu'on n'a aucune garantie sur l'ordre d'évaluation des critères pendant le suivi des références. Pour une règle donnée r_i, on note $n_{\&}(r_i)$ le nombre de critères évaluables

uniquement pendant le suivi de références. Le nombre maximum d'états nécessaires pour la règle en question est fonction du nombre de permutations possibles de ces critères, soit $n_{\&}(r_i)!$. Si l'état initial est compté pour la règle en question, le nombre d'états nécessaires est donc $n_{\&}(r_i)!+1$. Pour une politique entière, si on note N_r le nombre de règles de cette politique, le nombre maximum d'états $\overset{max}{n_e}$ nécessaires pour cette politique est donc :

$$\overset{max}{n_e} = \sum_{i=1}^{N_r} (n_{\&}(r_i)! + 1) \tag{5.5}$$

En exemple, prenons le cas d'une politique complexe, avec :
- 20 règles d'allocation ;
- 10 règles de collecte comprenant seulement 1 critère de référence ou 1 critère de valeur de champ ;
- 10 règles de collecte comprenant 3 critères de ce type.

Dans ce cas, le nombre maximum d'états est égal à 100 (application de la formule 5.5, ce qui demande seulement 7 bits dans l'en-tête des objets. Ce chiffre ne prend pas en compte les diverses optimisations par les compilateurs, dont le bénéfice dépend fortement de la politique considérée. Ce chiffre impacte également la taille des matrices générées.

Taille des matrices générées De façon à diminuer la taille des matrices générées par le compilateur, les états sont assignés à chaque nœud des graphes selon quelques règles bien précises. On note n_e le nombre total d'états requis, I l'ensemble des états initiaux ($card(I)$ le nombre d'éléments dans cet ensemble), et F l'ensemble des états finaux. Les numéros d'états sont attribués de cette façon :
- Les numéros codant un état final démarrent à 0 et sont consécutifs. Cela réduit la taille du tableau P ;
- Les numéros dans l'ensemble $\{I \backslash F\}$ (c'est à dire l'ensemble des états initiaux qui ne sont pas finaux). Cela réduit la taille des tableaux U, L, et T si les numéros d'états servant d'indices dans ces tableaux sont décrémentés de $(card(F)) + card(I \backslash F)$.

Au final, avec $card(N_v)$ (respectivement $card(N_L)$) la cardinalité de l'ensemble des valeurs produites par la fonction V (respectivement L), les tailles suivantes peuvent être observées (s_{state} est le nombre d'octets nécessaires pour coder un état) :

L : $(n_e - card(F) - card(I \backslash F) * s_{state}$;

U : $card(N_v) * (n_e - card(F)) * s_{state}$;

T : $card(N_L) * (n_e - card(F)) * s_{state}$;

P : $card(F)$.

Il n'existe aucune contrainte sur la représentation d'un numéro d'état pour un objet donné. Ceci dit, dans notre propre implémentation, nous avons choisi de réserver un certain nombre de bits sur les octets codant le numéro d'état pour coder directement la location d'un objet. Ce nombre de bits va la plupart du temps être compris entre 1 et 3 puisqu'il est peu vraisemblable d'avoir moins de 2 partitions, et plus de 8. En réalité, dans notre implémentation, ces quelques bits codent directement l'identifiant du gestionnaire mémoire associé à la partition concernée, ce qui permet dans l'architecture présentée au chapitre précédent d'avoir accès directement à la structure décrivant le gestionnaire mémoire (section 4.2.4). Cet avantage aurait été perdu avec l'autre solution consistant à

considérer ce critère comme les autres et à le coder dans le numéro d'état global. Quoiqu'il en soit, ce choix a peu d'impact sur le code généré.

5.3.5 Code généré

Une fois les graphes unifiés, on obtient un ensemble de graphes à partir duquel le compilateur va générer le code natif complétant le gestionnaire mémoire. Quatre étapes successives sont donc réalisées :

Génération de la fonction calculant l'état initial des objets Ces fonctions sont appelées lors de l'allocation d'un objet. Cela correspond à la fonction A de la section 5.3.1. En première solution, on peut considérer un parcours de toutes les racines afin de générer une série de conditions conduisant à l'affectation d'un état si elles sont vérifiées. Cependant, certains critères exprimés peuvent être les mêmes pour deux règles ; dans ce cas la condition générée est commune aux deux règles.

Construction et génération du tableau P À cette fin, un simple parcours des états finaux permet de générer le tableau. De façon à ce que la taille de ce tableau soit la plus petite possible, les numéros identifiant un tel état sont donnés de telle sorte qu'ils soient consécutifs et aient les premières valeurs (*i.e.* de 1 à n).

Génération de la fonction V Le même principe que pour l'allocation est utilisé, mais appliqué aux seuls critères de valeur de champ.

Construction et génération du tableau U Pour chaque transition étiquetée par un critère de valeur de champ, une entrée est insérée à U[i][j], où i est l'entier donné par V, et j est l'état de l'objet référencé.

Construction des tableaux L et T Toutes les transitions de chaque arbre sont parcourues et pour chaque `transition` t entre deux noeuds A et B étiquetés par un critère de référence :

- une entrée est ajoutée dans le tableau à deux dimensions L, aux coordonnées i et j ; i étant le numéro d'ordre de la référence dans un objet du type considéré, et j le numéro d'état du noeud A. Cette entrée est un numéro choisi de telle sorte que la série de nombres choisis pour toutes les règles commence par 0 et soient consécutifs ;
- une entrée est ajoutée dans le tableau à deux dimensions T, aux coordonnées i et j ; i étant le numéro entré précédemment dans le tableau L et j le numéro d'état du noeud B.

La génération de code est la dernière étape du processus de compilation. Il en a été retiré un certain nombre de résultats expérimentaux.

5.4 Quelques résultats expérimentaux

La section précédente a évalué la taille nécessaire au niveau de l'en-tête des objets pour coder les informations d'une politique de placement. Il reste maintenant à évaluer l'impact de la solution en termes de tailles du code et d'efficacité.

5.4.1 Efficacité

Le coût induit par l'utilisation du langage dédié est lié à deux opérations :
- le coût d'attribution d'un état initial pendant l'allocation d'un nouvel objet ;
- le coût additionnel au niveau de la phase de marquage, dû à l'examen des propriétés d'un objet.

J'ai donc comparé le temps d'un algorithme de marquage traditionnel (présenté en section 4.2.1) avec l'algorithme modifié prenant en compte l'évaluation des propriétés d'un objet. Les mesures ont été réalisées sur un ensemble d'applications Java comprenant plusieurs benchmarks de la suite du *JavaGrande Forum* (*JGF Sequential Benchmark Suite* [Bull 99]). Pour chaque application, un ensemble de politiques de placement a été testé. Les résultats sont présentés dans la table 5.2, et sont, pour chaque application, une moyenne des temps obtenus pour les différentes collectes de données et pour les différentes politiques testées.

Application	Time of the modified marking phase
moldyn	+5.8 %
montecarlo	+11.6 %
series	+6.6 %
lufact	+11.8 %
crypt	+7 %
raytracer	+9.7 %
dhrystone	+6.6 %

TABLE 5.2 – Impact of examining the properties of objects on the execution time of the marking phase

Ces résultats montrent que l'examen des propriétés des objets a un impact très faible sur le temps d'exécution puisque le temps nécessaire à la phase de marquage modifiée est seulement de 5.5 à 11.6 % supplémentaire par rapport à l'algorithme inital. Il est important de noter que la phase de marquage prend au grand maximum 50% du temps nécessaire au mécanisme de ramasse-miettes pour collecter les données, et que le temps total passé à collecter les données est généralement inférieur à 10 % du temps d'exécution total [Blac 04]. Ce qui donne donc un impact de moins de 1% sur le temps d'exécution total.

5.4.2 Taille du code

Comparer la taille du code écrit en utilisant le langage dédié avec l'équivalent écrit à la main en C, c'est-à-dire sans utiliser de matrices ni les optimisations faites par le compilateur qui rendraient le code difficile à maintenir. Un autre résultat important est la taille du code généré par le compilateur. La table 5.3 montre les résultats des mesures faites uniquement sur le code nécessaire au placement des données. Ce code concerne donc la gestion du placement durant les allocations et pendant les collectes de données. Ces résultats montrent que le code écrit en utilisant le langage dédié est environ cinq fois plus compact que son équivalent en code natif. Cela illustre parfaitement que le

Code	Taille	Ratio
Écrit à la main (code C)	2393 characters	100%
DSL	483	20.1%
Généré (code C)	2121	88.6%

TABLE 5.3 – Taille de code pour le placement des données

pacap/cardIssuer/CardIssuer.java

```
...
18  private byte [] t1_4 =    JCSystem.makeTransientByteArray((short) 4,   private byte [] t1_4 = new byte[4];
19                            JCSystem.CLEAR_ON_DESELECT);                 private byte [] t2_4 = new byte[4];
20  private byte [] t2_4 =    JCSystem.makeTransientByteArray((short) 4,   private byte [] t3_8 = new byte[8];
21                            JCSystem.CLEAR_ON_DESELECT);                 private byte [] t3_8 = new byte[8];
22  private byte [] t3_8 =    JCSystem.makeTransientByteArray((short) 8,   private byte [] t5_16 = new byte[16];
23                            JCSystem.CLEAR_ON_DESELECT);                 private byte [] temp = new byte[37];
24  private byte [] t4_8 =    JCSystem.makeTransientByteArray((short) 8,
25                            JCSystem.CLEAR_ON_DESELECT);
26  private byte [] t5_16 =   JCSystem.makeTransientByteArray((short) 16,
27                            JCSystem.CLEAR_ON_DESELECT);
28  private byte [] temp =    JCSystem.makeTransientByteArray((short) 37,
29                            JCSystem.CLEAR_ON_DESELECT);
```

FIGURE 5.15 – Allocations spécifiques remplacées par des allocations normales dans une application Java Card

langage est plus lisible et simplifie l'écriture de code relatif au placement de données. De plus, on constate une diminution de la quantité de code C correspondant, ce qui est principalement le reflet des optimisations du compilateur.

5.4.3 Séparation des préoccupations

Le design du langage dédié présenté est orienté vers un but principal : exprimer le placement des objets. Tout en améliorant et facilitant l'écriture de code concernant cet aspect, il reste assez puissant et flexible pour être capable d'exprimer différents types de politiques. Un des pré-requis du langage détaillé est le fait qu'il doit être capable d'exprimer toutes les politiques de placement existant actuellement.

