Étude d'un GC pour la technique de coloration bi-directionnelle

Nicolas Desnos - Encadrant : Roland Ducournau

14 juillet 2004

Table des matières

1 Introduction **6**
1.1 Présentation . . . 6
1.2 Structure du document . . . 7

2 La gestion de la mémoire **8**
2.1 La mémoire . . . 8
2.2 Structure d'un processus . . . 8
2.3 Mémoire virtuelle et mémoire physique . . . 9
2.4 L'allocateur . . . 11
2.5 La désallocation . . . 14
2.6 Le problème de la fragmentation . . . 15

3 Les GC compteurs de références **17**
3.1 Principe . . . 17

4 Les GC explorateurs **18**
4.1 Introduction . . . 18
4.2 Les GC explorateurs de bases . . . 19
4.3 Les GC exacts . . . 24
4.4 Les GC conservatifs (et semi conservatifs) . . . 25
4.5 Les GC générationnels . . . 34

5 Principe de la coloration **37**
5.1 Principe de l'héritage simple . . . 37
5.2 Principe de l'implémentation standard . . . 38
5.3 La coloration . . . 39

6 Étude d'un GC pour la coloration bi-directionnelle **41**
6.1 Introduction . . . 41
6.2 Présentation générale de nos architectures . . . 42
6.3 Architecture . . . 42
6.4 Stratégies où une page peut contenir le plus gros objet du programme . . . 46
6.5 Stratégies où le plus gros objet peut dépasser une page . . . 52
6.6 Conclusion . . . 55

7 Conclusion du mémoire **59**

A Le problème des faux objets **63**

B L'implémentation **66**

Table des figures

2.1 Espace d'adressage virtuel d'un processus 10
2.2 La mémoire 10
2.3 Liste implicite 12
2.4 Free list 12
2.5 Allocation linéaire 14
2.6 Fuite de mémoire 14
2.7 Pointeur fantôme 15
2.8 La fragmentation de la mémoire 16

4.1 *Mark and Sweep :* le marquage 20
4.2 *Mark and Sweep* : le balayage 20
4.3 Fusion pour un *Mark and Sweep* 20
4.4 Recopie. Copie des objets racines. 22
4.5 Recopie. Mise à jour des pointeurs et copie des autres objets vivants. 23
4.6 Recopie. Position finale 23
4.7 Problème du repérage d'un vrai objet 25
4.8 Problème de la manipulation des pointeurs 27
4.9 Problème de l'opérateur & de C++ 27
4.10 Problème de cast 28
4.11 Boehm. Les petits objets 29
4.12 Boehm. Les gros objets 30
4.13 Mostly-Copying Collector 32
4.14 génération 0 35
4.15 génération 1 35
4.16 génération 2 36
4.17 génération 3 36

5.1 Implémentation standard de C++ 38
5.2 La coloration 40

6.1 Accès au bloc suivant pour un langage classique 42
6.2 Accès au bloc suivant et problème pour la coloration 42
6.3 Architecture d'un objet 43
6.4 Architecture d'un bloc libre. 44
6.5 Implication de l'alignement sur le mot 44
6.6 Aspect d'une table de méthodes 44
6.7 Table de méthodes de tableaux 45
6.8 *free lists* 46
6.9 Architecture d'une page 1 46
6.10 Structure de données permettant de savoir si une page est allouée 48
6.11 Détermination des racines et exploration 49
6.12 Marquage 49
6.13 Modélisation du *sweep* 50

6.14 Architecture d'une page 2 51
6.15 La 1ère phase 52
6.16 La deuxième phase 53
6.17 Phase 3 et 4 54
6.18 Architecture des pages 3 54
6.19 MSMC2. Page d'en-tête d'un gros objet 55

A.1 Problème de la modification d'un entier suite à la phase de marquage 63
A.2 Filtrage d'un pointeur candidat 64
A.3 Problème lié à un pointeur d'un objet mort situé juste après l'en-tête de cet objet faisant parti d'un faux objet 65

B.1 Tests avec 4 pointeurs par objet 67
B.2 Tests avec 8 pointeurs par objet 67

Remerciements

Je tiens à remercier les personnes qui m'ont permis de mener à bien ce mémoire. À commencer par mon maître de stage Roland Ducournau que je remercie tant, par sa disponibilité et le temps qu'il m'a consacré, que par ses précieux conseils. Grâce à lui j'ai énormément appris dans le domaine de l'objet. Je remercie également les tézards qui se trouvaient dans mon entourage de travail et, en particulier, Toufik qui m'a aidé à chaque fois que j'avais besoin de ses conseils, surtout pour le Latex et l'utilisation de petits logiciels pratiques. Je remercie Jean avec qui j'ai pu discuter du domaine et pour ses conseils précieux. Je remercie également Luc, Sylvain, Fabien et Mohamed pour leur soutient moral. Je remercie mon père et ma mère qui ont passé du temps en re-lecture.

Chapitre 1

Introduction

1.1 Présentation

Généralités

Durant ces vingt dernières années, le C et le C++ ont été en grande majorité employés dans le développement de logiciels. Le C++ est un langage à objets basé sur le C permettant d'être très proche de la machine et dont le but est d'obtenir le meilleur rendement possible (rapidité d'exécution) en privilégiant au maximum l'optimisation et en laissant au développeur une grande liberté au niveau de la gestion de la mémoire. Or de nos jours, les projets sont de plus en plus importants et limités en temps de conception. L'utilisation de ces langages est alors coûteuse en temps de développement notamment à cause du problème de la désallocation manuelle qui provoque bon nombre d'erreurs difficiles à déceler. Un garbage collector[1] est un récupérateur automatique de la mémoire. Nous utiliserons par la suite GC pour le désigner. Il permet de simplifier la tâche du programmeur en lui rendant les programmes plus lisibles et en faisant baisser considérablement le temps de débogage. Le programmeur peut alors se consacrer à sa tâche de développement et être ainsi plus productif. Le GC se présente soit sous la forme d'une librairie que l'on ajoute au programme, ce qui est le cas des GC qui s'adaptent à un langage existant (C et C++), soit il fait partie intégrante du compilateur, c'est le cas des GC construits en même temps que le langage (Java). Côté études, nous nous intéresserons au GC explorateurs, les GC compteurs de références étant considérés comme obsolètes dans la littérature.

La coloration

La coloration dans les langages à objets est une technique qui permet de conserver toute l'efficacité de l'implémentation classique par tables de méthodes de l'héritage simple en héritage multiple. Elle a été décrite pour la première fois par Dixon en 1989 puis par Pugh et Weddel en 1990 et implémentée en Yafool par Roland Ducournau en 1991. Cette technique se caractérise par le fait que la référence à un objet, bien qu'invariante, peut être à l'intérieur d'un objet.

Notre contribution

Notre but est d'étudier des techniques de GC pour un langage objet à typage statique implémenté suivant la technique de la coloration. Or les techniques de GC existantes considèrent que la référence à un objet se trouve au début de l'objet ou bien est quelconque. Notre objectif est de modifier une de ces techniques en adaptant ses invariants à la coloration. Notre travail s'inscrit dans le cadre du développement du compilateur Eiffel basé entre autres sur la coloration [12].

[1] Connu également sous le nom de ramasse-miette ou de glaneur de cellules. L'abréviation est GC.

1.2 Structure du document

Les chapitres 2, 3 et 4 composent un état de l'art, le chapitre 5 présente la technique de la coloration et le chapitre 6 est consacré à notre contribution.

Chapitre 1 Introduction.

Chapitre 2 Ce chapitre est consacré tout d'abord à la mémoire. On y présente entre autre, la notion d'alignement et les notions de mémoire physique et de mémoire virtuelle. En suite, nous nous intéressons à l'allocateur en étudiant les techniques de gestion de mémoire et d'allocation, notamment la gestion par *free list* et l'allocation linéaire. Puis nous introduisons la désallocation et nous montrons les méfaits de la désallocation manuelle. Pour finir, on introduit la notion de fragmentation.

Chapitre 3 Le chapitre 3 est une brève présentation des GC compteurs de références. On y explique les raisons pour lesquelles ils ne sont pas performants.

Chapitre 4 Le chapitre 4 est consacré aux GC explorateurs. On commence par présenter les problématiques et notamment le problème de la localisation des pointeurs dans la pile et celui lié à l'arithmétique des pointeurs. Dans un second temps, nous présentons les techniques classiques (*Mark and Sweep*[2], Recopie et *Mark and Compact*[3]). Avant de présenter les techniques conservatives, on donne un aperçu des techniques utilisées par les GC exacts (tags et structure de données). On s'intéresse ensuite en détails aux techniques conservatives et semi-conservatives. Dans cette partie, nous décrivons le problème des faux objets et nous expliquons pourquoi ce problème impose des techniques à base de structures (pages avec en-tête) pour localiser avec certitude tous les objets. On y présente ensuite les techniques introduites par Boehm, et par Bartlett et notamment le MCC[4] dont nous allons nous inspirer. Pour finir nous exposons le principe des techniques générationnelles.

Chapitre 5 Dans ce chapitre, nous présentons la coloration et nous la comparons à l'implémentation classique en héritage simple et à l'implémentation standard de C++.

Chapitre 6 Le chapitre 6 est consacré à notre proposition d'un GC pour la coloration bidirectionnelle. Nous présentons des modèles basés principalement sur le MCC dans lesquelles nous choisissons d'utiliser une stratégie d'allocation basée sur les free lists suivant les intervalles de taille[5]. En conclusion, nous dressons 3 tableaux comparatifs concernant la méthode de Boehm, celle du MCC et la nôtre. Nous finissons en décrivant les perspectives futures.

Chapitre 7 Le chapitre 7 est la conclusion du mémoire.

Annexe A Cette annexe contient des explications sur le problème des faux objets.

Annexe B Cette annexe contient le code C de l'implémentation d'un de nos modèle ainsi qu'une expérimentation du GC.

[2] L'appellation francophone est le Marquage et Balayage.

[3] L'appellation francophone est le Marquage et Compactage.

[4] Le MCC (Mostly-Copying Collector) est une variante du mostly-copy de Bartlett créé par F. Smith et par G. Morrisett. Nous l'étudierons dans la section 4.4.4.

[5] Nous étudierons cette technique dans la section 2.4.2.

Chapitre 2

La gestion de la mémoire

2.1 La mémoire

On modélise généralement la mémoire par une suite de cases où chacune d'entre elles a pour taille 1 octet. Chaque case mémoire possède une adresse afin de pouvoir y accéder en lecture et en écriture. Son rôle est d'y stocker des informations. Une donnée dont la taille est supérieure ou égale à 1 octet sera désignée à l'aide d'un pointeur pointant sur la case mémoire correspondante à son premier octet. Pour des contraintes d'efficacité matérielle (bus de données), l'accès à une donnée de la taille d'un mot dont l'adresse est alignée sur le mot sera un accès direct. Alors que l'accès à une donnée de la taille d'un mot dont l'adresse est non alignée nécessitera 2 accès mémoire. Ainsi, certains langages se servent de l'alignement dans un souci d'optimisation d'accès à certains types primitifs. Par exemple le langage C aligne les entiers et les pointeurs sur le mot. On pourra trouver plus d'informations dans [15].

2.2 Structure d'un processus

Un processus en mémoire est constitué de plusieurs parties.

2.2.1 La pile

La pile contient les données temporaires. Son rôle est de gérer les appels de fonctions et les appels de méthodes. Lors d'un appel de fonction, on empile les paramètres, certains registres (SP, FP) et l'adresse de retour dans le code. Au cours de l'exécution les variables locales rencontrées sont également empilées. Puis une fois le code de la fonction exécuté, on récupère l'adresse de retour dans le code, on restitue l'environnement précédent l'appel de fonction[1] et on empile le résultat. Pour une méthode, on se ramène à un simple appel de fonction dans lequel on rajoute entre autre, un attribut caché *this*. Généralement les variables sont de type pointeur ou de type primitif mais il est possible dans certains langages d'y trouver des objets de type expansé (C++ par défaut et Eiffel avec *expanded*). La durée de vie d'une variable se trouvant dans la pile commence à la mise en pile et finit lorsqu'elle est dépilée.

2.2.2 Le tas

Le tas est une zone «libre» de la mémoire qui permet de stocker des objets dynamiquement. Un objet alloué dans le tas a une durée de vie qui va de sa création jusqu'à sa désallocation (manuelle ou automatique selon les langages). On pourra trouver dans [9] des explications sur la relation entre le mot clef d'allocation et les méta-classes dans un contexte complètement objet.

[1] On remet les valeurs de SP et de FP à leur valeur qui était la leur avant l'appel de fonction. On dépile tout ce qui avait été empilé pour l'appel.

2.2.3 Les données statiques

Ce sont les structures de données utilisées à l'exécution. Par exemple les tables de méthodes qui permettent de résoudre la liaison tardive, les variables globales ou encore des structures utilisées par le GC. On trouvera également des renseignements sur les variables statiques et les allocation-class de CLOS dans [9].

2.2.4 Le code

Le code est le code compilé du code source.

2.3 Mémoire virtuelle et mémoire physique

2.3.1 Le processeur

Le processeur est constitué d'un jeu d'instructions, d'un ensemble de registres et d'un accès à la mémoire. Son rôle est d'interpréter les instructions du segment de code du processus qui a la main. Pour des raisons d'efficacité, un processeur peut avoir de la mémoire cache implantée directement sur lui.

2.3.2 La RAM

La mémoire RAM est la mémoire (physique) du processeur. Elle est découpée en pages physiques de même taille. Son rôle est d'accueillir les processus. Elle est gérée par le système d'exploitation. Celui-ci doit en partie gérer l'espace mémoire en allouant de la mémoire pour les processus et en la libérant lorsque ceux-ci se terminent. Il doit également traiter le swapping.

2.3.3 Le swapping

Lorsque qu'il n'y a pas assez de mémoire pour accueillir la totalité des processus ou lorsqu'un processus demande trop de mémoire, le système d'exploitation met dans la zone du swap (sur disque) des données non utilisées. Lors d'une exécution il est possible qu'il y est des échanges entre le swap et la RAM.

Remarque 1 : L'accès à une donnée sur disque est très coûteux en temps.