Le schéma adopté par l'actuelle spécification Java Card est facilement décrit dans le langage présenté puisqu'il ne considère aucun déplacement de données entre zones mémoire. Concernant les partitions, deux partitions suffisent : une occupant toute l'EEPROM, une autre occupant toute la RAM. Concernant l'allocation, il suffit d'indiquer que par défaut les nouvelles instances seront allouées en EEPROM et de spécifier les règles indiquant les sites où les objets doivent être alloués en RAM. Le bénéfice en terme de simplicité d'écriture d'une application est illustrée par la figure 5.15.

L'autre schéma de placement existant actuellement correspond au modèle générationnel. Une politique mettant en place plusieurs générations est simple à écrire puisqu'il suffit d'indiquer que toutes les allocations par défaut doivent être faites dans une (petite)

partition que l'on appellera donc « nursery », et que tous les objets de cette partition doivent être déplacés dans une autre partition. Ceci se fait au moyen du critère de *location* décrit dans la Section 5.1.3. Selon le même principe, plusieurs niveaux de génération peuvent être utilisés.

Les résultats obtenus montrent que les avantages de l'utilisation du langage dédié ne se fait pas en contrepartie de performances catastrophiques. Ils encouragent ainsi la recherche de perspectives quant à l'utilisation des politiques de placement.

5.5 Perspectives quant à l'utilisation des politiques de placement

Les perspectives quant à l'utilisation du langage dédié mis au point concernent trois axes principaux, dont le plus important est à mes yeux sa dynamicité à l'exécution.

Dynamicité à l'exécution

La forme du code généré par le compilateur a pour but premier d'être efficace et compact. En second lieu, cette forme de code a été mise au point de façon à faciliter le chargement dynamique de politiques de placement. Il serait bien sûr impossible d'envisager le chargement complet du code écrit en langage dédié et de le compiler sur la cible. Le but est plutôt de laisser le travail hors ligne au compilateur. Celui-ci produirait alors (au moyen d'un terminal spécifique par exemple) le code suffisant pour modifier partiellement ou entièrement la politique de placement sous une forme chargeable dynamiquement par le système Java.

Cependant, le chargement de code natif pose de gros problèmes de sécurité. De ce point de vue, la forme du code utilisé à l'exécution a de grandes qualités puisqu'elle est composée en grande partie de tableaux. Or, des tableaux sont facilement chargeables dynamiquement puisqu'il peuvent être donnés sous la forme de classes Java. De cette façon, le chargement reste sécurisé. Cependant, il reste le problème des fonctions natives utilisées pour évaluer les valeurs de champs, et les propriétés des objets à l'allocation. Gérer le placement des données dans un code de haut niveau est l'une des solutions que l'on pourrait proposer. Cela paraît tout à fait faisable et très séduisant, puisqu'on peut imaginer de charger des classes complètement responsables de la gestion du placement, tout comme il est possible en Java de charger son propre chargeur de classes. Cependant, le système d'interactions entre le code de haut niveau et les couches basses du gestionnaire mémoire reste à mettre au point, et la perte de performances que l'on imagine à évaluer.

Une seconde perspective concernant le langage dédié est son extension via l'ajout de critères de placement.

5.5.1 Ajout de critères de placement

La spécification du placement des données pourrait être améliorée en ajoutant, dans le langage et le compilateur, le support pour des critères de placement additionnels. Même si je reste convaincu que les principaux critères ont été décrits, on peut en imaginer d'autres. Par exemple, un critère d'atteignabilité via, non pas un seul saut comme pour

le critère de référence, mais un ou plusieurs, pourrait se révéler utile, bien que sa prise en charge soit complexe à réaliser. Il permettrait d'indiquer, par exemple, que les accès aux objets atteignables depuis une structure de données soient accélérés.

Enfin, on peut envisager non pas l'extension d'une politique elle-même, mais l'extension de *l'utilisation* des politiques de placement.

5.5.2 Extension de l'utilisation des politiques de placement

Le langage dédié mis au point permet l'expression de politiques de placement, dans le sens le plus général qui soit. Il peut être utilisé de diverses manières. Tout d'abord, certaines règles de placement peuvent être données pour adresser la sécurité des données : par exemple, placer les données sensibles de certaines applications dans une zone mémoire donnée (non persistante, cryptée, ...). Ensuite, comme on l'a vu pour des critères de performance purs : placer les données nécessaires aux bons endroits de façon à accélérer les accès.

Cependant, son champ d'utilisation est plus large que cela. Un des aspects liés au placement des données est la gestion de la persistance des données. Un moyen simple de gérer la persistance des données grâce à un tel langage est d'appliquer une politique de placement donnée au moment défini par le système (selon un critère de temps, ou en fonction de certains évènements). Cependant, cet aspect n'a pas été approfondi, et mériterait certaines évaluations additionnelles : efficacité, compatibilité avec des mécanismes tels que le commit transactionnel. De plus, l'ajout éventuel dans le langage de critères en liaison directe avec la propriété de persistance reste à définir.

Un autre domaine d'application éventuel est l'ensemble des systèmes cherchant à avoir des garanties temps-réel. En effet, ces systèmes utilisent souvent des régions ayant certaines propriétés temps-réel. Cependant, l'utilisation de tels systèmes peut être fastidieuse. Une extension du langage par l'ajout de critères de cette nature peut être envisagée. Cependant, cela demande un important support de telles propriétés par le reste du système.

À propos du reste du système, le langage dédié doit pouvoir permettre son optimisation.

5.5.3 Optimisation

De nombreuses informations sont contenues dans une politique de placement, forcément bien plus que si le code était écrit dans une langage générique. Ces informations supplémentaires pourraient être utilisées afin d'optimiser les accès aux données. En effet, comme certaines mémoires doivent être accédées d'une manière spéciale (par page par exemple), un système embarqué utilise souvent des *accesseurs*. Un accesseur est un morceau de code utilisé pour lire et/ou écrire en mémoire qui, typiquement, cherche dans quelle mémoire est stockée une donnée et utilise le bon moyen pour y accéder en fonction de son placement. À partir d'une politique de placement, on pourrait déduire le placement de certaines données afin de générer un accesseur pour ce type de données. Exemple : s'il est possible de détecter que tous les tableaux sont toujours placés dans une mémoire, on peut utiliser directement le bon moyen d'y accéder dans le traitement du bytecode ASTORE (qui stocke une donnée dans un tableau).

5.6 Conclusion

Ce chapitre décrit une façon nouvelle de gérer le placement de données dans le cas de mémoires à objets. La proposition principale est l'utilisation d'un langage dédié à cette tâche. Ce langage permet d'écrire très simplement des règles de placement afin d'adapter la gestion mémoire selon des préoccupations matérielles, systèmes, ou applicatives. Cette proposition a fait l'objet d'un brevet [Marq 05b] et a également été publiée [Marq 07a, Marq 07b].

L'utilisation complète de la solution proposée demande la réservation de quelques bits par objet afin de stocker l'état de l'objet. Les performances sont satisfaisantes puisque les politiques de placement testées ne ralentissent que très peu l'exécution des applications considérées tout en permettant un placement précis, intelligent et simple à mettre en place. Cependant, il manque une étude plus complète de différentes politiques de placement, afin de mesurer toute la flexibilité et le gain en terme de performances permises par le langage proposé.

Cette solution orientée langage complète et renforce très naturellement l'architecture présentée dans le chapitre précédent puisque le code généré par le compilateur correspond au composant appelé *Placeur*. Cependant, sa mise au point n'a pas du tout été faite spécifiquement pour cette architecture, l'aspect placement de données étant au cœur de tout gestionnaire mémoire. D'ailleurs, l'idée de ce langage est née avant la mise au point de l'architecture précédemment décrite.

Sixième Chapitre

Gestion de la mémoire et déploiement

« Nous ne pouvons pas prédire où nous conduira la Révolution Informatique. Tout ce que nous savons avec certitude, c'est que, quand on y sera enfin, on n'aura pas assez de RAM. »

Dave Barry,
Chroniques déjantées d'internet.

Ce chapitre présente des solutions pour la gestion de la mémoire aux problèmes liés au déploiement d'un système embarqué, en particulier, la présence de mémoire non-réinscriptible (Read-Only Memory — ROM). La présence de ce type de mémoire en grande quantité est l'une des spécificités des systèmes embarqués, liée au fait que ces mémoires ont un point mémoire très faible (vingt fois moindre que celui de la mémoire Flash par exemple) et donc également un faible coût.

Tout d'abord, le déploiement d'un système embarqué est détaillé, et plus spécifiquement les systèmes Java. Ensuite, la question du placement des données dans les différentes mémoires lorsque le logiciel est embarqué sur le matériel est abordée. Deux solutions pour améliorer ce placement afin de tirer davantage partie de la mémoire non-réinscriptible sont proposées. Puis, un problème indirectement lié à la gestion de la ROM est adressé : le choix des racines d'atteignabilité du ramasse-miettes. Enfin, l'intégration des solutions présentées dans un véritable système Java est détaillée.

6.1 Déploiement d'un système embarqué

Aujourd'hui, malgré les avancées continuelles réalisées au niveau des mémoires réincriptibles persistantes, notamment de type Flash, les mémoires de type ROM restent présentes dans nombre d'architectures matérielles, principalement grâce à leur faible coût et le peu de place qu'elles nécessitent sur le silicium. De ce fait, les systèmes embarqués doivent être conçus de manière à gérer efficacement ce type de mémoire. Dans la suite de ce chapitre, on utilisera l'abus de langage consistant à désigner sous le terme de ROM toute mémoire utilisée comme mémoire non-réinscriptible, qu'elle soit effectivement de type ROM ou non. Nous allons donc maintenant voir de quelle façon les solutions actuelles utilisent cet espace mémoire durant le processus de *ROMization*.

6.1.1 Le processus de ROMization

La *ROMization* est le processus consistant à créer, au sein d'un environnement hôte, une image mémoire d'un système, directement exécutable par un équipement cible une fois copié bit-à-bit dans la mémoire de ce dernier. D'une manière générale, c'est la ROM qui est utilisée pour stocker cette image initiale du système embarqué. Le schéma 6.1 détaille ce mécanisme permettant d'embarquer un logiciel sur une cible. La production de l'image mémoire prête à être embarquée se fait aux moyens d'outils de *compilation croisée*. Outre leur rôle de compilation vers un code exécutable par le processeur cible, ces outils produisent, associées aux données, des informations indiquant dans quelles mémoires doivent être placées ces données. Bien entendu, on parle ici de données au sens large, comprenant également le code exécutable.

Plus précisément, à partir des fichiers objets obtenus en compilant les sources du système, la phase d'édition de liens permet de constituer l'exécutable proprement dit. Durant cette phase, les *cross-compilateurs* utilisent un *linker script* décrivant dans quelles mémoires doivent être copiées les différentes données décrites dans les fichiers objets au chargement du système. Lorsque la cible matérielle démarrera, ces copies seront effectuées

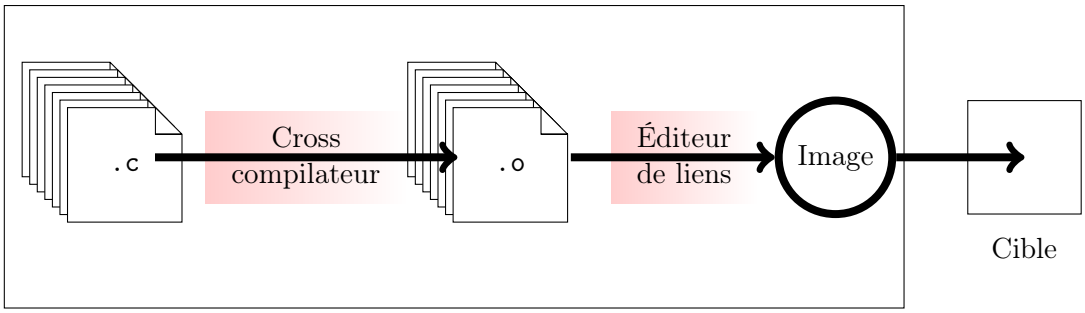


FIGURE 6.1 – Migration d'un système embarqué à partir d'un hôte vers le matériel cible

afin de mettre le système dans un état lui permettant de s'exécuter.

Même si les outils de cross-compilation comprennent des scripts de liaison (*linker scripts*) utilisables par défaut, il faut néanmoins préciser qu'il est possible, pour le programmeur, de donner ses propres indications quant au placement des données : soit en modifiant le script, soit en donnant des directives de compilation indiquant dans quelles sections des fichiers objets doivent se trouver les données (y compris le code).

Ce processus général est un peu particulier dans le cas de la ROMization de systèmes Java.

6.1.2 La ROMization dans le cas de systèmes Java

Dans le cas d'un système Java, le processus de ROMization est plus complexe et prend en charge les opérations suivantes :

- compiler la machine virtuelle ;
- pré-déployer des classes Java de façon à en obtenir une forme pré-chargée ;
- lier cette forme pré-chargée des classes avec la machine virtuelle.

Ce fonctionnement est illustré par le schéma 6.2. Ce schéma montre le processus de romization de J2ME ; le principe général est le même concernant les autres plateformes existantes.

L'opération de pré-déploiement est différente selon les variantes embarquées de Java, mais permet dans tous les cas :

- d'éviter de réaliser toutes les opérations d'initialisation de classes sur le matériel contraint ;
- d'optimiser l'utilisation de la mémoire. En effet, certaines données ne servent que pendant les opérations d'initialisation ; et peuvent donc être jetées une fois réalisées. De plus, une fois ces opérations réalisées, certaines données deviennent immutables (et peuvent donc rester en ROM) ou moins importantes (donc peuvent être mises dans une mémoire moins efficace).

J2ME Dans J2ME, l'outil *JavaCodeCompact* (JCC) permet de pré-charger les classes fournies. La forme pré-chargée ainsi obtenue peut être mise en ROM. Cependant, cet outil ne permet aucune modification du placement initial des données ;

Squawk Ce projet inclut une suite d'outils permettant de pré-déployer un ensemble de classes sur le système hôte, et notamment un *serializer* permettant la migration vers la plateforme cible ;

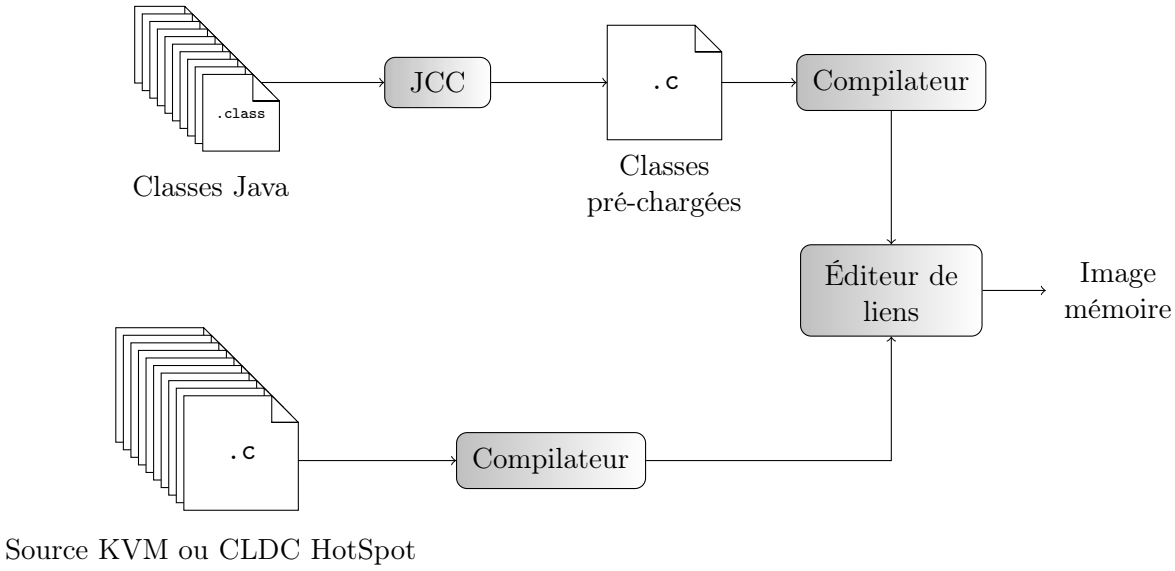


FIGURE 6.2 – Le processus de romization de J2ME.

JavaCard utilise le format de classe pré-chargée `cap`, prêt à l'exécution. De tels fichiers peuvent être liés à une machine virtuelle JavaCard, et les classes contenues placées en ROM ;

VM* et JEPES ont un fonctionnement sensiblement identique à ce niveau puisqu'ils lient tous deux les applications avec leur support d'exécution avant d'installer l'ensemble sur le système cible.

Comme on peut le constater, les solutions existantes se ressemblent beaucoup quant à la transition des données du système hôte vers le système cible. L'outil se chargeant de cette tâche est communément appelé un *ROMizer*. Il est responsable, entre autres, du placement des données initiales.

6.2 Placement des données initiales

Cette section montre les améliorations que l'on peut apporter au problème du placement des données initiales. Celles-ci sont principalement basées sur l'utilisation du langage dédié.

6.2.1 Utilisation du langage dédié

Les systèmes à base de machine virtuelle existants n'utilisent que des moyens très rigides — plus précisément : l'inscription en dur dans le code source — pour spécifier le placement des données initiales du système, y compris les données issues de la machine virtuelle telles que les instances de classes. En remplacement d'un tel modèle, l'objectif est d'extraire la préoccupation de gestion du placement, afin de la gérer séparément. Suivant le même raisonnement que pour la gestion du placement des données à l'exécution

(chapitre 5), la proposition est donc de gérer le placement des données grâce à une politique de placement écrite avec le même langage détaillé précédemment. En effet, ce langage présente toutes les qualités voulues puisqu'il permet de décrire facilement le placement des données en fonction de divers critères. Les avantages de l'utilisation de ce langage décrits au chapitre précédent sont d'autant plus vérifiés :

Expressivité la manipulation directe des concepts propres à la gestion mémoire facilite la compréhension et évite un grand nombre d'erreurs ;

Réutilisation de code les règles de placement d'une politique vont également pouvoir être réutilisées très simplement ;

Flexibilité du placement des données initiales :

- en fonction des applications déployées. Des règles spécifiques aux instances d'une application peuvent être écrites sans ajout de complexité, et sans toucher au code source du *ROMizer* ;
- de même, les règles peuvent être adaptées simplement en fonction du matériel disponible.

D'autres avantages se révèlent être d'autant plus vrais pour le placement des données initiales. En effet, la différence entre l'utilisation d'une telle politique de placement à ce niveau (placement des données initiales) et pour le placement des données à l'exécution consiste en deux points :

- nul besoin de gérer le placement des instances au fur et à mesure de leur création. Les données sont placées une fois le système figé et prêt à être *romizé*. Le graphe d'objet considéré à ce moment est donc figé et n'évoluera plus avant la reprise de l'exécution du système sur le matériel cible ;
- on dispose lors du pré-déploiement de tout le temps et toute la puissance nécessaires pour calculer un placement des données efficace et précis.

De ces deux points découle le fait que les politiques utilisées à ce niveau vont pouvoir être très précises et utiliser des analyses coûteuses en temps et en mémoire s'il le faut afin de placer chaque objet dans la mémoire la plus adaptée à son utilisation.

Plutôt que de considérer l'utilisation d'une deuxième politique dédiée au déploiement du système, il est préférable d'effectuer un retour sur les politiques de placement.

Retour sur les politiques de placement

La mise au point du langage dédié a été effectuée avec l'idée de pouvoir étendre son utilisation à d'autres moments où le placement des objets est nécessaire. Cette conception prend ici tout son sens. En effet, comme décrit dans le chapitre précédent, il suffit d'ajouter dans la politique de placement un ensemble de règles dans une section *init*. La figure 6.3 présente une petite politique disposant de cette section.

Du côté du compilateur, un terminal spécifique est bien sûr nécessaire afin de prendre en compte les différences avec le placement dynamique exposées précédemment. Celui-ci est en réalité plus simple puisqu'il ne génère aucun code et opère sur le graphe d'objets complet.

```
Partition part_rom {
    Memory is ROM;
};

Partition part_ram {
    Memory is RAM;
    Collector is compacting;
};

Partition part_eeprom {
    Memory is EEPROM;
    Collector is marksweep;
};

Class class;

String string;

char[] charArray;

Init {
    charArray => part_rom;
    class => part_rom;
    string => part_rom;
    part_ram;
};

Allocation {
    part_ram;
};

Collection {
     part_eeprom;
};
```

FIGURE 6.3 – Politique de placement étendue

Résultats expérimentaux

Nous allons évaluer le taux de placement en ROM à deux instants particuliers du déploiement en ROM. Le premier correspond à ce qui est fait dans la plupart des systèmes Java. Une fois les classes préchargées et liées. Le deuxième correspond à une approche plus poussée du déploiement : une fois les applications initialisées, c'est-à-dire après que les objets de type `Thread` aient été créés et les champs statiques des classes initialisés. Plus de détails sont donnés sur cette approche dans la thèse d'Alexandre Courbot [Cour 06].