2.3.4 Mémoire virtuelle et MMU

En fait, lorsqu'un programmeur manipule des adresses mémoire dans un programme, il ne manipule pas directement la mémoire physique (RAM), il manipule des adresses virtuelles (figure 2.1). Ceci présente deux intérêts : premièrement, un processus va pouvoir utiliser plus de mémoire que ce qui est disponible dans la RAM et deuxièmement, 2 processus différents vont pouvoir manipuler les mêmes adresses virtuelles. La mémoire d'un processus est donc composée d'adresses virtuelles et est découpée en pages virtuelles de même taille que les pages physiques. Ainsi, seules les pages nécessaires pour le processus seront chargées en mémoire physique, les autres resterons dans le swap. Pour faire le transfert d'adresses, le processeur comporte une unité mémoire (MMU) qui permet de manipuler des adresses virtuelles. Son rôle est de traduire les adresses virtuelles en adresses réelles. La figure 2.2 schématise cela.

Remarque 2 : L'accès à des données se trouvant sur une même page physique sera plus rapide que si elles sont éparpillées sur plusieurs pages physiques.

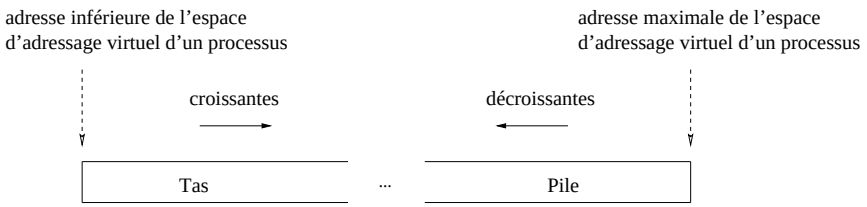

FIG. 2.1 – Espace d'adressage virtuel d'un processus

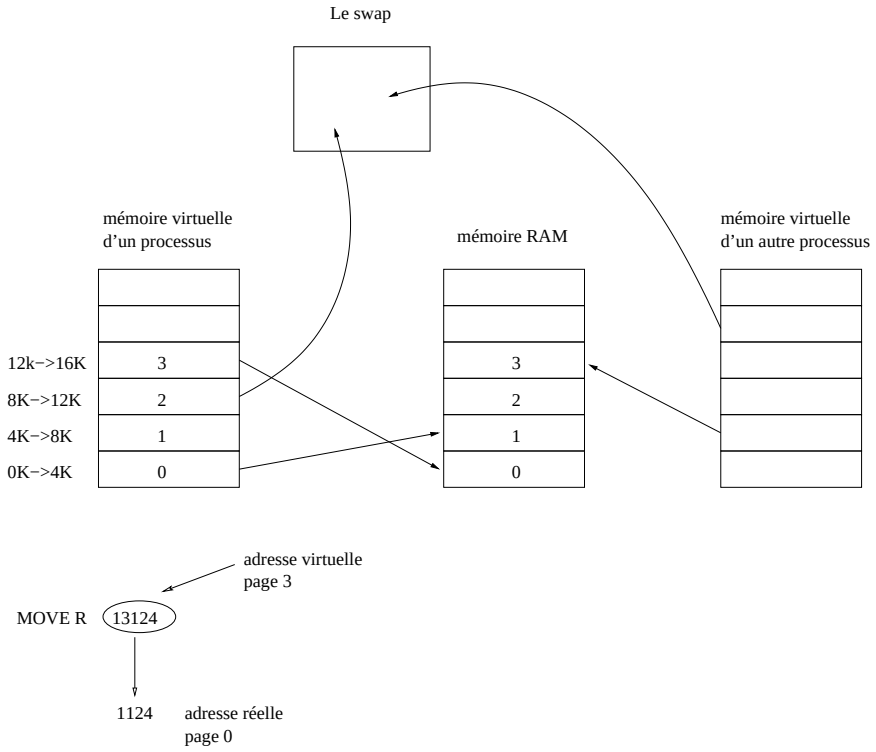

FIG. 2.2 – La mémoire

2.4 L'allocateur

Définition 1 : Dans un cadre de gestion de la mémoire automatique, on appelle mutateur le programme qui s'exécute. Il peut modifier des pointeurs (affectations) et faire des demandes d'allocation à l'allocateur.

Définition 2 : Un objet est une zone de la mémoire contenant des données.

Définition 3 : Un objet vivant est un objet accessible par le mutateur de manière directe ou indirecte.

Définition 4 : Un objet mort est un objet qui n'est plus accessible par le mutateur.

Principe

Le rôle de l'allocateur est de répondre à une requête faite par le mutateur en trouvant une zone de mémoire contiguë et disponible[2] pour accueillir l'objet. Le but est bien évidemment de réaliser l'allocation en temps quasi constant. Pour ce faire nous allons étudier diverses stratégies d'organisation de la mémoire.

2.4.1 Gestion de la mémoire par tableau de bits

On divise la mémoire en unités d'allocation. A chaque unité, on fait correspondre un bit dans le tableau de bits. Si le bit est à 0, cela signifie que l'unité de mémoire est libre et si il est à 1 c'est qu'elle est occupée. Côté *overhead* statique, ce tableau est défini statiquement et de taille fixe. L'espace mémoire consommé dépend de la taille de l'unité d'allocation. En effet, plus celle-ci est petite et plus le tableau sera grand. Quant à l'*overhead* dynamique, il dépend de la taille de l'unité d'allocation mémoire. Si l'unité est grande, alors dans le cas où l'on utilise beaucoup de petits objets, on aura des pertes de mémoire. Cependant, la recherche d'une zone libre pour accueillir un objet est loin d'être réalisée en temps constant. En effet, c'est une recherche séquentielle du tableau de bits à la recherche d'une suite de zéros adéquat. Il y a un donc un dilemme. Si on prend une grande unité d'allocation, alors on réduit la taille du tableau de bits, mais on peut perdre beaucoup de place mémoire dans le tas. Mais ce qui rend peu envisageable cette technique est surtout le temps potentiellement long lors d'une allocation d'un objet notamment si celui-ci est gros.

2.4.2 Gestion de la mémoire par listes chaînées

L'idée est d'éviter à l'allocateur d'avoir à faire un parcours séquentiel pour trouver une zone libre. Pour cela nous allons définir la notion de bloc.

Définition 5 : Un bloc est une zone de la mémoire possédant une en-tête.

Définition 6 : Une en-tête est une structure qui permet au gestionaire de la mémoire de gérer le bloc.

Définition 7 : Un bloc libre est un bloc qui est disponible pour l'allocateur.

Définition 8 : Un bloc occupé est un bloc que l'allocateur n'a pas le droit d'utiliser.

[2] Le choix de cette zone mémoire doit être la meilleure possible. C'est ce que nous étudions par la suite.

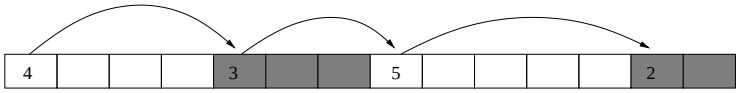

FIG. 2.3 – Liste implicite

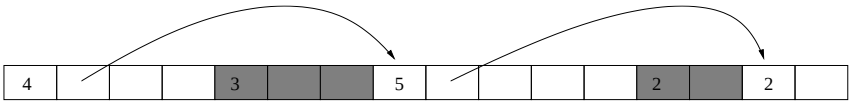

FIG. 2.4 – Free list

Dans ce modèle, la mémoire est une succession de blocs libres et de blocs occupés. Nous allons voir les diverses stratégies qui permettent de trouver le bon bloc libre et de répondre ainsi à la requête du mutateur.

liste implicite

L'en-tête d'un bloc contient sa taille, et son statut (ie 1 bit pour savoir s'il est libre ou occupé). La liste implicite qui est obtenue grâce à la taille dans l'en-tête administrative permet de passer d'un bloc à un autre. Cependant, une telle implémentation n'est pas réaliste. En effet, une allocation quelconque d'un objet va nécessiter un passage par des blocs occupés ce qui est inacceptable. Dans la figure 2.3 il y a 2 blocs libres (en blancs) et 2 blocs occupés (en gris). Les flèches sont des flèches implicites et représentent le passage d'un bloc à un autre grâce à la taille du bloc courant.

Remarque 3 : Dans un langage où un objet pointe sur une table de méthodes, on peut stocker l'information de la taille dans la table de méthodes et ainsi éviter un *overhead* dynamique de 1 mot mémoire supplémentaire par objet.

méthode de base avec une *free list*

L'idée est de remplacer le parcours de la méthode précédente par un parcours sur une liste liant tous les blocs libres (une *free list*). Cependant l'allocation est toujours loin d'être réalisée en temps constant. En effet, le temps de parcours de la *free list* dépend de la taille de l'objet que l'on veut créer, de la répartition des blocs libres de la *free list* et de sa longueur (ie dans le pire des cas il faudra parcourir toute la *free list*). Dans la figure 2.4, chaque bloc libre possède un pointeur que l'on a rajouté après l'en-tête administrative. Les flèches représentent donc ici des pointeurs.

Méthode avec plusieurs *free lists*

Cette méthode est abordée dans [13]. C'est une optimisation du modèle précédent qui consiste à diviser la *free list* en plusieurs *free lists* par intervalle de taille. On utilise les puissances de 2 et on prend comme intervalle, 2^n et $2^{n+1} - 1$ où n est le nombre de *free lists*. La première *free list* couvre donc l'intervalle $[2^0; 2^1 - 1]$, la seconde couvre l'intervalle $[2^1; 2^2 - 1]$, la ième couvre l'intervalle $[2^i; 2^{i+1} - 1]$. Lorsqu'une requête est faite à l'allocateur, celui-ci calcule le premier intervalle pouvant accueillir l'objet et sélectionne la bonne *free list*. Si elle est non vide et qu'il existe un emplacement suffisamment grand, on réalise l'allocation, si non, on réitère la chercher dans la *free list* suivante et ainsi de suite.

Définition 9 : La fragmentation[3] de la mémoire est la répartition de l'espace libre en plusieurs zones non contiguës.

Stratégies d'allocations

Lorsqu'un objet doit être alloué, l'allocateur dispose en général de trous suffisamment grands pour le contenir. La différence entre les diverses stratégies d'allocations se situe dans le choix du trou. Nous allons ici en faire une courte présentation. On pourra en avoir un bon aperçu dans [22, 17]. Ce sont des algorithmes d'allocation **qui prennent en entrée une *free list*.**

Le First Fit

C'est la stratégie d'allocation la plus simple. L'algorithme consiste à parcourir linéairement la *free list* à la recherche du premier emplacement libre convenable. Cette technique pose des problèmes : le bloc trouvé peut posséder une taille vraiment supérieure à celle nécessaire. Dans ce cas la fragmentation de cet espace mémoire peut condamner à terme l'allocation d'un gros objet. Ce qui caractérise cet algorithme est l'apparition d'une fragmentation de petite taille en début de *free list*. Une optimisation consiste à supprimer de la *free list* les trous de petites tailles que l'on considère inutilisables.

Le Next Fit

Cet algorithme est une variation de l'algorithme First Fit. Il consiste à ne pas parcourir la *free list* de la même façon. On va boucler la liste circulairement. L'allocation consiste à partir du trou dans lequel a été réalisé la dernière allocation.

Le Best Fit

Cet algorithme va chercher à optimiser First Fit. La stratégie consiste à déterminer le plus petit trou possible pour y mettre l'objet. Le résultat attendu étant de maximiser la gestion de l'espace mémoire libre. En cherchant la meilleure zone libre, on évite de fragmenter une zone mémoire plus grande qui aurait pu nous servir par la suite à accueillir un gros objet. Cependant cet algorithme à le défaut suivant : chercher le trou optimal implique un parcours en O(n) (si n est la taille de la liste). Un autre problème est qu'il peut générer de la petite fragmentation dans le cas où un objet est alloué dans un emplacement juste supérieur.

Le Worst Fit

[20] précise que certaines expériences ont montré que le First Fit produit moins de fragmentation que le Best Fit. Le Worst Fit consiste à déterminer le trou le plus grand disponible pour que la zone mémoire restante soit la plus importante possible.

Remarque 4 : **Nous verrons que la gestion de la mémoire par *free list* est surtout destinée au *Mark and Sweep* (section 4.2.1).**

2.4.3 Gestion de la mémoire par allocation linéaire

L'allocation linéaire est l'allocation la plus rapide. Elle consiste à utiliser un pointeur d'allocation. A chaque fois qu'une allocation est réalisée, on incrémente le pointeur d'allocation de la taille de l'objet fraîchement alloué (voir figure 2.5). L'avantage de cette technique est qu'elle ne nécessite pas de mises à jour comme dans la gestion par *free list*.

[3] Nous la discutons à la section 2.6.

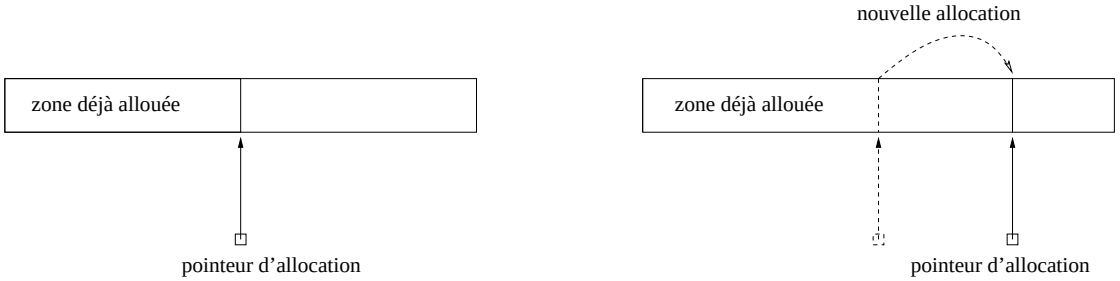

FIG. 2.5 – Allocation linéaire

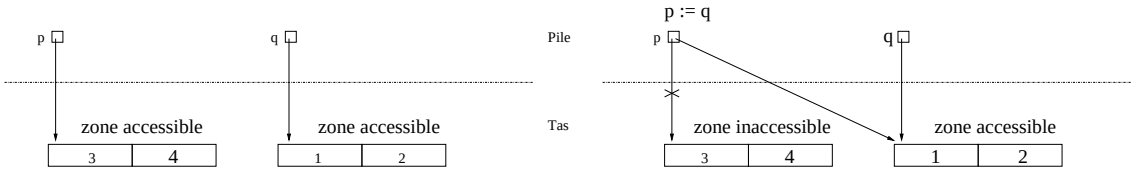

FIG. 2.6 – Fuite de mémoire

Remarque 5 : **Nous verrons que cette technique est destinée aux GC par Recopie et au Mark and Compact (section 4.2.2 et 4.2.3).**

2.5 La désallocation

La désallocation d'un objet consiste à rendre l'espace qu'il occupe à l'allocateur. Selon les langages de programmation, elle peut se faire manuellement ou automatiquement. Nous allons d'abord voir les défauts d'une désallocation manuelle et voir en suite les biens-faits d'un GC.

2.5.1 Gestion de la désallocation par le programmeur

Cette solution est abordée principalement dans des langages non objets comme Pascal, C, Ada, Modula-2, PL/I. C'est également le choix de C++. Dans cette configuration, la gestion de la désallocation de la mémoire est laissée entièrement à la responsabilité du développeur. Mais cette tâche nécessite une très bonne vue d'ensemble de l'exécution du programme. La difficulté à laquelle peut se heurter le développeur est l'incapacité à prévoir les durées de vie de certains objets dans des structures de données allouées. Les 3 erreurs classiques sont la fuite de mémoire, le pointeur fantôme et la demande de désallocation répété de la même zone mémoire.

la fuite de mémoire

La fuite de mémoire (*memory leak*) consiste à oublier de rendre au système, une zone mémoire, tout en perdant tout pointeur la référençant. La conséquence est que cette zone mémoire ne pourra plus jamais être réutilisée par l'allocateur. Si cette erreur est répétée un grand nombre de fois, on peut aboutir à une saturation de la mémoire. La figure 2.6 montre un exemple de fuite de mémoire.

Le pointeur fantôme

Un pointeur fantôme (*dangling pointer*) apparaît lorsqu'on restitue une zone mémoire alors qu'il y a encore des références sur cette zone. Une telle référence est un pointeur fantôme. Cette erreur est sournoise et peut être très difficile à déceler. La pire des situations arrive lorsque l'objet qui a été alloué sur cette zone mémoire est compatible avec l'ancien objet (par exemple même type). Dans ce cas, il n'y aura pas d'erreurs lors de l'exécution mais un comportement anormal très difficile à résoudre. Signalons que le débogage d'un pointeur fantôme est délicat car l'erreur peut

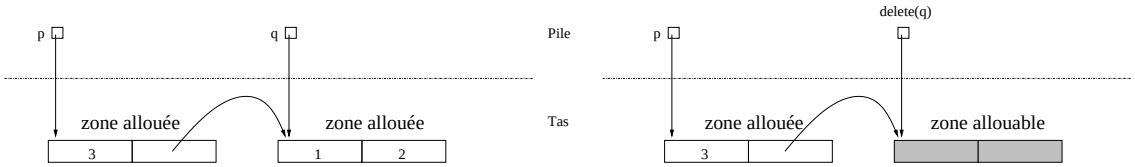

FIG. 2.7 – Pointeur fantôme

être difficile à reproduire. En effet, il faut avoir la même séquence d'allocation et de désallocation afin d'arriver à la même erreur. La figure 2.7 montre un exemple de pointeur fantôme.

Demande répétée de désallocation d'une même zone mémoire

Dans le cas de C et de C++, le comportement est indéterminé.

Discussion

Un des avantages qu'on pourrait citer est que le développeur est maître de ce qu'il fait et dans le cas où les désallocations sont parfaites, le programme sera plus rapide qu'avec un GC. Cependant les problèmes de pointeurs fantômes et de fuites de mémoire s'avèrent être très délicats à gérer sur des gros projets. Ces problèmes impliquent généralement de nombreuses heures de débogages ce qui limitent le temps passé par le développeur sur sa tâche principale.

Remarque 6 : On pourra trouver une discussion plus profonde sur les méfaits de la désallocation manuelle dans [21].

2.5.2 Gestion de la désallocation par le GC

Le rôle d'un GC est de soulager le programmeur en gérant automatiquement la désallocation et en apportant une sûreté dans la gestion mémoire. Boehm précise dans [6] que l'on peut gagner jusqu'à 40% du temps de développement sur un gros projet dans lequel on manipule des structures de données complexes. Côté performance, les spécialistes du domaine considèrent qu'un GC qui prend moins de 10% du temps d'exécution est un GC bien conçu. En ce qui concerne les techniques de GC, elles reposent toutes en partie sur la règle et sur l'invariant suivant :

Règle 1 : Un GC ne doit en aucun cas supprimer d'objets encore vivants.

Invariant 1 : Un objet qui n'est plus référencé à un instant t de l'exécution ne le sera pas non plus à un instant $t + dt$ quelque soit dt.

Remarque 7 : Un GC ne va s'occuper que de la désallocation des objets alloués dans le tas puisque ceux de la pile (objets expansés) s'il y en a seront automatiquement détruits lors du dépilement.

Remarque 8 : Un objet qui n'est plus référencé est inutile.

2.6 Le problème de la fragmentation

On rappelle que la fragmentation de la mémoire est la répartition de l'espace libre en plusieurs zones non contiguës. La figure 2.8 expose cela. Les zones libres étant en blanc. Dans une gestion par *free list*, le problème qu'elle pose est que l'allocateur peut être dans l'incapacité à réutiliser de la mémoire libérée. En effet, il est possible que celui-ci ne trouve pas assez de place pour y allouer un objet alors que le total de la zone libre disponible est largement suffisante. Dans ce cas

Fig. 2.8 – La fragmentation de la mémoire

le meilleur moyen de la combattre est de réaliser la fusion des trous contigus. En revanche, dans une gestion par allocation linéaire, il n'y a pas de problèmes de fragmentation.

Chapitre 3

Les GC compteurs de références

3.1 Principe

3.1.1 Fonctionnement

L'algorithme de comptage de références est l'un des plus anciens en ce qui concerne la gestion automatique de la mémoire. Pour mémoire il date des années 1960 avec Gelernter et Collins. Son principe est simple est intuitif. Chaque objet possède un compteur qui va déterminer le nombre de références pointant vers cet objet. Lors de la création d'un objet, son compteur est initialisé à 0. Dès qu'il y a une affectation d'une variable référence vers cet objet, on incrémente son compteur, et dès qu'il y a un déréférencement, on le décrémente. La suppression de l'objet s'effectue par le GC lorsque le compteur est égal à zéro.

3.1.2 Avantages

1. C'est un GC naturellement incrémental. En effet le GC ramasse un objet mort dès que celui-ci le devient. Le fait d'être incrémental a pour avantage de ne pas provoquer de blocage durant l'exécution. Ce qui fait qu'un tel GC peut être avantageux pour le temps réel.

3.1.3 Inconvénients

1. Ce GC ne gère pas les cycles.
2. Le temps d'exécution global du programme est ralenti. En effet, il y a un sur-coût sur toutes les manipulations de références par le médiateur (décrémentation, incrémentation du compteur, test si le compteur est égal à zéro) même s'il n'y a pas suppression d'objet. Par exemple, une simple affectation peut impliquer 2 modifications de compteur. Nous pouvons noter également que la mise à jour systématique des compteurs peut en plus être ralentie par le fait que les 2 compteurs à mettre à jour ne se trouvent pas sur la même page physique en mémoire RAM, ou bien pire que l'un des compteur soit dans le Swap.
3. Il y a un *overhead* dynamique d'un mot mémoire par objet.

Remarque 1 : [20, 22, 21] propose des alternatives, comme rajouter un GC explorateur, mais ces méthodes restent peu convaincantes.

Chapitre 4

Les GC explorateurs

4.1 Introduction

Définition 1 : Une référence exacte est une valeur dont le GC est certain que c'est une référence.

Définition 2 : Une référence ambiguë est une valeur qui est potentiellement une référence, mais le GC ne peut le prouver.

Définition 3 : Une racine est un pointeur sur un objet du tas qui se trouve dans la pile, dans les registres ou dans les données statiques (variables globales ou variables déclarées statiques).

Définition 4 : Une racine ambiguë est une racine contenant une référence ambiguë.

Définition 5 : Une racine exacte est une racine contenant une référence exacte.

4.1.1 Présentation

On a vu que les compteurs de références sont naturellement incrémentaux et qu'ils permettent ainsi de faire du temps réel. Cependant, un de leur gros défaut est qu'ils ralentissent l'exécution globale de l'application ce qui est inacceptable dans certaines applications. Les GC explorateurs vont permettre d'y remédier. Leur fonctionnement reposent tous sur le même principe : ils utilisent une phase d'exploration du graphe d'objets afin de détecter les objets vivants. Cette exploration se fait de la façon suivante : les objets en mémoire forment un graphe orienté d'objets. L'idée est de partir des racines et d'atteindre récursivement tous les objets atteignables. Le sous graphe obtenu est un graphe dans lequel les noeuds sont des objets vivants et les arcs des pointeurs. Nous allons exposer le principe du *Mark and Sweep*, du GC par Recopie et du *Mark and Compact* qui sont les GC explorateurs de bases. Mais avant, nous allons définir la règle que doit respecter tout GC explorateur afin de garantir la règle 1 du chapitre 2 :

Règle 1 : Tout objet vivant est traversé par le GC.

4.1.2 Problématiques

Les pauses

Les GC explorateurs de bases sont des GC asynchrones, c'est-à-dire qu'ils ne sont pas en étroite collaboration avec le mutateur. Un tel GC ne se déclenche que lorsque le besoin de mémoire se fait sentir. Par exemple lorsqu'une demande d'allocation faite par le mutateur à l'allocateur échoue. Le GC se déclenche et interrompt le mutateur. Un GC explorateur ne va pas ralentir l'exécution

générale du programme, cependant, il peut provoquer des pauses dues à l'interruption du mutateur. Il existe des GC qui essayent de lutter contre cela : Les GC générationnels et les GC explorateurs incrémentaux. Nous n'étudierons pas les GC incrémentaux, mais on pourra en avoir un bon aperçu dans [21, 20].

Détermination des racines et localisation des pointeurs à l'intérieur d'un objet.

Pour pouvoir réaliser l'exploration, on a besoin de connaître la localisation des pointeurs en particulier dans la pile et dans les objets. A partir de là, nous pouvons classer les GC explorateurs en 3 grandes familles. Premièrement, les GC exacts. Ces GC ont le moyen de localiser tous les pointeurs et donc de disposer de références exactes. Deuxièmement, les GC conservatifs. Ces GC n'ont aucun moyen de les localiser avec certitude et donc disposent de références ambiguës. Troisièmement, les GC semi-conservatifs. Ces GC ont juste le moyen de localiser les pointeurs à l'intérieur des objets. Nous verrons en particulier dans la partie 4.4 ce qu'est un parcours conservatif et les raisons pour lesquelles ils sont parfois nécessaires.

Le problème de l'arithmétique des pointeurs.

Ce problème concerne les GC qui sont destinés à des langages qui permettent la manipulation directe des pointeurs et des casts à la C.

4.2 Les GC explorateurs de bases

4.2.1 Le Mark and Sweep

Ce GC fonctionne principalement dans un modèle de gestion de mémoire par blocs et par *free list*[1]. Il s'effectue en 2 phases. Premièrement, les objets sont explorés récursivement à partir des racines. C'est la phase dite de marquage. Durant cette phase, chaque objet atteint est marqué. Il est judicieux d'utiliser un bit de marquage en guise de marqueur afin de ne pas avoir d'*overhead*. Puis dans une seconde phase dite phase de balayage (*sweep*), le collecteur balaye séquentiellement le tas en récupérant tous les objets morts (ie les objets non marqués) qu'il transforme en blocs libres. Il est à noter qu'on pourra effectuer d'autres traitements lors de la phase de balayage comme par exemple remettre à zéro chaque marqueur (pour la prochaine collecte), regrouper les blocs disponibles adjacents (fusion) et créer les free lists en y mettant les blocs libres rencontrés ou ceux fraîchement créés. Le premier schéma de la figure 4.1 montre un exemple d'état de la mémoire avant le déclenchement du GC. Le deuxième schéma de la figure 4.1 montre la phase de marquage (les objets assombris étant les objets atteints par le GC). La figure 4.2 montre la phase de sweep avec la création d'une *free list*.

Gestion des blocs libres et fusion

Dans un *Mark and Sweep*, comme le montre la figure 4.3, la fusion s'effectue simplement lors de la phase de *sweep* en utilisant un simple pointeur d'analyse.

Avantages

1. Le *Mark and Sweep* n'a pas de problèmes de cycles.
2. Il autorise les GC conservatifs (C, C++).
3. Il ne ralentit pas l'exécution globale du système du fait qu'il n'est pas en étroite collaboration avec le mutateur.
4. Il n'a pas d'*overhead* puisqu'il utilise un bit de l'objet.