Trois applications sont mesurées. La première est un simple « Hello World ». La seconde est une application de test provenant de chez *Sun* : *AllRichards*. Cette application exécute plusieurs versions de l'algorithme d'ordonnancement Richard. Cette application est intéressante car elle contient un grand nombre de classes (76). Enfin, l'application de test *Dhrystone* est mesurée et montre des résultats particuliers dûs aux structures de données statiques qu'elle utilise. Les mesures données ne tiennent pas compte des parties natives du sytème (couches basses de la machine virtuelle : gestionnaire mémoire, boucle d'interprétation de bytecodes). En effet, bien que cette partie comporte presque entièrement du code pouvant être mis en ROM, sa taille peut varier selon les implémentations.

Classes liées à la machine virtuelle Une fois les classes préchargées et liées, toutes les classes et méthodes peuvent être placées en mémoire non-réinscriptible. Les *bytecodes* associés aux méthodes sont immutables également car les dernières transformations de bytecodes ont lieu pendant la liaison des classes. Les résultats sont donnés dans la table 6.2.1.

Ces résultats indiquent clairement que la politique de placement est assez précise pour obtenir de très bons taux de placement en ROM. Ces taux sont donc d'environ 80% à ce stade du déploiement.

Benchmark	Taille (Ko)	en ROM (Ko)	% en ROM
HelloWorld	242	181	78%
AllRichards	319	258	81%
Dhrystone	247	194	79%

TABLE 6.1 – Taille du système (en Kilo-octets) après la phase de liaison

Applications déployées A ce moment du déploiement, les applications sont instanciées et certaines de leurs données sont créées, via les champs statiques des classes. La politique de placement prend alors tout son sens puisque le placement des données va pouvoir être spécialisé en fonction de l'application en écrivant du code de haut niveau dans le langage dédié :

```
int[] cryptArray {
    referredBy crypt.JGFCryptBench bench;
};
```

Conclusion Les bons résultats des taux de placement des données en ROM montrent que l'usage du langage dédié permet une description très précise et de haut niveau. Ces qualités du langage dédié développé et utilisé ici s'ajoutent à celles décrites dans le chapitre 5 qui sont communes à la plupart des langages dédiés (compacité, simplicité, flexibilité, réutilisation, etc.). Cependant, ces qualités sont difficilement comparables à d'autres solutions à cause du manque d'outils équivalents. Notamment, le fait de pouvoir placer, lors du déploiement, des données spécifiques à une application dans certaines mémoires (autres que la ROM), est difficilement comparable numériquement à d'autres travaux mais est un vrai bénéfice. Par exemple, les objets et codes des méthodes d'une application donnée peuvent, grâce à une politique de placement adéquate, être placés dans une mémoire efficace afin que son exécution soit plus rapide que d'autres.

6.2.2 Optimisation en monde fermé

Déployer et initialiser les fils d'exécution correspondant aux applications permet de réduire drastiquement la taille du système, si l'on sacrifie sa capacité à charger de futures applications ou données en cours d'exécution (mécanisme de chargement dynamique de classes). En effet, il est possible dans ce cas d'utiliser une analyse abstraite de code permettant de détecter quelles données seront accédées en lecture uniquement (et donc peuvent être placées en mémoire non-réinscriptible) par les applications déployées, ainsi que celles devenues inutiles (donc pouvant être collectées avant même la migration vers le système cible). Une telle analyse utilisée en conjonction avec une politique de placement adéquate donne de très bons résultats ainsi que le montre le tableau 6.2.2. On atteint ainsi des taux de placement en ROM avoisinant les 90% dans les cas standards. Le taux correspondant à l'application `Dhrystone` est faible pour une bonne raison : cette application alloue en donnée statique un tableau d'une taille anormalement élevée de 65 Ko qui sera rempli en cours d'exécution. Ces résultats sont le fruit d'une collaboration avec Alexandre Courbot qui a précisé et amélioré le principe de l'analyse abstraite de code utilisé ici [Cour 05].

Application	Taille	en ROM	% en ROM
HelloWorld	11326	9795	86%
AllRichards	74134	68315	92%
Dhrystone	86558	14316	16%

TABLE 6.2 – Taille du système (en octets) après la création des processus et leur initialisation

6.3 Optimisation des racines d'atteignabilité

L'état initial du système a un impact important sur son comportement en fonctionnement. Un de ces impacts concerne les racines d'atteignabilité du ramasse-miettes. Cette section détaille cet aspect et présente plusieurs optimisations de l'ensemble d'objets constituant ces racines. On nommera ces objets *objets racines*. Les objets en mémoire réinscriptible seront caractérisés *mutables* tandis que les autres seront dits *immutables*.

6.3.1 Le rôle et la définition des racines d'atteignabilité

Comme mentionné précédemment, quand le système démarre, un certain nombre de données sont copiées dans les mémoires réinscriptibles disponibles (RAM, EEPROM, Flash, etc.) et qui servent de mémoire de travail. Une fois démarré, le système est donc constitué d'objets en ROM et d'autres en mémoire réinscriptible, qui se référencent mutuellement, constituant un graphe d'objets. Au cours de l'exécution du système, ce graphe va évoluer en fonction des allocations d'objets, des collectes automatiques, et des créations et effacements de références entre objets.

Lorsqu'une collecte de données est déclenchée, un certain nombre d'objets de ce graphe va être utilisé comme racines d'atteignabilité pour la phase de marquage. Cette phase est nécessaire pour la plupart des algorithmes de ramasse-miettes, comme vu précédemment. Cet ensemble d'objets varie naturellement selon la machine virtuelle considérée ; cependant, on va y trouver traditionnellement la liste des *threads* en cours d'exécution ou leurs piles d'exécution associées, ainsi que les chargeurs de classes (*classloaders*). En effet, les piles d'exécution possèdent des références vers tous les objets rattachés à des variables locales ; la liste des *classloaders* permettant d'atteindre toutes les classes, et donc toutes les méta-données et objets attachés à des variables statiques encore utilisées. On nommera *racines d'origine* l'ensemble des objets racines dans le système en pré-déploiement.

De très loin, cet ensemble d'objets peut donc être constitué de n'importe quels objets du graphe d'objets au moment de la collection de données. Cependant, les racines sont forcément présentes dans le système initial. En effet, le graphe d'objets durant une collection est une modification, au gré des créations/destructions d'objets/références. Donc premièrement, même l'état initial de ce graphe d'objets doit pouvoir subir une collection et contient donc des objets racines. Deuxièmement, les données non présentes dans le système initial sont susceptibles d'être effacées en cas de redémarrage du système. Par conséquent, les véritables racines d'atteignabilité du système, à partir desquelles n'importe quel objet peut être atteint, sont forcément contenues parmi les objets immortels, c'est-à-dire dans le système initial. De plus, à partir des expériences menées, j'ai pu ob-

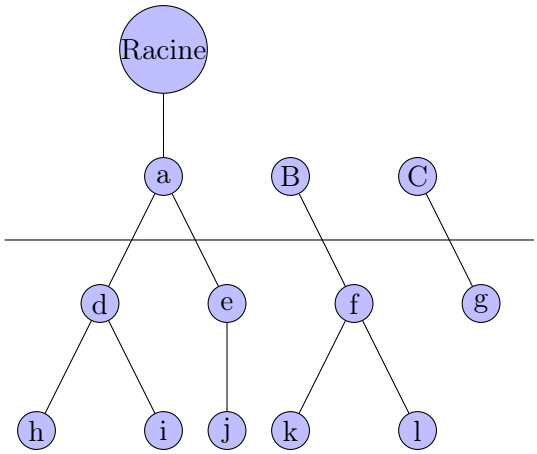


FIGURE 6.4 – Système initial et racines d'atteignabilité

server que les racines sont souvent situées en ROM ou référencées depuis la ROM. De plus, on peut toujours se ramener à ce dernier cas en créant un objet en ROM contenant uniquement des références vers les éventuels objets-racines qui ne se conformeraient pas à une de ces deux possibilités. On va donc considérer dans la suite que cette hypothèse est vérifiée. La figure 6.4 schématise cet état initial du système, avec un ensemble d'objets en mémoire réinscriptible et d'autres en ROM.

La suite détaille comment ce premier ensemble d'objets racines peut être optimisé, notamment pour éviter le parcours des références en ROM.

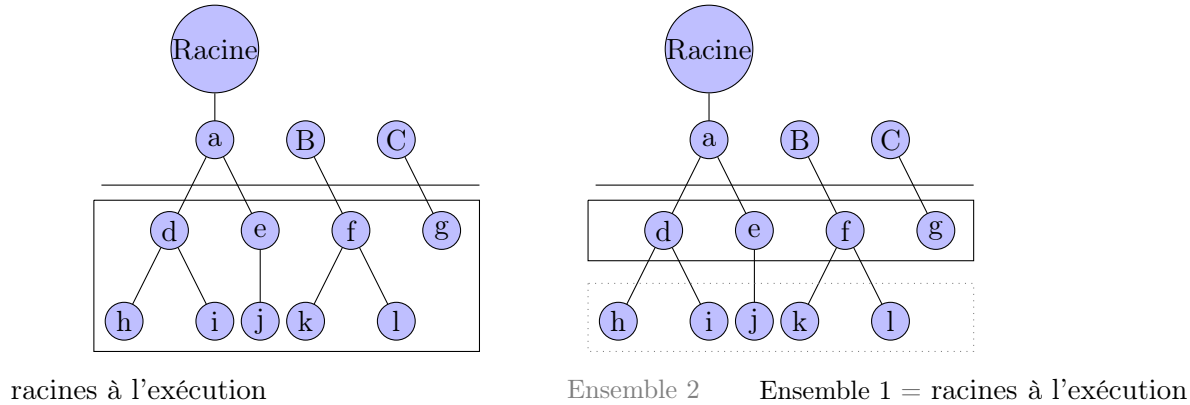
6.3.2 Éviter le parcours des références en ROM

Plusieurs problèmes se posent dès lors que l'on a des références vers des objets en mémoire non-réinscriptible. Tout d'abord, lorsqu'une référence vers un objet est suivie, si cet objet est en ROM, aucune marque ne peut être mise puisque la mémoire est non-réinscriptible. Cela implique donc de mettre les marques des objets en ROM dans des champs de bits, ce qui peut amener à réserver une large partie d'une mémoire réinscriptible à cette seule fin puisque le nombre d'objets en ROM peut être très important.

Ensuite, la phase de marquage est coûteuse, à cause du parcours des références de chaque objet. Or, le fait de parcourir les objets en ROM peut paraître inutile puisque la mémoire dans laquelle ils sont stockés est irrécupérable. Ces objets ne peuvent donc être effectivement collectés. De plus, il est impossible d'envisager de ne pas marquer les objets en ROM puisque l'algorithme de marquage utilise la marque mise afin de ne pas boucler indéfiniment.

Une première approximation

Considérons une première optimisation, consistant à :
- ne pas suivre les références vers des objets en ROM lors du marquage ;
- prendre comme ensemble racine tous les objets en mémoire réinscriptible présents dans le système initial. Cet ensemble est mis en évidence par la Figure 6.5(a).



(a) Une première approximation

(b) Une solution satisfaisante

FIGURE 6.5 – Racines d'atteignabilité : deux optimisations successives

De cette manière, les objets en ROM ne sont pas parcourus et leurs références ne sont pas examinées, ce qui réduit le temps consacré au marquage. Le marquage reste correct puisque les objets alloués par le système — s'ils doivent être conservés — seront forcément rattachés à un objet en mémoire réinscriptible. Par construction, chacun de ces objets nouvellement alloués sera donc référencé par au moins un objet mutable présent lors du démarrage, c'est-à-dire l'ensemble racine considéré.

Cette solution ne permet pas de collecter tous les objets en mémoire modifiable puisque certains objets mutables présents lors du démarrage peuvent devenir inutilisés. Faisant partie de l'ensemble racine, de tels objets ne pourraient être collectés.

Cette limitation nous amène à optimiser cette proposition afin d'obtenir une solution satisfaisante.

Une solution satisfaisante

Si on regarde d'un peu plus près le graphe d'objets, on distingue deux types d'objets mutables existent :

1. Les objets directement référencés par un objet immutable.
2. Les autres, nécessairement référencés par les premiers.

Les objets de la première catégorie ne peuvent pas bouger en cours d'exécution puisqu'il existe une référence vers eux contenue dans un objet immutable. Cette référence ne peut donc être mise à jour. De plus, ces objets ne peuvent même pas être collectés à moins d'avoir un moyen de détecter les objets en ROM immutables, ce qui est coûteux en espace et en temps, comme mentionné précédemment. L'ensemble racine peut donc être constitué uniquement de ces objets. La figure 6.5(b) illustre ce mécanisme. On y trouve ainsi les objets mutables mais non-déplaçables (ensemble 1) et les objets mutables déplaçables. Cette solution n'a que des avantages sur la précédente puisqu'elle permet de diminuer la taille de l'ensemble et autorise la collecte de tous les objets inutilisés de l'ensemble 2. Elle permet également de ne pas parcourir les objets immutables, mais ne

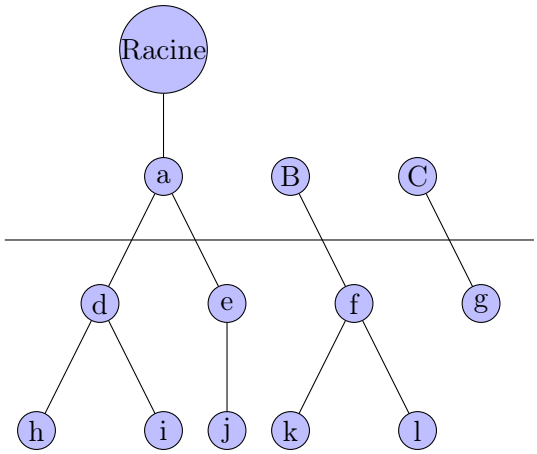


FIGURE 6.6 – Réduction de l'ensemble racine

détecte donc pas les objets immutables de l'ensemble 1 inutilisés.

6.3.3 Réduction de l'ensemble

Le principal défaut des solutions présentées précédemment est qu'elles ne permettent pas de détecter les objets immutables qui seraient devenus inutiles (inatteignables depuis la racine d'origine). Bien que ces objets ne puissent être collectés effectivement, certains objets référencés par ceux-ci (ensemble 1 du schéma 6.5(b)) pourraient l'être s'ils n'étaient pas dans l'ensemble racine puisqu'ils sont en mémoire réinscriptible.

Plus précisément, l'ensemble des objets immutables pouvant éventuellement être détectés comme inutiles sont ceux référencés uniquement par des objets mutables. Cet ensemble est noté B sur le schéma 6.6. L'ensemble (C sur ce même schéma) des objets mutables pouvant être détectés comme inutiles sont ceux référencés par un objet de l'ensemble B. On considère donc l'optimisation suivante :

- Les objets de l'ensemble C sont exclus de l'ensemble racine ;
- Le problème est qu'ils ne seront pas atteints lors d'une éventuelle phase de marquage. Pour palier à cela, lors de la production de l'image binaire, on joint au système la liste des objets accessibles depuis chaque objet en ROM de l'ensemble B ;
- Lors d'une phase de marquage, lorsqu'une référence vers un objet de cet ensemble B est suivie, la liste associée à cet objet donne les références vers les objets mutables qui doivent être marqués.

De cette manière, le marquage reste correct : on atteint finalement les objets de l'ensemble C via des listes constituées à l'avance plutôt que par un parcours récursif des références des objets de l'ensemble B. Si un objet de l'ensemble B n'est pas atteint, les objets mutables uniquement accessibles par lui ne seront pas marqués et vont pouvoir être collectés. Concernant ces listes, leurs tailles sont évidemment le défaut principal de cette solution. Cependant, deux éléments font que ce défaut est à relativiser :

- Ces listes sont définitives. Elles ne sont pas mises à jour en cours d'exécution car les références des objets en ROM ne changent plus. Elles vont donc pouvoir être stockées en ROM ;

– Des listes identiques n'ont pas besoin d'être instanciées plusieurs fois, puisqu'elles n'évolueront plus. De plus, un grand nombre d'objets peuvent avoir des listes identiques.

Au final, cette solution demande un peu de place en mémoire non-réinscriptible afin de pouvoir collecter des données en mémoire réinscriptible. Par rapport à la solution précédente, le temps de marquage est augmenté puisque des listes sont parcourues. Ce temps est bien sûr nécessaire puisqu'on marque des objets supplémentaires mais il est bien moindre que le temps nécessaire pour marquer les mêmes objets en parcourant récursivement les références de l'ensemble B.

6.4 Intégration dans un système Java : JITS

Cette section détaille la mise en place des solutions proposées dans cette thèse dans un véritable système Java embarqué. Le système considéré ici est encore une fois JITS. Le système JITS est donc présenté, en particulier son schéma de déploiement. Puis, l'intégration des solutions proposées sera expliquée. Le lecteur trouvera également les informations concernant la mise en place des contributions détaillées dans les chapitres 4 et 5.

6.4.1 Approche retenue

Comme vu précédemment, la phase de pré-déploiement d'un système Java sur le système hôte comporte plusieurs phases, en fonction de la solution considérée. Sur les architectures les plus récentes, un certain nombre d'outils sont en charge des opérations de pré-déploiement. Par exemple, la suite d'outils de Squawk intègre un charger de classes, un vérifieur de classes, un optimiseur de code, ainsi que le *serializer* chargé de ROMizer le système pré-déployé. L'architecture de JITS est comparable à celle-ci, avec la mise en œuvre de plusieurs outils ayant chacun un rôle bien défini. L'approche retenue pour implémenter les solutions proposées dans ce chapitre est la suivante : créer un outil indépendant qui se charge des opérations de pré-déploiement spécifiques à la gestion mémoire.

6.4.2 Déploiement dans JITS

Pour déployer le système sur la machine hôte, JITS utilise plusieurs outils indépendants. L'un d'entre eux, que l'on nommera *amorceur* par la suite, est un outil d'amorçage servant à créer un système minimal lui permettant de s'exécuter (par exemple, l'ensemble minimal de classes nécessaires sont chargées). Le système produit est un ensemble d'instances de classes que l'on appellera un `ObjectsPool` (du nom de la classe Java qui le représente). Cette opération est appelée *bootstrap*.

La migration vers un code pouvant être compilé puis lié aux parties natives de la machine virtuelle (interpréteur, fonctions natives, gestionnaire mémoire) est réalisée par le *migreur*. Son rôle est donc sensiblement le même que le *Serializer* de Squawk.

Entre ces deux outils de début et fin de déploiement, plusieurs outils peuvent être utilisés. Ces outils sont complètement indépendants les uns des autres du point de vue du code et architecturés autour de la notion d'ObjectsPool dont ils prennent une instance en

entrée et produisent une instance en sortie après avoir, soit modifié les données présentes, soit ajouté des données, soit ajouté de l'information via l'utilisation de méta-données. Les différents outils utilisés par JITS sont les suivants :

- Le *configurateur* permet de modifier le système sans avoir à faire appel à la machine virtuelle, par accès direct à ses objets. Il peut notamment être utilisé pour remplir des opérations comme affecter une adresse IP et une table de routage à la pile TCP/IP du système ;
- Le *vérifieur* s'assure de la consistance du système. En particulier, il vérifie qu'aucun flux d'entrée/sortie n'est ouvert au moment de la migration ;
- Le *visualisateur* fournit une vue graphique du système, en montrant notamment les références entre objets et en donnant certaines informations globales comme le nombre d'instances d'une certaine classe. Il permet ainsi de visualiser les particularités et éventuellement les problèmes d'un système ;
- La *JJVM* La *machine virtuelle Java-Java*[1] (*Java-Java Virtual Machine — JJVM*) prend en entrée un `ObjectsPool`, et le fait évoluer selon la spécification de la machine virtuelle Java [Lind 99]. Il s'agit donc d'une machine virtuelle au sens premier du terme. N'ayant pas d'état interne contenu ailleurs que dans l'`ObjectsPool`, l'état du système peut être capturé à n'importe quel moment de son exécution par simple sérialisation ;
- Le *spécialiseur* de système prend en entrée un `ObjectsPool` et en fournit une version spécialisée en sortie, fonctionnellement équivalente mais purgée de tous les objets détectés comme inutilisés dans le flot d'exécution futur du système, et dont les éléments restants sont spécialisés pour son exécution future (méthodes, structures de données, etc.).

Bien que ces outils soient indépendants au niveau du code, certains ont besoin de se communiquer des informations concernant le système déployé. De plus, par souci d'efficacité, un certain nombre d'informations sur les données contenues dans l'ObjectsPool ont besoin d'être centralisées. Par exemple, à chaque objet est associé un nom qui est stocké dans une table. Ou encore, le nom de la cible matérielle pour laquelle le système est construit peut également concerner plusieurs outils. Pour cela, deux sortes de méta-données peuvent être ajoutées dans un ObjectsPool :

- des *attributs d'objet* peuvent être associés à chaque instance de classe ;
- des *attributs de pool* peuvent être ajoutés à l'`ObjectsPool` lui-même.