[1] On rappelle que cette technique a été présentée à la section 2.4.2

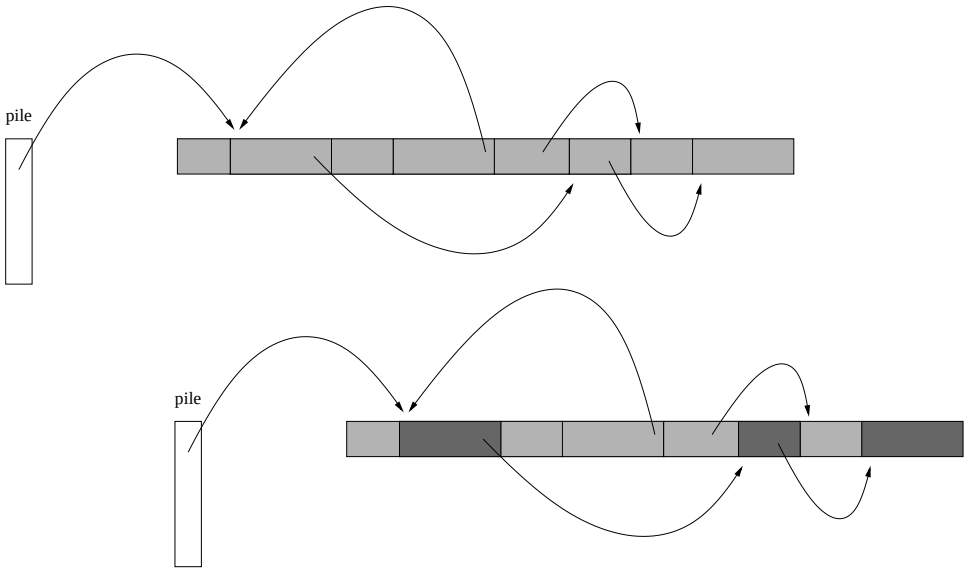

FIG. 4.1 – *Mark and Sweep :* le marquage

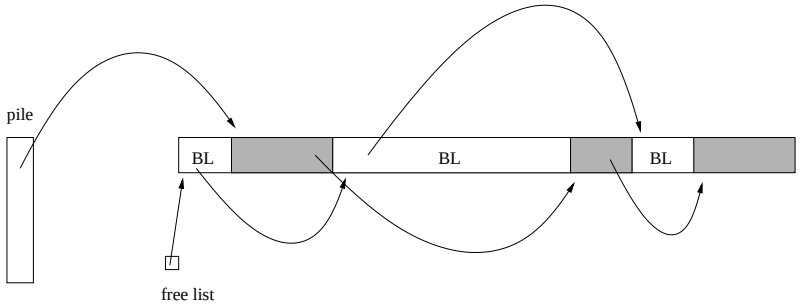

FIG. 4.2 – *Mark and Sweep* : le balayage

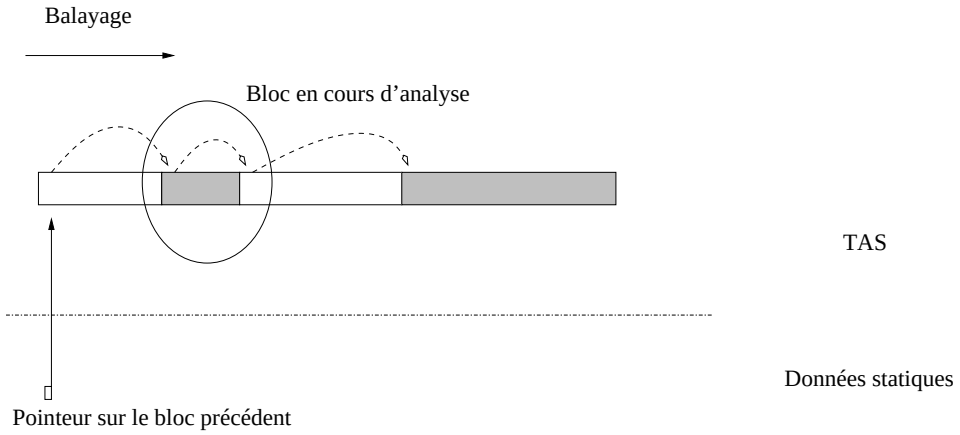

FIG. 4.3 – Fusion pour un *Mark and Sweep*

Inconvénients

1. La phase de balayage nécessite un parcours du tas qui peut se révéler être long et entraîner, dans certains cas, de courtes pauses lors de l'exécution[2]. Nous verrons par la suite, qu'une solution consiste à restreindre la collecte à certaines zones du tas (GC générationnels) mais que d'autres problèmes apparaissent.
2. A l'issu des 2 phases, le *Mark and Sweep* laisse le tas fragmenté.

4.2.2 La Recopie

Ce GC fonctionne dans un modèle de gestion mémoire par allocation linéaire[3]. La phase de marquage du *Mark and Sweep* ne touche que les blocs accessibles (ie les objets vivants). Par contre, le problème du *Mark and Sweep* est que la phase de balayage implique un parcours de tous les blocs (même des blocs libres) afin de déterminer ceux qu'on doit récupérer (ie ceux qui ne sont pas marqués). Le *Mark and Sweep* provoque de la fragmentation. L'objectif de la Recopie est de la faire disparaître.

Principe

Le principe de la Recopie consiste à doubler l'espace mémoire et à le diviser en 2 sous-espaces de même taille. L'un va servir d'espace d'allocation (espace actif) et l'autre va servir à faire la copie (espace mort ou espace destination). Lorsqu'il n'y a plus assez de place dans l'espace actif pour satisfaire une demande d'allocation, un processus de ramasse miettes se déclenche. Le principe est simple, on fait une exploration dans le genre du marquage dans le *Mark and Sweep* et chaque fois qu'un objet est rencontré on le copie dans la zone destination. La zone destination se comportant comme une pile. A la fin du processus, tous les objets vivants qui se trouvaient dans la zone active avant le déclenchement du GC se retrouvent dans l'espace destination les uns à la suite des autres. La zone d'allocation se trouvant alors juste après. La zone active devient zone morte et la zone morte devient zone active. Bien entendu, il faut faire une mise à jour des pointeurs dans les objets. Nous allons voir comment cela est réalisé dans l'algorithme de Cheney.

Algorithme de Cheney

L'algorithme de Cheney est un algorithme non récursif dans lequel les objets sont recopiés par un parcours en largeur. Le premier schéma de la figure 4.4 montre l'état de la mémoire avant le déclenchement du GC par recopie.

- Étape 1. L'opération de copie entre les deux espaces commence par la copie des blocs de la zone active qui sont repérés par des pointeurs de l'ensemble racine. L'original est marqué «copié» (noté C dans les figures 4.4 et 4.5) et un pointeur sur la copie y est inséré[4] (ie le contenu de l'original pouvant être détruit). Puis, on met à jour les racines qui pointent désormais vers les copies. Dans le second schéma de la figure 4.4, l'étape 1 est terminée. Il est à signaler qu'aucun pointeur n'est mis à jour à l'intérieur des copies.
- Étape 2 (figure 4.5). L'espace destination est alors vu comme une file[5]. On utilise un pointeur d'analyse qui va la parcourir. Soit Ba un bloc de la file. Considérons que p pointe sur un champ de Ba de type pointeur (soit ptr ce champ) qui, lui, pointe sur un bloc Bd de l'espace de départ. Il y a 2 possibilités : soit Bd n'est pas marqué «copié» et dans ce cas, il faut le copier (l'insérer dans la file), mettre un pointeur dans Bd vers sa copie et faire la mise à jour de ptr, soit Bd est marqué et dans ce cas, on récupère le pointeur qui pointe sur sa copie pour mettre à jour ptr. Le processus s'arrête (figure 4.6) une fois que le pointeur d'analyse a passé tous les éléments de la file.

[2] Son temps d'exécution étant proportionnel au nombre de blocs.
[3] On rappelle que cette technique a été présentée à la section 2.4.3.
[4] Ce pointeur est aussi nommé *forward pointer*.
[5] C'est une file implicite qui est également nommée *cheney queue*.

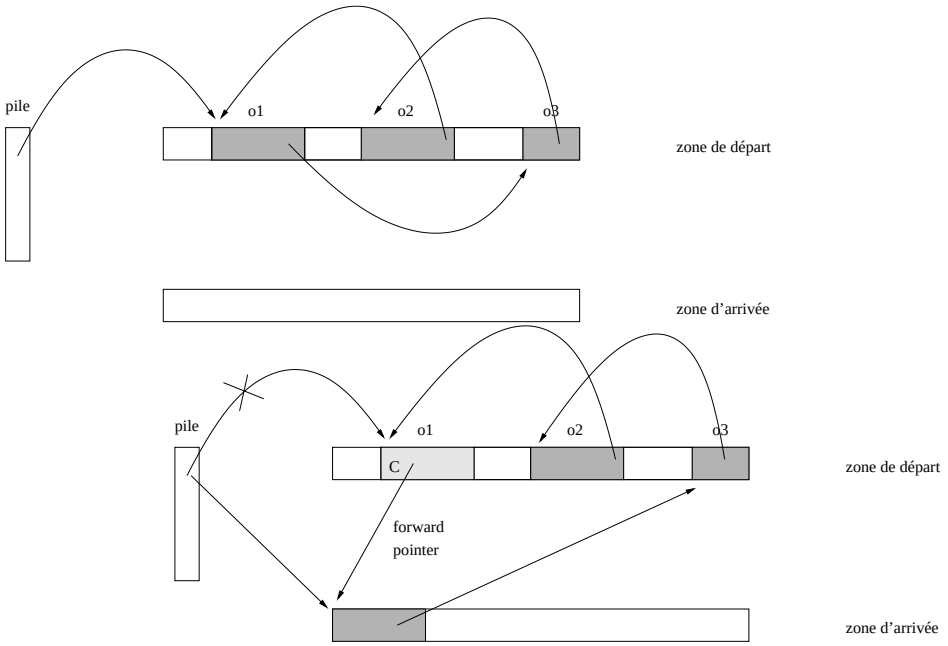

FIG. 4.4 – Recopie. Copie des objets racines.

Avantages

1. Il n'y a plus de fragmentation. (ie le tas est compact).
2. Plus de problèmes de cycles.
3. Pas d'*overhead*.
4. Son temps d'exécution n'est plus proportionnel au nombre de blocs du tas mais à la taille et au nombre des objets référencés (accessibles par le mutateur).
5. La gestion des espaces mémoires libres par *free list* n'a plus lieu d'être (puisque l'espace libre se trouve juste après les objets vivant contigus que l'on vient de copier) ce qui implique une allocation très rapide.
6. Vérifier que l'allocation est correcte est très rapide. En effet, cela consiste juste à faire une comparaison de pointeurs entre le sommet courant de l'espace actif et le sommet maximum de cet espace.
7. Ce GC de copie est très performant en ce qui concerne le traitement des objets jeunes. En effet, on constate que plus un objet est jeune et plus il a de chances d'avoir une durée de vie très courte. Ainsi un GC de copie qui ne s'applique qu'à une jeune génération aura peu d'objets à copier puisque la plupart seront inactifs. De ce fait, certains GC générationnels utilisent une combinaison de l'algorithme de copie pour les objets jeunes et un *Mark and Sweep* pour les vieux objets.

Inconvénients

1. Toutes les références manipulées comme des pointeurs doivent être des référence exactes. En effet, si la référence n'est pas exacte on risque de modifier une valeur qui n'est pas un pointeur lors de la mise à jour.
2. Problème de la recopie d'objet de longue vie que l'on va recopier a chaque fois.

Remarque 1 : Le fait qu'il faille deux fois plus de mémoire que pour le *Mark and Sweep* n'est pas un vrai problème puisque la mémoire utilisée est de la mémoire virtuelle. Le problème qui peut se

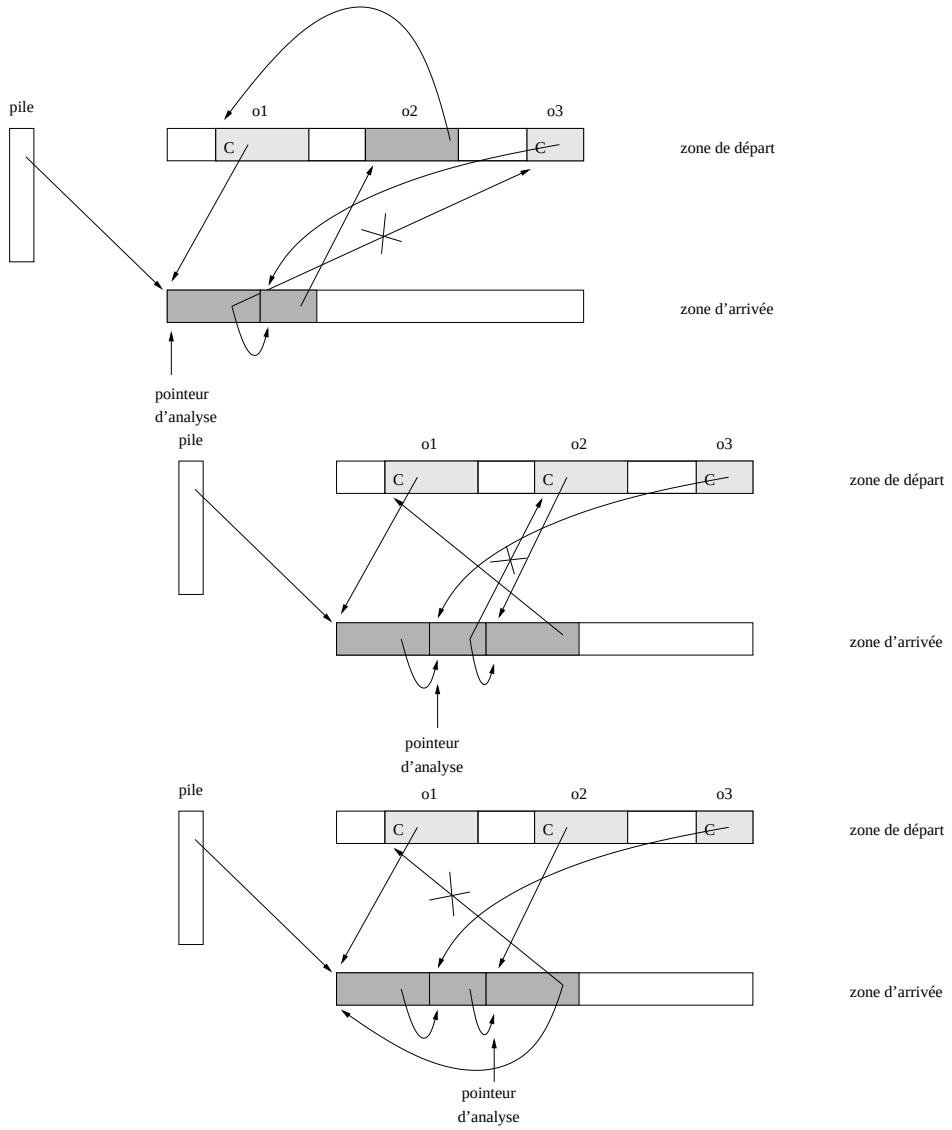

FIG. 4.5 – Recopie. Mise à jour des pointeurs et copie des autres objets vivants.

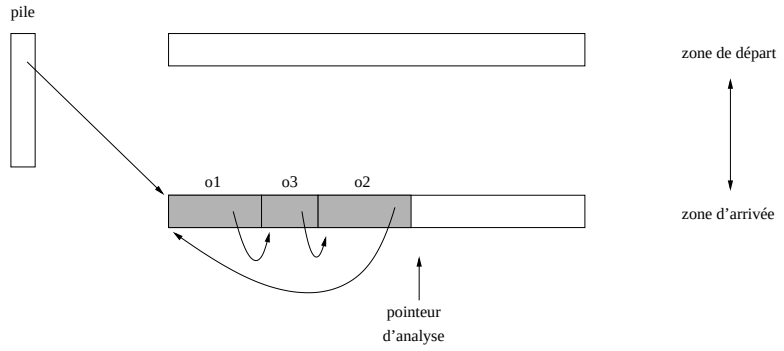

FIG. 4.6 – Recopie. Position finale

poser et que le système peut swapper et perturber l'exécution du GC. Si le GC est paramétrable, il suffit de choisir une taille en fonction de la machine que l'on utilise.

4.2.3 Le Mark and Compact

Comme le GC de Recopie, il fonctionne dans un modèle de gestion mémoire par allocation linéaire. Nous allons faire une présentation rapide puisque nous ne l'utiliserons pas pour nos modèles.

Principe Contrairement à la Recopie, il ne va utiliser qu'un seul espace. Son principe est simple, il va décaler tous les objets vivants vers le début du tas de telle manière qu'ils soient tous les uns à la suite des autres. Pour ce faire, ce GC va nécessiter un pointeur supplémentaire par objet. Celui-ci va permettre de localiser la zone où sera déplacé l'objet. En effet, il est évident que l'on ne doit déplacer les objets que lors de la dernière phase. On commence par faire une phase de marquage comme celle de *Mark and Sweep*. Puis on parcourt le tas et pour chaque objet marqué, on calcule sa nouvelle adresse de destination. On parcourt à nouveau le tas et on met à jour les pointeurs dans les objets. Pour finir, on déplace les blocs.

Discussion

Il offre les mêmes avantages que la Recopie et il ne nécessite pas un découpage de la mémoire en 2 parties ce qui limite donc les problèmes de *swapping*. Cependant il nécessite 4 phases et est donc plus lent que le GC par Recopie. Les objets que l'on déplace doivent être pointés par de vrais pointeurs. De plus ce GC provoque un *overhead* de 1 mot mémoire dû au pointeur supplémentaire par objet.

4.3 Les GC exacts

Nous allons voir brièvement les deux techniques principales que sont la technique des tags et la technique par structures de données.

4.3.1 La technique des tags

Cette technique est une technique utilisée «naturellement» par les langages à typage dynamique afin de différencier une valeur immédiate d'une référence. Elle consiste à sacrifier un bit ou plusieurs bits afin de faire cette distinction. [21, 16, 10] traitent cette technique.

4.3.2 Avec des structures de données (JVM)

La JVM est une machine virtuelle qui simule le fonctionnement d'un processeur gérant plusieurs *Thread*. Chacun d'eux possède sa propre pile (dont l'opérande est de 1 mot mémoire), son propre segment de code et son propre registre PC. Tous les Thread vont se partager le tas géré par la JVM[6]. [1] présente dans leur article une technique utilisée par la JVM pour faire une collection exacte. La pile d'un Thread contient des *stack frame*. Un *frame* est poussé sur la pile à chaque invocation d'une méthode et est dépilé lors du retour de cette méthode. Un *frame* est composé principalement des variables locales, des paramètres et d'un pointeur sur une structure de données qui indique la disposition des pointeurs dans le *frame*. Toute valeur empilée le sera sur la taille d'un mot mémoire, même un *char* ou un *short*. La taille d'un *frame* est déterminée à la compilation, contrairement à une gestion des appels de méthodes classique[7]. La JVM doit réserver de la place pour les variables locales dès la compilation ce qui l'oblige à réserver de la place pour toutes les variables locales potentiellement rencontrables lors de l'exécution de la méthode.

[6] On pourra trouver dans [14] des précisions sur le fonctionnement de la JVM.

[7] Dans une gestion classique les variables locales s'empilent dans le cadre de travail de la méthode au fur et à mesure et suivant le chemin suivi.

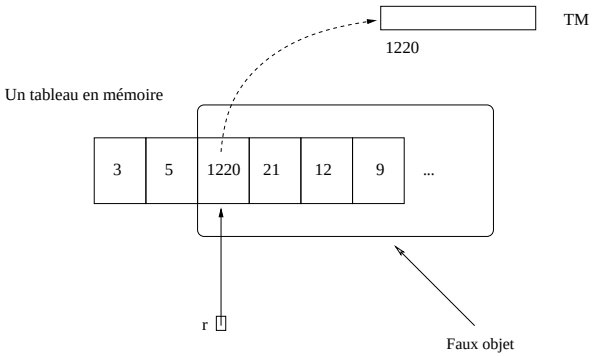

FIG. 4.7 – Problème du repérage d'un vrai objet

4.3.3 Discussion

Les GC qui utilisent un parcours exact sont des GC qui sont construits en même temps que le compilateur ce qui permet aux concepteurs de prévoir les structures de données adéquates pour résoudre le problème de la localisation des pointeurs dans la pile.

4.4 Les GC conservatifs (et semi conservatifs)

4.4.1 Parcours conservatif

Définition 6 : Un vrai objet est un objet qui a été alloué par l'allocateur.

On effectue un parcours conservatif lorsqu'on ne sait pas localiser les pointeurs dans la zone mémoire que l'on examine (pile ou objet). Cette zone mémoire apparaît donc au GC comme une suite de cases mémoires. Le principe est le suivant : tout emplacement mémoire pouvant contenir un pointeur, sera considéré comme un pointeur. Pour ce faire, on parcourt la zone séquentiellement en considérant que chaque mot est un pointeur candidat. A partir de là, seuls les pointeurs candidats qui pointent sur un vrai objet sont traités comme des pointeurs, les autres sont écartés. Nous allons dans un premier temps présenter les problèmes qui se posent pour repérer un objet valide, dans un second temps, présenter la *«misidentification»* et dans un troisième temps présenter les problèmes dus à l'arithmétique des pointeurs et des casts à la C.

Comment repérer un vrai objet à partir d'un pointeur candidat ?

Au sein d'un même langage, les objets ont des invariants d'implémentation. Par exemple, dans le cas d'un langage qui fait de l'héritage simple et qui utilise la technique des tables de méthodes, un objet sera tout simplement une zone contiguë de mémoire dont le premier mot est un pointeur sur une table de méthodes suivi des ses champs. On a donc un moyen de repérer un objet. Or ceci n'est pas suffisant. La figure 4.7 montre un cas où les données pointées en mémoire ressemblent à un objet.

Définition 7 : Un faux objet est une suite d'octets qui ressemble à un vrai objet.

Les risques provoqués par les faux objets

1. Pour un *Mark and Sweep*, un faux objet est dangereux lors de la phase de marquage car croyant marquer un objet on peut modifier en fait des données d'un vrai objet.

2. Pour un GC dans le genre du *Mostly-Copying Collector*[8] , un faux objet est dangereux lors de la copie d'un objet pointé par un champs de ce faux objet (ie ce champ peut être par exemple un entier).

Remarque 2 : Les faux objets sont étudiés en annexe A.

Technique sans structures de données

Dans cette situation, la seule façon de déterminer si un pointeur candidat pointe sur un objet valide est de commencer à scanner le tas à partir du début en passant d'un bloc au suivant jusqu'à déterminer s'il pointe ou non sur un objet valide. Mais cette technique est trop coûteuse. En outre, pour un GC qui a accès aux tables de méthodes, nous avons trouvé qu'il est possible d'obtenir un certificat exact qui garantit qu'une valeur est une adresse de début d'une table de méthodes, en d'autres termes, qu'un pointeur pointe sur une table de méthodes. Ceci permet d'arriver à une probabilité très faible de faux objets. Ainsi nous avons en premier lieu essayé de trouver une technique basée sur le *Mark and Sweep* avec une phase supplémentaire de démarquage des faux objets. Mais ceci se révèle quasiment impossible. Nous décrivons en annexe A ce certificat ainsi que les problèmes liés aux faux objets pour cette technique.

Technique à base de structures de données Les techniques à base de pages dans le genre de Boehm et de Bartlett sont les solutions les plus envisageables. Elles permettent de savoir exactement où sont situés les objets en mémoire. Boehm découpe le tas en pages de même taille et y stocke dans chacune d'entre elles des objets de même taille. Bartlett utilise un vecteur de bit par page où chaque bit correspond à un mot ce qui permet d'indiquer le commencement des objets dans la page. Cependant, ces techniques provoquent des *overheads*.

La *"misidentification"*

La *"misidentification"* est le terme anglais de mauvaise identification. La *"misidentification"* est le fait de prendre un non pointeur (par exemple un entier) pour un pointeur. En d'autres termes, le pointeur candidat n'est pas un pointeur mais un entier et ce pointeur candidat correspond bien à l'adresse d'un vrai objet. On peut donc aboutir à une rétention de mémoire. Cependant, contrairement aux faux objets, elle n'est pas dangereuse pour la cohérence de l'exécution. A partir du moment où l'on fait un parcours conservatif de la pile, le risque de *"misidentification"* est inévitable. Cependant il est possible de faire baisser les risques :

1. Les adresses de début et de fin de tas sont connues ce qui implique que toute valeur hors de ces bornes n'est pas une adresse.
2. Les objets alloués sont alignés sur une adresse de mot. Toute valeur ne correspondant pas à un alignement correct sera considérée comme une valeur immédiate.
3. Utilisation d'une liste noire (*black list*). La liste noire est un système préventif. Son principe est d'y placer des adresses qui ont posé problèmes ; par exemple un pointeur candidat qui pointe dans une zone allouable (ie dans les limites du tas, correspondant à une adresse alignée sur le mot ...) mais qui ne pointe pas sur un objet. L'objectif est d'éviter d'utiliser une de ces adresses pour une prochaine allocation afin de prévenir une "misidentification". Bien évidemment, si l'allocateur n'a plus assez de mémoire pour allouer un objet, il ira en chercher dans la liste noire.

Le problème de la *"misidentification"* est abordé dans [5]. L'impact au niveau rétention de mémoire dépend des structures de données utilisées. Par exemple si l'on utilise de très longues files, l'impact d'une *"misidentification"* pourra être terrible dans le cas où elle n'est plus accessible par

[8] Le Mostly-Copying Collecteur est étudié à la section 4.4.4.

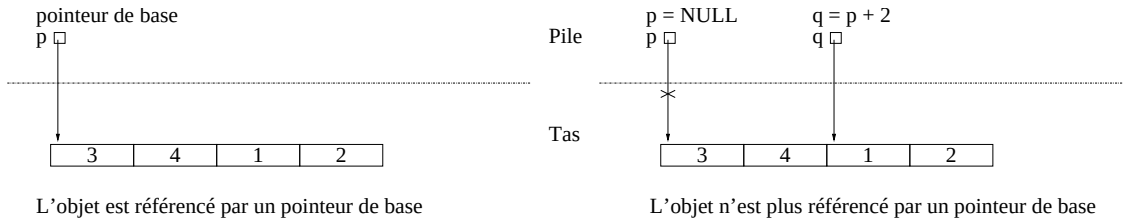

FIG. 4.8 – Problème de la manipulation des pointeurs

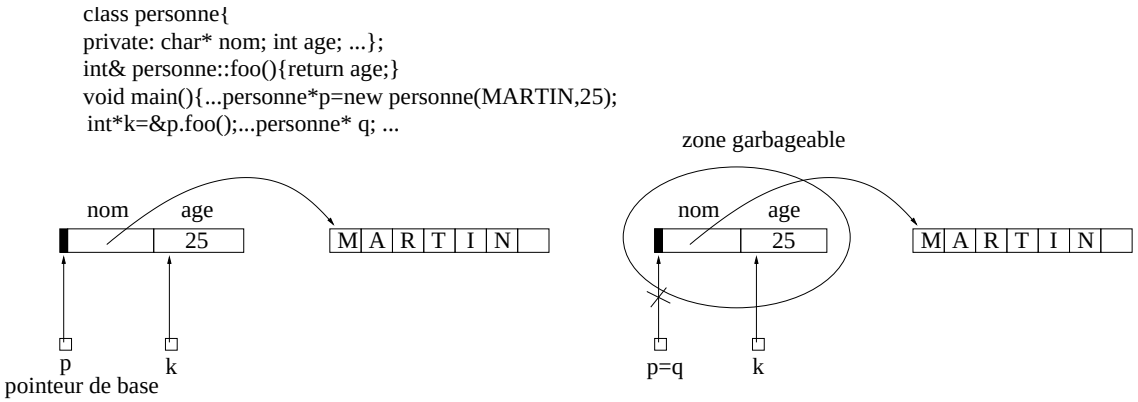

FIG. 4.9 – Problème de l'opérateur & de C++

le mutateur et où la *"misidentification"* touche le premier élément. Ainsi toute la file sera conservée en mémoire. Cependant, il a été montré que dans la grande majorité des cas la rétention n'est pas importante. De plus, Boehm précise que l'utilisation d'une liste noire permet de faire chuter grandement le risque de *"misidentification"* (quasi nul).

Le problème de l'arithmétique des pointeurs

Définition 8 : Un pointeur de base est un pointeur qui pointe au début de l'objet.

Pour déterminer si un pointeur candidat pointe sur un vrai objet, le GC va vérifier si ce pointeur est un pointeur de base. Or si le programmeur a la possibilité de manipuler directement les pointeurs, on peut se retrouver dans une situation où un objet est vivant c'est à dire atteignable par le mutateur et où le GC ne le traverse pas (voir figures 4.8 et 4.9). Ce qui viole la règle 1 du chapitre 4. Une première solution est de ramener le pointeur candidat vers le pointeur de base et donc de considérer que tout pointeur candidat qui pointe à l'intérieur d'un vrai objet est un pointeur. Ceci est réalisable avec les techniques à bases de pages. Cette technique est utilisé dans [6]. Une deuxième solution est de fournir à l'utilisateur du GC des invariants. Par exemple, toujours garder un pointeur au début de l'objet. Une étude précise et détaillée est réalisée dans [11] concernant une utilisation d'un sous-ensemble des possibilités de C++. On pourra trouver dans [18] un discours sur l'arithmétique des pointeurs.

Remarque 3 : En Java ce problème n'existe pas car une variable ratachée à un objet pointe toujours au début de cet objet.

Remarque 4 : Il est dit dans [11] que considérer un pointeur candidat qui pointe dans un vrai objet comme un pointeur fait augmenter dans le pire des cas le temps global du GC de 20% et que dans la majorité des cas, ce n'est pas un handicap.