Cette architecture interne d'un ObjectsPool est schématisée par la figure 6.7. C'est au sein de cette architecture qu'il s'agit de réaliser l'intégration de la gestion mémoire.

6.4.3 Intégration de la gestion mémoire

Conformément à l'approche décrite, un outil dédié à la gestion de la mémoire a été mis en place. Typiquement, cet outil précède l'action du migreur dans la chaîne de pré-déploiement. À ce moment du pré-déploiement, tous les objets du système sont présents et le graphe d'objets initial est constitué ; toutes les opérations relatives à ce graphe peuvent donc être effectuées.

Lorsqu'il est lancé, cet outil effectue, dans l'ordre indiqué, les tâches suivantes :

1. La machine virtuelle Java-Java porte ce nom car il s'agit d'une machine virtuelle Java écrite elle-même en Java.

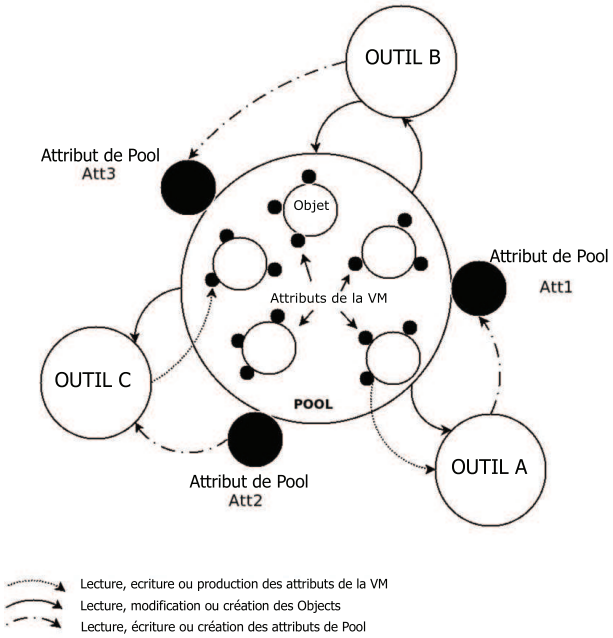


FIGURE 6.7 – Architecture du système de pré-déploiement

1. Prend en entrée, outre un `ObjectsPool`, un certain nombre de paramètres, dont les principaux sont :
   – nom du fichier contenant la politique de placement,
   – nom du fichier contenant la description du matériel,
   – nom de la cible matérielle ;
2. Lance la compilation de la politique de placement, ce qui inclut les sous-tâches d'analyses lexicale et sytaxique, et de réalisation des optimisations et vérifications. Enfin, les différents terminaux du compilateur font leur office :
   – les différentes structures associées à chaque gestionnaire mémoire sont générées,
   – le code du *placeur* est généré,
   – le back-end de la section *init* annote les objets avec l'indication de placement ;
   Pour cette dernière tâche, un *attribut d'objet* est associé à chaque objet de l'*ObjectsPool* ; un *attribut de pool* est ajouté, décrivant les propriétés des partitions décrites dans la politique.
3. Constitue l'ensemble des racines d'atteignabilité en se basant sur les annotations du compilateur et en analysant les liens entre les objets ;
4. Donne en sortie un `ObjectsPool` annoté.

Cet outil dédié à la gestion de la mémoire réalise donc l'intégration de toutes les propositions effectuées dans ce chapitre et les deux précédents au sein de la plateforme JITS. À noter que les annotations qu'il effectue sur l'`ObjectsPool` sont également utiles au migreur qui se base notamment sur celles-ci pour générer des tas contenant les objets placés dans les partitions correspondantes.

Ce faisant, il contribue à séparer la préoccupation de gestion de la mémoire, en l'isolant au sein d'un outil dédié, au rôle précis, et aux paramètres d'entrée et de sortie

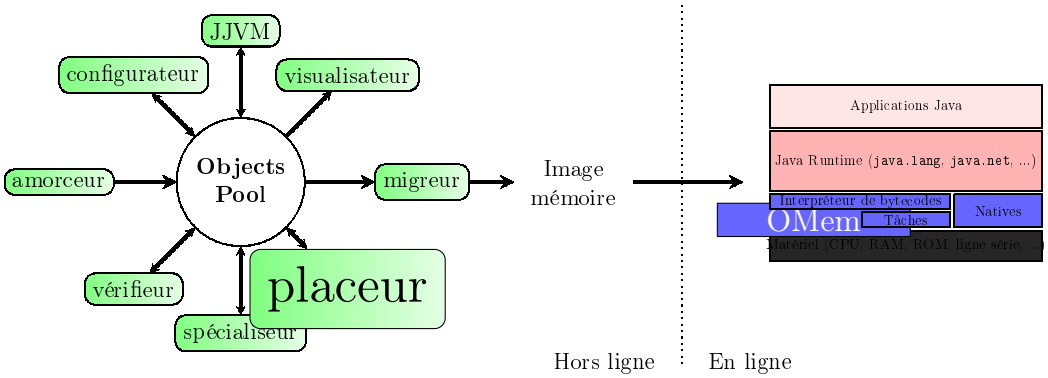


FIGURE 6.8 – Gestion de la mémoire dans JITS

bien identifiés.

6.5 Conclusion

Ce chapitre a décrit comment le déploiement d'un système à objets pouvait impacter la gestion de la mémoire à l'exécution. Une amélioration de la gestion du placement des données pendant ce déploiement a été décrite [Marq 05a] ; elle utilise le langage décrit dans le chapitre précédent. Un autre impact, également lié à ce placement, est la définition des racines d'atteignabilité [Grim 05]. Ces aspects sont principalement affectés par la présence d'espaces mémoire physiquement ou logiquement interdits en écriture.

Ce chapitre a également été l'occasion de présenter comment le système de gestion mémoire s'intégrait dans une véritable plateforme Java. De même, le déploiement a permis de montrer l'utilité du langage de placement précédemment décrit dans un autre contexte que l'allocation et la collection de données puisque l'écriture d'une politique de placement a été enrichie d'un troisième ensemble de règles réservé à l'initialisation du système. La facilité d'intégration de cette extension aux politiques est encourageante quant à l'utilisation du langage dédié pour d'autres contextes que ceux présentés jusqu'à présent.

La figure 6.8 montre la place des travaux effectués dans l'architecture du projet JITS.

Septième Chapitre

Conclusion et perspectives

« Il ne faut pas penser à l'objectif à atteindre, il faut seulement penser à avancer. C'est ainsi, à force d'avancer, qu'on atteint ou qu'on double ses objectifs sans même sans apercevoir. »

Bernard Werber,
La révolution des fourmis.

Ce chapitre clôt le présent mémoire de doctorat. Il a trois fonctions. Premièrement, fournir un résumé des contributions de la thèse. Ensuite, critiquer les travaux effectués en présentant les limites de l'approche adoptée. Enfin, présenter les perspectives et besoins d'enrichissements que les travaux effectués ne manqueront pas de susciter.

7.1 Contributions

Le but de cette thèse était de réconcilier systèmes embarqués et gestion automatique de la mémoire. Ce domaine est aujourd'hui largement sous-étudié, les solutions existantes étant très spécifiques. Par exemple, elles ne sont opérationnelles que sur un unique type d'architecture. Cette thèse est donc un premier pas dans la mise au point de solutions plus génériques de gestion de la mémoire. J'ai principalement cherché à mettre au point des solutions indépendantes mais consistantes entre elles, et formant ensemble une proposition complète et fonctionnelle.

Un premier résultat de ma thèse est d'avoir réalisé une étude de la complexité des algorithmes de ramasse-miettes existant dans le contexte de l'embarqué. À partir de cette étude, j'ai pu identifier les paramètres fins qui influent sur l'efficacité de ces algorithmes. J'ai également pu montrer qu'il est nécessaire d'adapter, en terme de choix de ramasse-miettes, la gestion mémoire à la fois aux applications et au matériel.

J'ai défini et implémenté, au sein d'un vrai système Java, une nouvelle architecture de gestion de la mémoire adaptée aux systèmes visés. Le modèle proposé fait le pont entre les différentes contraintes exprimées. D'une part, les contraintes matérielles dues aux propriétés très différentes des mémoires que l'on peut trouver sur les classes d'appareils considérées. D'autre part, les spécificités des différents mécanismes de ramasse-miettes existant. Le principe de base de cette architecture est de permettre la gestion de chaque espace mémoire (physique ou logique) par un gestionnaire adapté aux propriétés de chacun de ces espaces, plutôt que d'implémenter un gestionnaire spécifique à une seule architecture mémoire.

Afin d'isoler la gestion du placement des données tout en améliorant la flexibilité, la portabilité et les performances de la gestion mémoire dans le contexte présenté, j'ai mis au point un langage dédié à la gestion de mémoires à objets. Ce langage dédié permet d'écrire des politiques de placement complètes, qui sont ensuite validées et optimisées par le compilateur de ce langage. Ce compilateur produit un code intégré à la machine virtuelle. Une politique de placement comporte à la fois la définition des espaces mémoire et des règles de placement permettant de spécifier le placement des données aux niveaux système et applicatif.

J'ai adressé le problème de la présence éventuelle de mémoire non-réinscriptible en proposant plusieurs optimisations. Premièrement j'ai proposé des moyens de maximiser son utilisation. Deuxièmement, le marquage des objets survivants à une collection de données a pu être amélioré.

Toutes les contributions exposées constituent un ensemble cohérent, preuve en est qu'elles ont toutes été implémentées au sein de la même plateforme d'expérimentation, JITS, développée au sein de l'équipe.

L'approche retenue pour cette thèse a bien sûr ses limitations, imposant des limites

aux travaux présentés.

7.2 Limites aux travaux présentés

La première limite des travaux présentés concerne à mon avis un problème humain. En effet, les solutions proposées autour d'une nouvelle architecture de gestion mémoire et du placement des données nécessite une mise au point par un acteur qui n'est ni le programmeur d'application ni le concepteur de circuits électriques. Il s'agit donc d'un concepteurs du système ayant à la fois la connaissance du matériel utilisé, notamment les performances des mémoires, et du domaine applicatif. Cependant, cette tâche est facilitée grâce à l'utilisation du langage dédié dont la conception intègre en quelque sorte l'expertise du domaine. De plus, cette contrainte est à reconsidérer au regard de la situation habituelle pour laquelle la préoccupation de gestion mémoire est incluse dans les applications elles-mêmes.

L'évaluation numérique des solutions proposées aurait gagné à être plus poussée. En particulier, la robustesse des solutions proposées auraient pu être éprouvée sur un plus grand nombre d'applications, aux profils variés. Cependant, les mesures effectuées valident les mécanismes proposés.