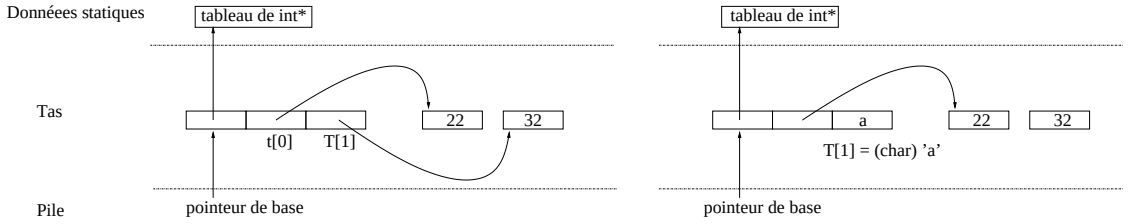

FIG. 4.10 – Problème de cast

4.4.2 Pourquoi faire un parcours conservatif ?

Les parcours conservatifs sont utilisés dans le cas où le GC est destiné à se greffer à un langage existant. En effet, dans une telle situation, il n'est pas envisageable de faire un parcours exact de la pile car celui-ci imposerait de faire d'importantes modifications au niveau du compilateur et notamment de la génération de code. A partir de là, il existe deux possibilités, faire un parcours exact des objets, ou bien un parcours conservatif des objets. On pourra trouver une présentation sur les parcours conservatifs et semi-conservatifs dans [10]. [21, 5, 7] montrent l'intérêt des GC conservatifs.

Remarque 5 : Seul le compilateur sait où se trouve le pointeur *this*.

GC semi-conservatif

Dans un GC semi-conservatif, le GC possède des informations sur les types des champs des objets. Pour obtenir ces informations, le concepteur du GC peut soit, s'il en a la possibilité, faire des modifications de haut niveau du compilateur afin d'obtenir des informations sur les types soit laisser la possibilité au programmeur de spécifier les champs de types pointeurs lors de l'allocation ([11] avec les *scanning method* et la méthode *traverse()* dans [2]). Un GC semi-conservatif peut reposer sur l'invariant suivant :

Invariant 1 : Un champ d'un vrai objet pointe sur un vrai objet.

Cet invariant permet d'éviter le test de savoir si un pointeur pointe au début d'un objet lorsque ce pointeur est un champ de type pointeur et ainsi de ne pas ralentir le GC. Cependant, si les casts à la C sont permis, cet invariant peut être violé. La figure 4.10 montre un problème lié à un cast. Un autre problème lié aux casts est de caster un entier en pointeur afin de réaliser une affectation de valeur à un champ de type pointeur. Dans ce cas, on peut arriver à pointer n'importe où et à tromper le GC.

GC purement conservatif

Un GC purement conservatif travaille en l'absence totale d'informations sur la localisation des pointeurs. L'avantage d'un tel GC est qu'il ne requiert aucune modification du compilateur et qu'il est très facile à utiliser pour le développeur. De plus ces GC peuvent permettre de lutter contre les problèmes provenant de l'arithmétique des pointeurs et des casts à la C.

Un mot sur C++

L'implémentation standard de C++[9] est une implémentation par sous-objets ce qui fait qu'un pointeur pointant sur un sous-objet peut tromper le GC puisque celui-ci n'est pas le pointeur de base. Il faut donc en tenir compte lors de la création d'une classe dérivant d'une classe virtuelle. Nous pourrons voir cela dans [11, 2].

[9] On pourra trouver une description de l'implémentation standard de C++ au chapitre 5 section 2.

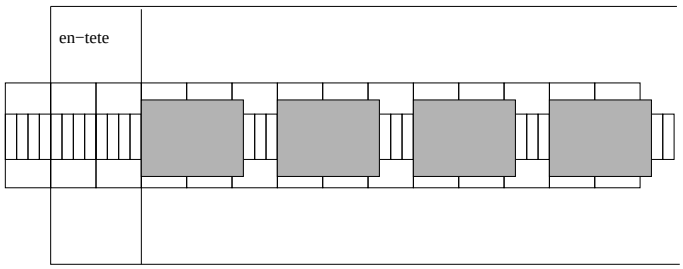

FIG. 4.11 – Boehm. Les petits objets

4.4.3 Le GC purement conservatif de Boehm

Présentation

Boehm a créé des GC qui se basent sur un *Mark and Sweep* purement conservatif afin de fonctionner dans un environnement hostile (le plus connu est le BDW). En d'autres termes, il s'adapte à des langages tel que C et C++. Son succès pour de tels langages repose sur 3 points. Le premier est qu'il ne touche pas au compilateur, le deuxième est qu'il permet de lutter contre le problème de l'arithmétique des pointeurs et le troisième est que ses performances sont comparables à une gestion explicite de la mémoire dans la plupart des programmes. On pourra trouver le principe de fonctionnement dans [19, 18]. [5] explique comment rajouter une liste noire et [4] donne des explications sur le filtrage des pointeurs candidats.

Organisation de la mémoire

Le tas est découpé en page de 4 Ko. Elles sont alignées sur leur taille et peuvent contenir des petits objets, faire partie d'un gros objet ou bien être libre. Chaque page a une entrée dans une table et à chaque entrée on fait correspondre un entier.

Les petits objets

Une page contient des petits objets qui possèdent tous la même taille. Chacune de ces pages possède une en-tête composée d'un mot qui permet de connaître la taille des objets se trouvant dans la page et une *bitmap* (ie tableau de bit) qui permet de savoir quels objets sont marqués. Les pages sont allouées lorsqu'on en a besoin par un simple malloc, quant aux pages libres, on les maintient dans une *free list*. De plus, on maintient une *free list* par taille d'objet.

Dans la figure 4.11, les objets sont représentés en gris et ont pour taille 5 octets. Nous voyons qu'il y a le respect de l'alignement sur l'adresse d'un mot.

Remarque 6 : Afin d'éviter d'assigner trop de pages pour chaque taille, l'allocateur va d'abord chercher à trouver une place dans une page contenant des objets de taille juste supérieure. Les objets créés tôt dans l'exécution sont suceptibles d'être assignés avec exactement la taille demandée.

Les gros objets

Un gros objet (figure 4.12) est un objet qui va nécessiter plus d'une unité pour le contenir.

Le marquage

L'idée de Boehm est d'éviter de faire un marquage récursif. Pour ce faire, il utilise une structure de stockage appelée *mark stack*. Quand une collection commence, le collecteur recherche l'ensemble

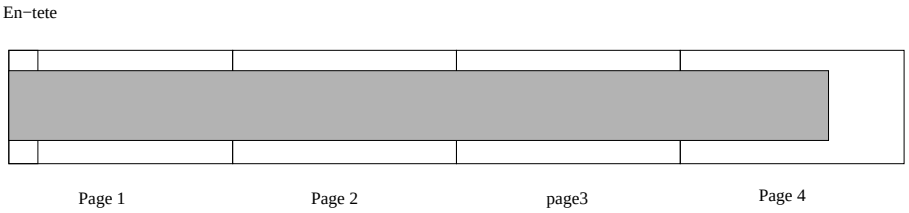

FIG. 4.12 – Boehm. Les gros objets

des racines. Chaque référence à un objet trouvé est empilée dans le *mark stack* et l'objet associé est marqué. On gère le *mark stack* comme une file. Tant que le *mark stack* n'est pas vide, on extrait un élément et on scanne l'objet associé. Lors du scannage de l'objet, on y empile les références aux objets non marqués et on marque ces objets.

Comment trier les pointeurs candidats

1. On vérifie que le pointeur candidat pointe sur une adresse qui est alignée sur le mot.
2. On regarde si l'adresse se trouve dans les limites du tas alloué.
3. On calcule l'adresse de la page correspondante et on recherche l'entier correspondant dans une table. Chaque entrée de la table contient soit 0 indiquant que la page ne fait pas partie du tas du GC, soit un petit entier n indiquant que la page fait partie d'un gros objet commençant à n pages en arrière, soit un pointeur sur le descripteur de la page. Dans le premier cas, le pointeur candidat n'est pas un vrai pointeur. Dans les 2 derniers cas, on peut obtenir un descripteur pour la page.
4. On vérifie que le pointeur candidat pointe au début d'un objet. (un simple calcul puisque les objets dans la page sont tous de la même taille)
5. On vérifie que l'objet est alloué en passant par une table.

Remarque 7 : Le point 4 ne traite pas l'arithmétique des pointeurs. Par contre il exite une interface optionnelle dans laquelle on demande à ce qu'ils soient pris en considération.

La liste noire

Pour des raisons de performances et de simplicités, il met des pages entières dans la liste noire plutôt que des adresses individuelles. Cette *black list* peut être implémentée par un tableau de bits indexé par les numéros de page. Dans son GC, la liste noire n'est examinée que lorsque l'allocation d'une page est nécessaire.

Le sweep

Il consiste à parcourir les pages et à analyser la *bit-map* pour regarder quels objets sont marqués et lesquels ne le sont pas.

Discussion

Cette technique à base de page qui consiste à stocker uniquement des objets de même taille permet d'éviter tout overhead supplémentaire dans les objets. Cependant, cette technique peut provoquer des *overheads* dans les pages. En effet on n'alloue pas des pages pour chaque taille d'objet. Ainsi lorsque l'objet ne sera pas de la taille exacte il y aura un *overhead*. Ce GC ne sera donc pas performant dans le cas où le mutateur demande énormément d'allocation pour des objets de taille différente. De plus une page peut contenir un autre type d'*overhead*, c'est la place

perdue à la fin d'une page dans le cas où le découpage de la page ne tombe pas juste. En outre, il demande peu de mémoire mais provoque de la fragmentation et impose un parcours du tas (phase de *sweep*). L'allocation reste coûteuse par rapport à une allocation linéaire (GC par copie).

4.4.4 Le MCC (Mostly-Copying Collector)

Le principe a été introduit par Bartlett dans [3] avec son *Mostly-Copying Collector* semi-conservatif. L'idée de base est de faire de la copie malgré un environnement hostile (racines ambiguës) en détectant les objets non copiables (objets racines) et en copiant les autres. Dans les premières versions, Bartlett utilise une stratégie de gestion mémoire à la Boehm et vise les structures de données de petites tailles et peu nombreuses comme les cellules. Une autre stratégie consiste à découper le tas en pages de même taille et d'utiliser un vecteur de bits par en-tête de page afin de localiser le début des objets. Côté fonctionnement, le concept des 2 espaces va être préservé mais ces 2 espaces ne sont pas figés comme dans le GC par Recopie classique. Dans une première phase, on scanne la pile et on marque non copiables les pages qui contiennent des objets racines et on les relient dans une liste (*to-space*). Dans une seconde phase, le GC réalise la copie à la manière de l'algorithme de Cheney en utilisant les objets marqués comme non copiables dans *to-space* comme des racines. Il existe plusieurs versions du *Mostly-Copying Collector*. Le GC de Bartlett fait de la copie «agressive» en copiant lors de la première phase tous les objets se trouvant dans les pages non copiables. Puis en utilisant l'algorithme de cheney. A la fin, il remet à leur place les objets qui étaient non copiables. Cette technique provoque de la fausse rétention (les objets morts qui se trouvent dans les pages non copiables) pouvant aller jusqu'à 50% [2]. Le CMM[10] est une optimisation qui a pour but d'éviter les fausses rétentions en utilisant une *LiveMap*. Mais leur algorithme provoque de la récursion [2].

Avantages

L'avantage du *Mostly-Copying Collector* est qu'on a un contrôle de la fragmentation et que les données sont réparties sur un nombre limité de pages physiques. Dans le premier schéma de la figure 4.13 les objets sont de 2 sortes de taille. Les pages du GC permettent d'accueillir au maximum 3 objets. Dans ce schéma il y a 4 pages virtuelles dont chacune d'entre elles est constituée de 4 pages du GC. Dans cet exemple, les objets utilisent 4 pages virtuelles. Les objets A et D sont des objets racines et donc leur page respective est marquée. Les objets F, G, H et I sont des objets morts. Dans le second schéma de la figure 4.13 le GC a été effectué. La fragmentation a diminuée et il n'y a plus que 2 pages virtuelles utilisées. Les objets A, B et C qui sont chaînés entre eux font partie de la même page virtuelle et donc de la même page physique. Le parcours de la chaîne A-B-C sera donc plus rapide que dans le premier schéma de la figure 4.13.

Inconvénients

Il y a de la fragmentation dans les pages non copiables.

Le MCC

Le MCC est une idée de F. Smith et de G. Morrisett introduit dans [18, 19]. C'est un *Mostly-Copying Collector* qui va éviter les fausses rétentions, qui ne va pas être récursif et qui va faire qu'une seule passe sur les objets vivants. C'est en fait un mélange de *Mark and Sweep* et du *Mostly-Copying Collector*. Le *Mark and Sweep* va être utilisé pour les gros objets et pour les objets ne pouvant pas être déplacés. Le *Mostly-Copying Collector* pour les autres. Contrairement au GC de base de Bartlett ou au BDW, le MCC va utiliser une en-tête pour les objets. L'objectif annoncé est de garder les avantages du GC par recopie en utilisant l'allocation linéaire. Il organise la mémoire de la façon suivante : il utilise des grands blocs (1/2 MB) contenants des pages (2 KB). La taille d'un grand bloc pouvant contenir le plus gros objet. Ces blocs sont alignés sur des

[10] CMM : Customizable Memory Management.