Les diverses évaluations auraient parfois gagné à être comparées avec les performances de systèmes existants. Cette tâche s'est avérée très difficile car on ne trouve pas d'équivalent aux solutions proposées dans cette thèse, en raisoin des contraintes prises en compte : portabilité, non-modification du code source applicatif, validité de l'architecture sur différentes configurations matérielles. Néanmoins, il aurait été intéressant de connaître le bénéfice en terme de performances par exemple de l'utilisation de diverses politiques de placement. Mais étant donné les vitesses d'accès des mémoires, je suis convaincu que même en terme de performances, le coût de l'utilisation des politiques de placement, couplé ou non à l'architecture proposée, est compensé par l'intelligence et la flexibilité du placement pouvant être exprimé.

De manière plus précise, les travaux présentés reposent sur certaines hypothèses, que je pense cependant être mineures. Ainsi, l'utilisation du langage dédié nécessite, au moins dans la version proposée dans ce document, l'utilisation de quelques bits dans l'en-tête des objets. Bien que ces quelques bits ne constituent pas une grosse contrainte, elle n'est tout de même pas sans conséquences, même légères, sur la conception du système sous-jacent. Ensuite, le langage dédié n'a été testé que sur un système Java. Cependant, les mécanismes sont les mêmes pour d'autres langages orientés objet (par exemple C#) et l'adaptation à l'un de ceux-ci ne poserait pas de problèmes particuliers.

7.3 Perspectives

Les travaux effectués autour de la gestion de la mémoire ouvrent, du fait de leur nouveauté, un certain nombre de perspectives. J'imagine ces perspectives selon quatre axes.

Le premier concerne l'utilisation plus large du langage dédié. Je pense en effet que cette utilisation peut être étendue pour gérer le placement des données à différents moments de la vie du système. Ainsi, de la même façon que le placement des données est

évalué à l'initialisation, pendant l'allocation des données, et pendant leur collecte, on pourrait envisager d'appliquer un ensemble de règle de placement à d'autres moments. Par exemple, un système en cours d'exécution produit des données que l'on veut persistantes, donc placées dans une mémoire adéquate. La persistance des données pourrait être définie par l'application d'une politique de placement à un moment judicieux : fermeture d'un flux, extinction de l'appareil, ou encore indication explicite par le programmeur.

Les perspectives liées au langage incluent également les améliorations de celui-ci. Ainsi, on peut imaginer la prise en charge de critères et propriétés additionnelles. Le processus de compilation pourrait lui aussi être amélioré. En effet, l'utilisation d'un tel langage, parce qu'il est spécifique à un domaine particulier, pourrait permettre de calculer des propriétés additionnelles. En particulier, on pourrait chercher à analyser une politique afin de connaître le placement de chaque objet pour une application donnée. Par exemple, ces informations pourraient être utilisées pour obtenir des propriétés temps-réel plus précises (par exemple le calcul du pire temps d'exécution). Ou encore, comme on l'a mentionné dans le chapitre 5, les accès aux données pourraient être accélérés par la génération de code spécifique.

Le second axe est en rapport avec le premier. Puisqu'il est possible d'optimiser la gestion mémoire en exploitant les informations du modèle orienté objet, on pourrait envisager d'exploiter ces informations pour d'autres aspects. Par exemple, un besoin des systèmes embarqués est le maintien de la cohérence des données dans le cas où le système est susceptible de subir des arrêts brutaux. Cette cohérence est généralement assurée par des mécanismes transactionnels. De la même façon que l'exploitation du modèle de programmation à objets permet l'écriture de politiques de placement très précises, il doit permettre l'optimisation de ces mécanismes transactionnels.

Le troisième axe concerne la dynamicité de la gestion mémoire à l'exécution. J'imagine un modèle dans lequel une application pourrait être chargée dynamiquement en même temps que des informations sur les besoins mémoire de cette application. À partir de ces informations, la gestion de la mémoire pourrait être dynamiquement adaptée à ces besoins. Par exemple, le type de ramasse-miettes pourraient être changé afin de coller aux besoins de la nouvelle application. Ou encore, un espace mémoire particulier à cette application pourrait être créé, de façon à répondre à des exigences très spécifiques. En dernier exemple, on peut imaginer un affinage dynamique du placement des données afin de s'adapter aux données qui seront allouées par l'application.

Quatrièmement, au vu des différences considérables entre les types de mémoires existant, il est possible de mettre au point des algorithmes de ramasse-miettes adaptés à certaines de ces mémoires. Ainsi, je pense que la conception d'un ramasse-miettes dédié à la gestion de la mémoire Flash est prometteuse, car les accès par pages et les temps d'accès de ce type de mémoire nécessitent d'y accéder de manière ad hoc et intelligente.

Enfin, un axe peu abordé précédemment est l'optimisation de la consommation énergétique de la gestion mémoire. Il devrait se révéler intéressant de mettre au point des solutions de gestion de la mémoire économes de ce point de vue. Un premier travail pourrait être d'étudier le modèle énergétique de différentes mémoire afin de les gérer par un ramasse-miettes adapté. Une deuxième optimisation à réaliser consisterait à placer et déplacer les données de façon à n'allumer que le minimum de bancs mémoire. Cela nécessite d'évaluer le coût énergétique de l'allumage et l'extinction des bancs ainsi que le coût du déplacement des objets. À partir de ces données, différentes stratégies de placement

de données et d'extinction de bancs devraient permettre de réduire la consommation énergétique due à la gestion mémoire.

Bibliographie

[Appe 87] A. W. Appel. "Garbage Collection can be Faster than Stack Allocation". *Information Processing Letters*, Vol. 25, No. 4, pp. 275–279, 1987.

[Atta 01] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. "A Comparative Evaluation of Parallel Garbage Collectors". In : *Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, Cumberland Falls, KT, Aug. 2001.

[Baco 04] D. F. Bacon, P. Cheng, and D. Grove. "Garbage collection for embedded systems". In : *EMSOFT '04 : Proceedings of the fourth ACM international conference on Embedded software*, pp. 125–136, ACM Press, 2004.

[Blac 02] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. "Beltway : Getting Around Garbage Collection Gridlock". In : *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, pp. 153–164, ACM Press, Berlin, June 2002.

[Blac 04] S. M. Blackburn, P. Cheng, and K. S. McKinley. "Myths and realities : the performance impact of garbage collection". *SIGMETRICS Perform. Eval. Rev.*, Vol. 32, No. 1, pp. 25–36, 2004.

[Blan 98] B. Blanchet. "Escape Analysis : Correctness Proof, Implementation and Experimental Results". In : *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pp. 25–37, ACM Press, Montreal, June 1998.

[Bull 99] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. "A methodology for benchmarking Java Grande applications". In : *Proceedings of the ACM 1999 conference on Java Grande (JAVA '99)*, pp. 81–88, ACM Press, New York, NY, USA, 1999.

[Chen 02] G. Chen, R. Shetty, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. "Tuning Garbage Collection in an Embedded Java Environment". In : *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pp. 92–, IEEE Computer Society, Boston, MA, Feb. 2002.

[Chen 70] C. J. Cheney. "A Non-Recursive List Compacting Algorithm". *Communications of the ACM*, Vol. 13, No. 11, pp. 677–8, Nov. 1970.

[Cher 04] S. Cherem and R. Rugina. "Region Analysis and Transformation for Java Programs". In : A. Diwan, Ed., *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, ACM Press, Vancouver, Oct. 2004.

[Chin 04] W.-N. Chin, F. Craciun, and S. Qin. "Region Inference for an Object-Oriented Language". In : *Proceedings of SIGPLAN 2004 Conference on Programming Languages Design and Implementation*, ACM Press, Washington, DC, June 2004.

[Choi 99] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. "Escape Analysis for Java". In : *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, pp. 1–19, ACM Press, Denver, CO, Oct. 1999.

[Cohe 83] J. Cohen and A. Nicolau. "Comparison of Compacting Algorithms for Garbage Collection". *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 4, pp. 532–553, 1983.

[Coll 60] G. E. Collins. "A Method for Overlapping and Erasure of Lists". *Communications of the ACM*, Vol. 3, No. 12, pp. 655–657, Dec. 1960.

[Cour 05] A. Courbot, G. Grimaud, J.-J. Vandewalle, and D. Simplot. "Application-Driven Customization of an Embedded Java Virtual Machine". In : *proceedings of the Second International Symposium on Ubiquitous Intelligence and Smart Worlds (UISW2005)*, Nagasaki, Japan, December 2005.

[Cour 06] A. Courbot. *Spécialisation tardive de systèmes Java embarqués pour petits objets portables et sécurisés*. PhD thesis, Univ. Lille 1, France, september 2006.

[Dahl 68] O.-J. Dahl. *SIMULA 67 common base language*. 1968.

[Dete 02] M. Deters and R. Cytron. "Automated Discovery of Scoped Memory Regions for Real-Time Java". In : D. Detlefs, Ed., *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, pp. 25–35, ACM Press, Berlin, June 2002.

[Detl 02] D. Detlefs, Ed. *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM Press, Berlin, June 2002.

[Deut 76] L. P. Deutsch and D. G. Bobrow. "An Efficient Incremental Automatic Garbage Collector". *Communications of the ACM*, Vol. 19, No. 9, pp. 522–526, Sep. 1976.

[Diec 99] S. Dieckmann and U. Hölzle. "A Study of the Allocation Behaviour of the SPECjvm98 Java Benchmarks". In : *Proceedings of 13th European Conference on Object-Oriented Programming, ECOOP99*, pp. 92–115, Lisbon, July 1999.

[Dijk 78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. "On-The-Fly Garbage Collection : An exercise in Cooperation". *Communications of the ACM*, Vol. 21, No. 11, pp. 965–975, Nov. 1978.

[Edel 92] D. R. Edelson. "A Mark-and-Sweep Collector for C++". In : *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM Press, University of California, Santa Cruz, Jan. 1992.

[Edwa 03] L. A. Edwards. *Embedded System Design on a Shoestring*. Newnes, 2003.

[Engl 98] D. R. Engler. *The exokernel operating system architecture*. PhD thesis, Massachusetts Institute of Technology, October 1998.

[Feni 69] R. R. Fenichel and J. C. Yochelson. "A Lisp Garbage Collector for Virtual Memory Computer Systems". *Communications of the ACM*, Vol. 12, No. 11, pp. 611–612, Nov. 1969.

[Fitz 00] R. Fitzgerald and D. Tarditi. "The Case for Profile-Directed Selection of Garbage Collectors". In : T. Hosking, Ed., *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, ACM Press, Minneapolis, MN, Oct. 2000.