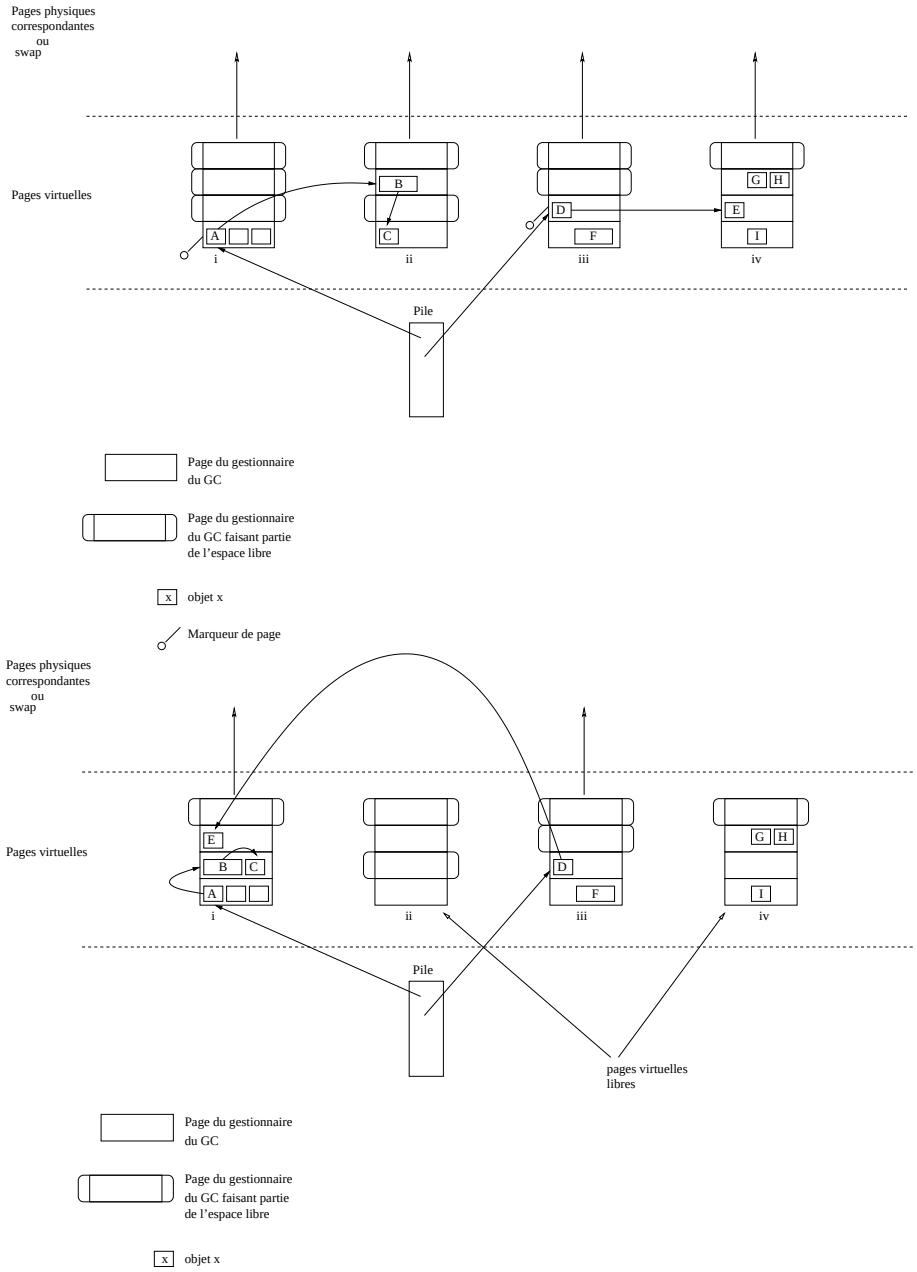

FIG. 4.13 – Mostly-Copying Collector

adresses multiples de la taille d'un grand bloc et les pages sont ainsi alignées sur des adresses multiples de la taille d'une page. Chaque grand bloc possède une en-tête composée de *structures Pageinfo*. On associe à chaque page une structure *Pageinfo* (4 mots). Un objet va avoir un mot d'en-tête déterminant la taille, le type et son état. Le GC reconnaît trois types d'objets : Les objets dont on peut déterminer les champs qui sont de type pointeur (objets sûrs), les tableaux et les objets conservatifs. Pour les objets sûrs, une *bitmask* permet de déterminer la localisation des pointeurs dans l'objet. Lors d'une collection, l'objet peut se trouver dans l'un de ces 4 états :

1. Normal : L'objet n'est pas encore traversé par le GC.
2. *Forwarded* : L'objet a été copié et un *forwarding pointer* pointe sur la copie.
3. *Pinned* : L'objet ne peut pas être déplacé.
4. Marqué : L'objet est *"pinned"* et a été déjà atteint par le GC.

Le GC maintient 6 structures de données :

1. *free list* : Une free list des pages qui ne sont pas utilisées.
2. *from space* : la liste des pages de l'espace courant.
3. *cheney queue* : la liste des pages qui contiennent les copies (c'est une liste implicite).
4. *mark queue* : une liste contenant les références aux objets atteints mais pas *«pinned»*. Cette liste est analogue à celle utilisé par Boehm (*mark stack*).
5. La liste des pages *«pinned»* (contenant des objets qui ne peuvent pas être déplacés).
6. La liste des objets conservatifs.

L'allocation d'un objet sûr est effectué par une allocation linéaire. La collection est réalisée de la façon suivante :

- Dans une première étape, on détermine les objets qui ne peuvent pas être déplacés en scannant la pile et la liste des objets conservatifs. Le filtre qui permet d'écarter les mauvais pointeurs candidats est quasiment similaire à celui de Boehm. Cependant la différence principale se situe dans la technique qui consiste à repérer le début d'un objet. Ici les objets stockés dans les pages sont de tailles différentes. La solution utilisée est de se ramener au début de la page et de passer séquentiellement les objets (ie on connaît la taille) jusqu'à pouvoir déterminer si le pointeur pointe au début d'un objet. On marque ces objets comme atteint et comme *«pinned»*, puis on les place dans la liste *mark queue*. On marque également leur page comme *«pinned»* et on *«pin»* tous les autres objets de la page.
- Dans une deuxième étape, on réalise l'exploration du graphe d'objets à partir de la cheney queue et de la *mark queue*. Tant que ces 2 files ne sont pas vides, on extrait un élément de l'une des 2 files (arbitraire) et on traite chacun de ses enfants en regardant si :

1. *pinned*, on le marque comme atteint et on place sa référence dans *mark queue*.
2. marqué comme atteint, on ne fait rien.
3. *forwarded*, on met à jour le pointeur.
4. non atteint, on le met dans la cheney queue (ie on le copie) et on met un *forwarded pointer* dans l'original vers la copie.

Dans une troisième phase, on réalise un *sweep* sur la liste des pages *"pinned"*. On démarque les objets marqués et on efface les objets morts afin d'éviter les futures fausses références. Puis on concatène la liste des pages *"pinned"* à la *cheney queue* et on libère toutes les pages de *from space* en les insérant dans *free list*. Enfin, la *cheney queue* devient *from space*.

Remarque 8 : Les gros objets sont traités à part par un Mark and Sweep.

4.4.5 Discussion

Les GC conservatifs et semi-conservatifs ont pour intérêt de pouvoir se greffer à un langage qui n'avait pas était conçu à la base pour fonctionner avec un GC. Ces GC s'exposent à la «misidentification». Dans la plus part des cas, elle s'avère être rare surtout si l'on a un bon filtrage des pointeurs candidats et qu'on utilise une liste noire. [18, 19] compare la technique de Boehm avec leur MCC. Il en ressort que le choix du GC dépend du contexte d'exécution. Le MCC fonctionne bien dans le cas où la mémoire RAM est abondante (sinon problème de sawpping), les objets vivants sont en petit nombre et où le taux d'allocation est élevé. Par contre, le BDW se révèle être meilleur lorsqu'il y a peu de mémoire RAM, et lorsque les objets ont une longue durée de vie. Pour un langage qui ne manipule que des cellules, le MCC peut provoquer jusqu'à 50% d'*overhead* par cellule et pas le BDW. L'avantage du BDW s'atténue cependant car on aligne tous les objets de taille supérieure à deux mots sur le double mot. De plus le BDW n'utilise pas les mots qui servent à pointer sur les tables de méthodes. Dans le cas où on alloue beaucoup d'objets de taille différente, le MCC et le BDW sont comparables sur la plupart des *Bench*. Par contre sur certains *Bench*, le MCC est meilleur de 40%. En effet, le BDW a des *overheads* dus à sa structure de données qui consiste à stocker des objets de même taille dans chaque page. En ce qui concerne la détermination d'un pointeur candidat, un pointeur qui ne passe pas le filtre coûtera le même prix pour le MCC et pour le BDW (ie peu coûteux). Cependant, un pointeur candidat qui passe le filtre avec succès nécessitera 30 instructions machines pour le BDW et environ 200 instructions pour le MCC. Au final, Smith et Morrisett montre que leur MCC est meilleur que le BDW sur la plupart des *Bench*.

Remarque 9 : On pourra voir une autre solution basée sur les pages dans [22].

4.5 Les GC générationnels

4.5.1 Présentation

On a vu que les GC explorateurs étaient constitués de 2 taches importantes : la construction du graphe d'objets et la collecte. Dans cette optimisation, c'est au niveau de la phase de ramassage des objets que l'on intervient avec pour idée de réduire la zone du tas que l'on va parcourir. Les GC générationnels se basent sur le constat suivant : plus un objet est ancien, moins il a de chance de disparaître et plus un objet est jeune, plus il a de chance de disparaître. L'idée va être de hiérarchiser le tas en plusieurs zones suivant l'ancienneté des objets. Lors de la phase de ramassage, on n'examinera uniquement certaines zones. Les GC générationnels sont abordés dans [21].

4.5.2 Le mark and compact générationnel

Le Mark and compact générationnel est utilisé dans le GC de .NET et dans celui de la JVM. Ce GC se base sur une sûreté des racines (parcours exact). Le GC va se déclencher suite à un échec d'une requête d'allocation faite par le mutateur à l'allocateur. Lors du tout premier compactage, il n'y a pas encore de générations. La figure 4.14 montre que le tout premier compactage s'effectue sur tout le tas puisque tous les objets alloués font partie de la nouvelle génération (génération 0)[11]. Puis des allocations se font jusqu'à ce qu'il y ait un manque de mémoire. La figure 4.15 montre un autre déclenchement du GC. Les objets qui faisaient partie de la génération 0 passent en génération 1 et ceux que l'on a alloués «fraîchement» sont en génération 0. Le GC ne fait du compactage que sur la génération 0 si la place libérée est suffisante. C'est le cas dans cette figure. La figure 4.16 montre une situation similaire à la précédente dans laquelle la génération 1 devient la génération 2, la génération qui était à 0 passe en génération 1 et les objets «fraîchement» alloués

[11] A noter que la lettre D sur les figures 6.14 à 6.17 signifie déréférencement de pointeur.

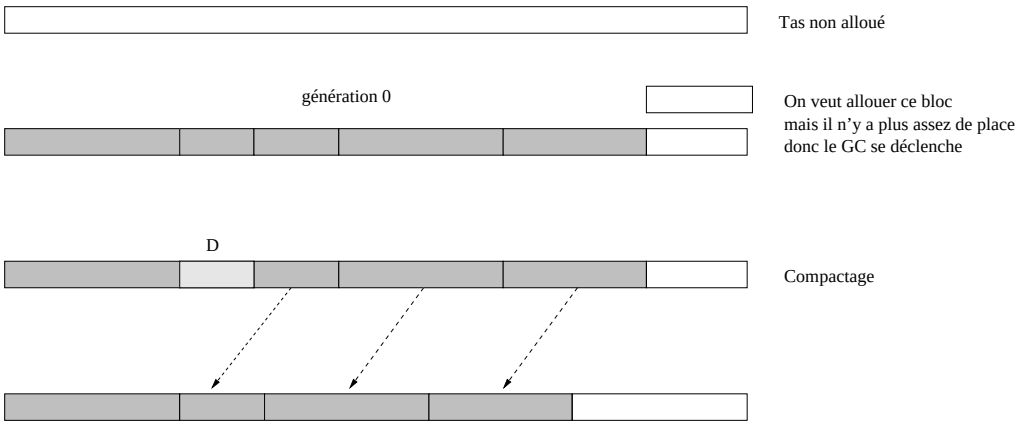

FIG. 4.14 – génération 0

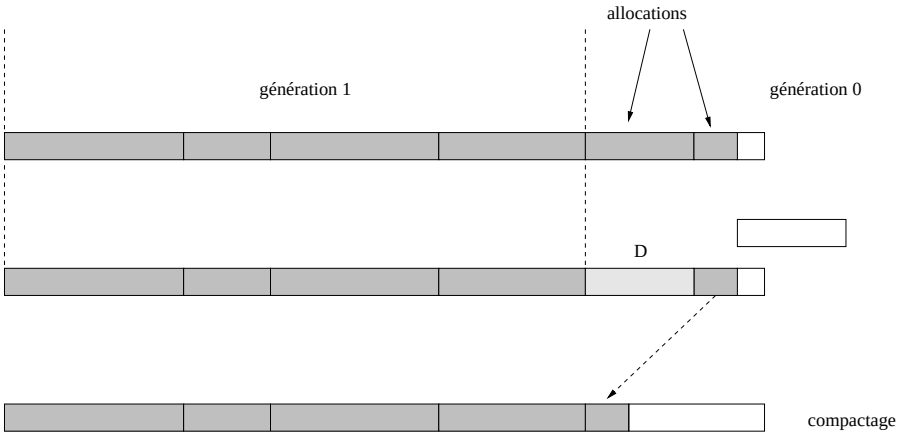

FIG. 4.15 – génération 1

sont en génération 0. La figure 4.17 montre par contre une situation dans laquelle le compactage sur la génération 0 n'est pas suffisante pour obtenir de la mémoire nécessaire demandée. A ce moment là, on fait du compactage sur la génération 1. Si ce n'est pas suffisant, il faut faire du compactage sur la génération 2 et ainsi de suite.

Remarque 10 : Avec cette implémentation le concept de génération est trivial à implémenter car le tas respecte l'ordre de création des objets.

Remarque 11 : Cette technique permet d'éviter la fragmentation du tas et permet de réduire le temps d'attente du mutateur.

4.5.3 Discussion

L'un des problèmes qui se pose avec les GC générationnels est la gestion des pointeurs qui vont de la vieille génération vers la génération des jeunes. Cependant, il s'avère que ce cas est rare. Pourtant de telles références existent et le GC doit en tenir compte. Une des solutions employée est de conserver dans une zone mémoire les pointeurs pointant vers la jeune génération. Ainsi lors d'un GC sur la jeune génération, on considère ces pointeurs comme des racines.