[Gajs 94] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and design of embedded systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[Gans 03] J. Ganssle and M. Barr. *Embedded Systems Dictionary*. CMPBooks, 2003.

[Gay 01] D. Gay and A. Aiken. "Language Support for Regions". In : *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM Press, Snowbird, Utah, June 2001.

[Gay 98] D. Gay and A. Aiken. "Memory Management with Explicit Regions". In : *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pp. 313–323, ACM Press, Montreal, June 1998.

[Gold 83] A. Goldberg and D. Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[Grim 05] G. Grimaud, K. Marquet, and D. Simplot-Ryl. "Optimization of the Root Set for Object-Oriented Memory Management of Smart Objects". In : *Proc. 11th ECOOP Workshop on Mobile Object Systems (MOS-11)*, Glasgow, UK, 2005.

[Grim 07] G. Grimaud, Y. Hodique, and I. Simplot-Ryl. "A verifiable Lightweight Escape Analysis Supporting Creational Design Patterns". In : *Proc. 2007 IEEE International Symposium on Ubisafe Computing (UbiSafe-07)*, 2007.

[Gros 02] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. "Region-Based Memory Management in Cyclone". In : *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, pp. 282–293, ACM Press, Berlin, June 2002.

[Gued 91] P. Guedes and D. P. Julin. "Object-oriented interfaces in the Mach 3.0 multiserver system". In : IEEE, Ed., *Proceedings. 1991 International Workshop on Object Orientation in Operating Systems, Palo Alto, CA, USA, October 17–18, 1991*, pp. 114–117, IEEE Press, 1991. IEEE catalog number 91TH0392-1. IEEE Computer Society Press order number 2265.

[Hart 88] P. H. Hartel. *Performance Analysis of Storage Management in Combinator Graph Reduction*. PhD thesis, Department of Computer Systems, University of Amsterdam, Amsterdam, 1988.

[Hirz 03] M. Hirzel, A. Diwan, and M. Hertz. "Connectivity-based Garbage Collection". In : *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM Press, Anaheim, CA, Nov. 2003.

[Hosk 00] T. Hosking, Ed. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, ACM Press, Minneapolis, MN, Oct. 2000.

[Hosk 93] A. L. Hosking and R. L. Hudson. "Remembered sets can also play cards". In : *ACM OOPSLA'93 Workshop on Memory Management and Garbage Collection*, Washington, DC, October 1993.

[Hugh 85] R. J. M. Hughes. "A Distributed Garbage Collection Algorithm". In : J.-P. Jouannaud, Ed., *Record of the 1985 Conference on Functional Programming and Computer Architecture*, pp. 256–272, Springer-Verlag, Nancy, France, Sep. 1985.

[JITS] "JITS : Java In The Small". http ://jits.gforge.inria.fr.

[Jone 96] R. E. Jones. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[Klei 98] G. Klein. "JFlex : the fast lexical analyzer generator for Java". 1998. http ://jflex.de.

[Knut 73] D. E. Knuth. *The Art of Computer Programming*, Chap. 2. Vol. I : Fundamental Algorithms, Addison-Wesley, second Ed., 1973.

[Kosh 05] J. Koshy and R. Pandey. "VMSTAR : synthesizing scalable runtime environments for sensor networks". In : *SenSys '05 : Proceedings of the 3rd international conference on Embedded networked sensor systems*, pp. 243–254, ACM Press, New York, NY, USA, 2005.

[Lawa 02] J. L. Lawall, G. Muller, and L. P. Barreto. "Capturing OS expertise in an event type system : the Bossa experience". In : *EW10 : Proceedings of the 10th workshop on ACM SIGOPS European workshop : beyond the PC*, pp. 54–61, ACM Press, New York, NY, USA, 2002.

[Lieb 83] H. Lieberman and C. E. Hewitt. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". *Communications of the ACM*, Vol. 26(6), pp. 419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[Lind 99] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Marq 05a] K. Marquet, A. Courbot, and G. Grimaud. "Ahead of Time Deployment in ROM of a Java-OS". In : *Proc. 2nd Int. Conf. on Embedded Software and System (ICESS 2005)*, pp. 63–70, Springer-Verlag, Berlin, Xi'an, China, 2005.

[Marq 05b] K. Marquet, G. Grimaud, and J.-J. Vandewalle. "Brevet français GEM_1755". 2005.

[Marq 07a] K. Marquet and G. Grimaud. "A DSL Approach for Object memory Management of Small Devices". In : *Proc. International Conference on Principles and Practice of Programming In Java (PPPJ 2007)*, Lisboa, Portugal, 2007.

[Marq 07b] K. Marquet and G. Grimaud. "A DSL Approach for Object memory Management of Small Devices". In : *Proc. Workshop on Programming Languages and Operating Systems (PLOS 2007)*, Stevenson, Washington, USA, 2007. À paraître.

[Marq 07c] K. Marquet and G. Grimaud. "Garbage Collection for Tiny Devices : A Complexity Study". In : *Proc. International Conference on Sensor Technologies and Applications (SENSORCOMM 2007)*, Valencia, Spain, 2007. À paraître.

[Marq 07d] K. Marquet and G. Grimaud. "An Object Memory management Solution for Small devices with Heterogeneous Memories". In : *Proc. 5th Workshop on Intelligent Solutions in Embedded Systems (WISES'07)*, pp. 228–238, Madrid, Spain, 2007.

[Marq 07e] K. Marquet and G. Grimaud. "An Object Memory management Solution for Small devices with Heterogeneous Memories". 2007. Poster session.

[McCa 60] J. McCarthy. "Recursive Functions of Symbolic Expressions and their Computation by Machine". *Communications of the ACM*, Vol. 3, pp. 184–195, 1960.

[Meri 00] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. "Devil : an IDL for hardware programming". In : *OSDI'00 : Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pp. 2–2, USENIX Association, Berkeley, CA, USA, 2000.

[Micr] S. Microsystems. "The CLDC HotSpot Implementation Virtual Machine".

[Micr 00] S. Microsystems. "J2ME Building Blocks for Mobile Devices". 2000.

[Micr 01] S. Microsystems. "The Java HotSpot Virtual Machine". 2001. Technical White Paper.

[Micr 03] S. Microsystems. "Java Card Virtual Machine Specification". 2003.

[Moha 84] K. A. Mohammed-Ali. *Object-Oriented Storage Management and Garbage Collection in Distributed Processing Systems*. PhD thesis, Royal Institute of Technology, Dept. of Computer Systems, Stockholm, Sweden, 1984.

[Qian 02] F. Qian and L. Hendren. "An adaptive, region-based allocator for java". In : D. Detlefs, Ed., *ISMM '02 : Proceedings of the 3rd international symposium on Memory management*, pp. 127–138, ACM Press, Berlin, June 2002.

[Rovn 85] P. Rovner. "On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language". Technical Report CSL–84–7, Xerox PARC, Palo Alto, CA, July 1985.

[Sala 07] G. Salagnac, C. Rippert, and S. Yovine. "Semi-Automatic Region-Based Memory Management for Real-Time Java Embedded Systems". In : *RTCSA'07 : Proc. 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, IEEE, 2007. to appear.

[Sans 92] P. M. Sansom. "Combining Copying and Compacting Garbage Collection". In : S. L. Peyton Jones, G. Hutton, and C. K. Hols, Eds., *Fourth Annual Glasgow Workshop on Functional Programming*, Springer-Verlag, 1992.

[Schu 03] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. L. Knudsen. "Compiling java for low-end embedded systems". In : *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pp. 42–50, ACM Press, 2003.

[Shay 03] N. Shaylor, D. N. Simon, and W. R. Bush. "A java virtual machine architecture for very small devices". In : *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pp. 34–41, ACM Press, 2003.

[Smit 91] A. J. Smith. "Second bibliography on Cache memories". *SIGARCH Comput. Archit. News*, Vol. 19, No. 4, pp. 154–182, 1991.

[Smit 98] F. Smith and G. Morrisett. "Comparing Mostly-Copying and Mark-Sweep Conservative Collection". In : R. Jones, Ed., *ISMM'98 Proceedings of the First International Symposium on Memory Management*, pp. 68–78, ACM Press, Vancouver, Oct. 1998.

[Soma 04] S. Soman, C. Krintz, and D. F. Bacon. "Dynamic selection of application-specific garbage collectors". In : *ISMM '04 : Proceedings of the 4th international symposium on Memory management*, pp. 49–60, ACM Press, New York, NY, USA, 2004.

[Spoo 05] D. Spoonhower, G. Blelloch, and R. Harper. "Using Page Residency to Balance Tradeoffs in Tracing Garbage Collection". In : J. Vitek, Ed., *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, ACM Press, Chicago, IL, June 2005.

[Sris 00] W. Srisa-an, J. M. Chang, and C.-T. D. Lo. "Do Generational Schemes Improve the Garbage Collection Efficiency ?". In : *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 58–63, IEEE Press, Austin, TX, Apr. 2000.

[Stef 94] D. Stefanović and J. E. B. Moss. "Characterisation of Object Behaviour in Standard ML of New Jersey". In : *Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming*, pp. 43–54, ACM Press, June 1994.

[Tard 00] D. Tarditi. "Compact Garbage Collection Tables". In : T. Hosking, Ed., *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, ACM Press, Minneapolis, MN, Oct. 2000.

[Unga 84] D. M. Ungar. "Generation Scavenging : A Non-Disruptive High Performance Storage Reclamation Algorithm". *ACM SIGPLAN Notices*, Vol. 19, No. 5, pp. 157–167, Apr. 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[Unga 87] D. Ungar and D. Patterson. "What price Smalltalk ?". *Computer*, Vol. 20, No. 1, pp. 67–74, 1987.

[Weiz 69] J. Weizenbaum. "Recovery of Reentrant List Structures in SLIP". *Communications of the ACM*, Vol. 12, No. 7, pp. 370–372, July 1969.

[Wils 92] P. R. Wilson. "Uniprocessor Garbage Collection Techniques". In : Y. Bekkers and J. Cohen, Eds., *Proceedings of International Workshop on Memory Management*, Springer-Verlag, University of Texas, USA, 16–18 Sep. 1992.

[Wise 79] D. S. Wise. "Morris' Garbage Compaction Algorithm Restores Reference Counts". *ACM Transactions on Programming Languages and Systems*, Vol. 1, pp. 115–120, July 1979.

[Zorn 90] B. Zorn. "Comparing Mark-and-Sweep and Stop-and-Copy Garbage Collection". In : *Conference Record of the 1990 ACM Symposium on Lisp and Functional Programming*, ACM Press, Nice, France, June 1990.