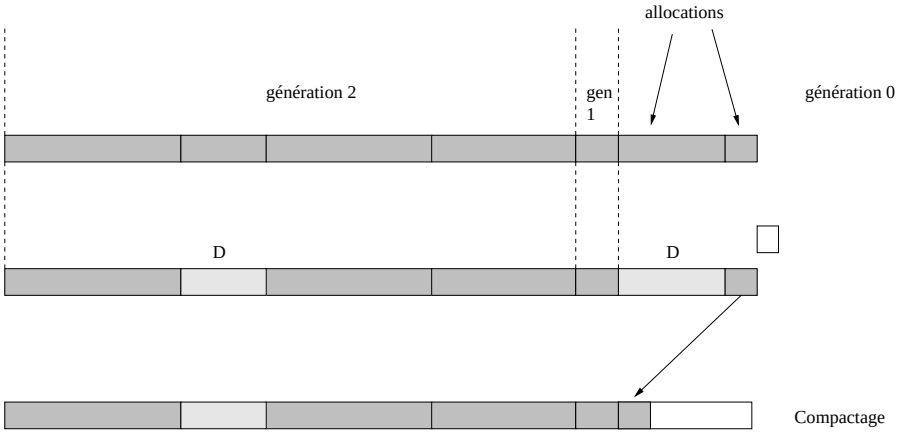

FIG. 4.16 – génération 2

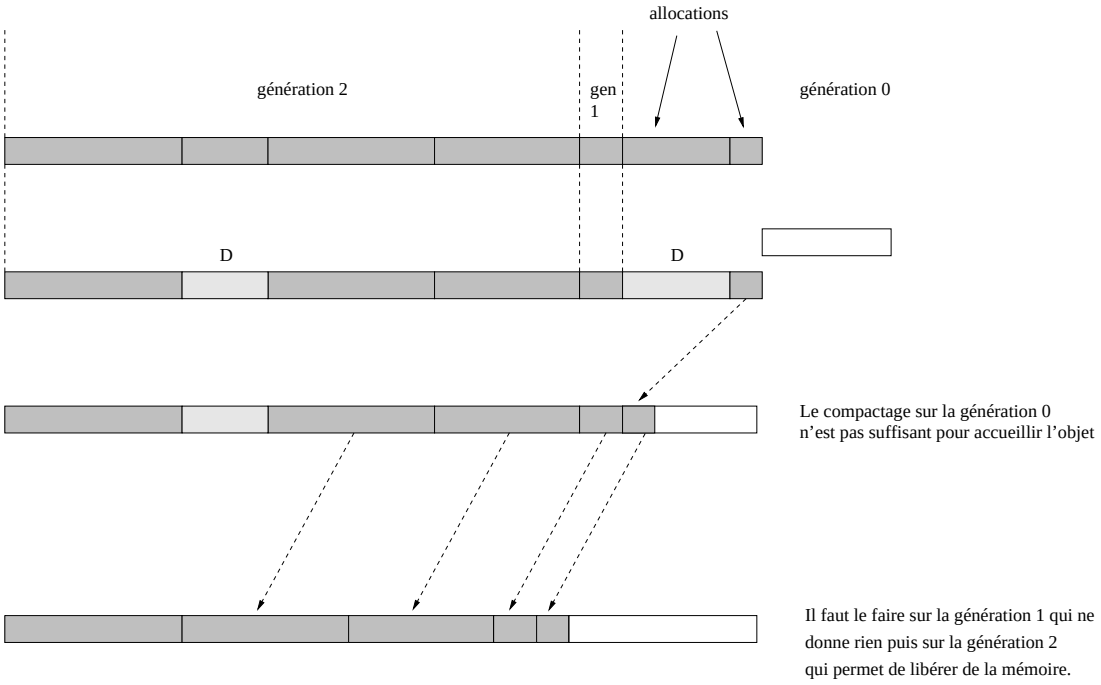

FIG. 4.17 – génération 3

Chapitre 5

Principe de la coloration

L'objectif principal de cette technique est de donner une implémentation de l'héritage multiple aussi performante que celle de l'héritage simple. Nous allons rappeler brièvement la technique d'implémentation de l'héritage simple, puis présenter les problèmes liés à l'implémentation standard et nous présenterons enfin la coloration. On pourra trouver une étude complète de la coloration dans [8] et une étude des techniques des langages objets à typage statique dans [9].

5.1 Principe de l'héritage simple

L'implémentation classique en héritage simple repose sur 2 invariants :

5.1.1 Les invariants

1. Invariant de référence.
2. Invariant de position (par héritage).

L'invariant de référence signifie que tout pointeur sur un objet ne dépend pas de son type statique à l'exécution. L'invariant de position signifie que toute méthode (Resp. attribut) introduite dans une classe et se trouvant à une position p dans sa table de méthode (Resp. une instance de cette classe) se trouvera également à la position p dans toute table de méthode (Resp. instance d'une sous-classe directe ou indirecte) d'une sous-classe directe ou indirecte.

Remarque 1 :
Dans cette implémentation, tout pointeur sur un objet pointe au début de l'objet.

5.1.2 Les tables de méthodes

Il y a une table de méthodes par classe.

5.1.3 Liaison tardive

Elle s'implémente facilement. Lors de la compilation, on récupère la propriété générique la plus spécifique dans le type statique du receveur. On détermine sa position dans sa table de méthodes. A l'exécution, on va choisir la bonne propriété locale en se branchant sur la table de méthodes du type dynamique du receveur. On récupère alors l'adresse se trouvant à la position calculée lors de la compilation.

Remarque 2 : La liaison tardive se fait en temps constant.

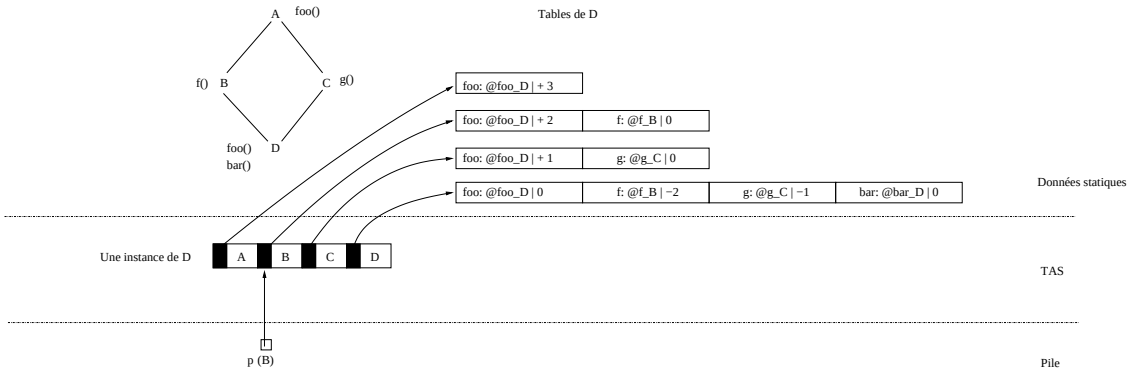

FIG. 5.1 – Implémentation standard de C++

5.1.4 L'accès à un attribut

C'est trivial :
load [object + #attribut_offset], attribut

5.1.5 Conclusion

Cette implémentation a un bon rendu au niveau place statique (nombre de tables linéaires dans le nombre de classes). La place dynamique est optimale. Au niveau temporel, le passage de paramètres ou l'affectation ne nécessitent pas de casting ascendant. Il n'y a donc aucune pertes temporelles.

Remarque 3 : Cette implémentation est "propre" du point de vu théorique car les types statiques disparaissent lors de l'exécution.

5.2 Principe de l'implémentation standard

La limitation de l'implémentation de l'héritage simple intervient dès qu'il y a de l'héritage multiple. En effet, si l'on essaye de préserver les invariants de l'héritage simple, cela devient impossible sur un héritage multiple de type losange. Il y a alors violation de l'invariant de position. Le but de l'implémentation standard est de trouver une implémentation compatible avec l'héritage multiple. Cette implémentation se base sur le concept de sous-objet et sur les 2 invariants suivants :

5.2.1 Les invariants

1. Un pointeur sur un objet pointe toujours sur le sous-objet correspondant à son type statique (qui, lui, pointe sur sa propre table de méthodes).
2. Chaque méthode (Resp. attribut) a une position invariante et non ambiguë dans le contexte du type statique qui l'introduit.

5.2.2 Table de méthodes

Une classe a autant de tables de méthodes que de super classes (directes et indirectes) plus une. Dans chaque table de méthodes, on va associer à chaque méthode une adresse et un décalage vers le type qui l'a introduit. Dans la figure 5.1, p est un pointeur de type statique B et de type dynamique D. p pointe sur le sous-objet de son type statique.

5.2.3 Liaison tardive

Lors de la compilation, on récupère la propriété générique la plus spécifique dans le type statique du receveur. On détermine la position de la méthode dans le contexte de ce type statique. A l'exécution, on va choisir la bonne propriété locale en se branchant sur la table de méthodes du type dynamique correspondante au type statique du receveur. On va d'abord effectuer le décalage du pointeur this vers le sous-objet du type qui l'a introduit. Pour ce faire, on va à la position calculée précédemment et on récupère ce décalage. Puis on récupère, de même, la bonne adresse.

Remarque 4 : La liaison tardive se fait en temps constant.

5.2.4 Conclusion

C'est une implémentation lourde. Au niveau statique, le nombre de tables est quadratique dans le nombre de classes et le code généré est plus important à cause des décalages. Au niveau dynamique, chaque objet possède un *overhead* de un pointeur (vptr) par sous-objet voire plus dans certains compilateurs C++ qui y mettent aussi les vbptr visant à accélérer l'accès aux attributs (en supprimant une indirection). Au niveau temporel, le passage de paramètres ainsi que l'affectation nécessitent un cast ascendant (puisqu'il n'y a pas invariant de référence).

Remarque 5 : Cette implémentation est donc très dépendante des types statiques à l'exécution.

5.3 La coloration

La coloration est une technique qui nécessite une étude globale. Mais elle peut, tout de même, être utilisée en compilation séparée en l'effectuant lors de l'édition de liens. Son objectif principal est de se ramener aux invariants de l'héritage simple (que l'héritage soit simple ou multiple).

5.3.1 Les invariants

1. Chaque attribut (Resp. méthode) a une couleur (position) invariante par héritage.
2. Invariant de référence

5.3.2 Attribution des couleurs

Si on fait une attribution des couleurs naïve, on peut se retrouver avec des tables possédant des trous. Ces trous pouvant être très nombreux. Trouver une solution optimale est un problème NP-complet. La coloration est une heuristique qui va optimiser l'attribution des couleurs.

Remarque 6 : La coloration consiste à essayer de minimiser la taille des tables.

Remarque 7 : La technique classique de l'implémentation par table de l'héritage simple est en fait un cas particulier de la coloration. Dans ce cas, l'occupation de l'espace est optimal (ie pas de trous).

La figure 5.2 montre un exemple d'application de la coloration. On précise que la zone noire est un trou et que *foo()* est redéfinit dans D.

5.3.3 Coloration bi-directionnelle

La coloration bi-directionnelle consiste à utiliser des couleurs positives et négatives. L'objectif étant de réduire encore plus le nombre de trous. Trouver une solution optimale reste un problème NP-complet. Il est à noter cependant que si le graphe de la hiérarchie de classes est biparti, on sait trouver une solution optimale en temps polynomial.

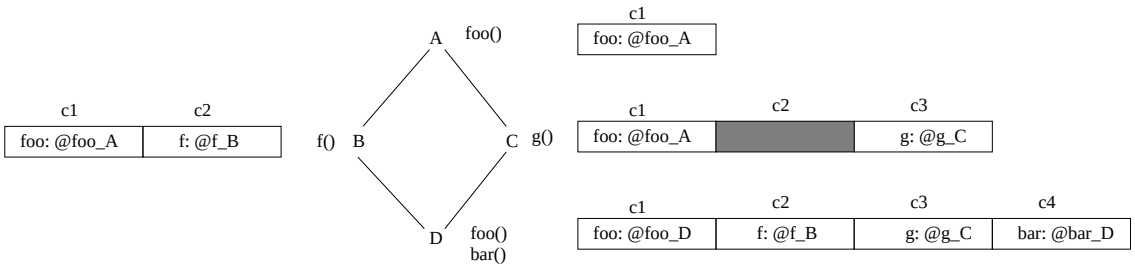

FIG. 5.2 – La coloration

5.3.4 Schéma de compilation

Comme nous venons de le voir, nous sommes dans un cadre de compilation séparée. Ce schéma de compilation se décompose en 2 phases.

1. La phase locale.
2. La phase globale.

La phase locale La phase locale est la phase de compilation d'une classe. Cette compilation va fournir 3 informations :

1. - le schéma externe (interface + super-classes).
2. - le schéma interne (circulation des types dans les méthodes).
3. - le code décoré de balises.

La phase globale Dans un premier temps, on va réaliser l'analyse de types. Cette analyse va permettre de détecter les classes vivantes, les méthodes vivantes et les appels monomorphes (permettant de faire une liaison statique). Dans un second temps, on va réaliser la coloration bidirectionnelle. Elle va donc créer des identifiants et des couleurs. Et dans un troisième temps, on va procéder à la substitution de balises dans le code. On obtient alors un code propre pour l'édition de liens.

Chapitre 6

Étude d'un GC pour la coloration bi-directionnelle

6.1 Introduction

6.1.1 Les hypothèses

Nous venons de voir dans le chapitre précédent la technique de la coloration bi-directionnelle ainsi que ses invariants. Notre objectif est d'adapter un GC à un langage objet à typage statique dont le compilateur est implémenté suivant cette technique. Nous partirons avec les hypothèses suivantes :

1. Tout est objet (au sens programmation par objet).
2. Les objets manipulés sont des tableaux et des instances de classe classiques.
3. Tout objet a une table de méthodes.
4. Deux instances d'une même classe ont la même taille et leur taille est contenue dans la table de méthodes.
5. Deux objets de type tableau peuvent avoir une taille différente. La taille est alors stockée dans l'objet.
6. Un pointeur sur un objet (instance d'une classe) pointe au milieu de l'objet.
7. Un pointeur sur un objet (tableau) pointe au début du tableau.
8. Comme en Java, les pointeurs sont transparents pour le programmeur.

Définition 1 : On appellera PTM le pointeur qui pointe vers sa table de méthodes.

6.1.2 Quelles stratégies sont-elles envisageables ?

Nous nous trouvons dans une situation dans laquelle le GC est un élément externe au compilateur et dans laquelle on a accès à celui-ci (ie modifications de haut niveau). Nous nous dirigeons donc vers un GC semi-conservatif avec un parcours conservatif de la pile et un parcours exact des objets. De plus, nous pourrons utiliser les tables de méthodes comme structure de données afin de connaître les champs de type pointeur. Nous allons éviter le problème des faux objets en utilisant des structures à base de pages. Quant aux problèmes de l'arithmétique des pointeurs, ils sont trivialement écartés par hypothèses. Seule la "misidentification" subsiste mais elle est inévitable et pas dangereuse. De plus, celle-ci peut être en grande partie évitée avec notamment le principe de la liste noire. Dans nos architectures, nous allons utiliser le *Mark and Sweep* et nous allons nous inspirer des techniques du MCC et de Boehm.

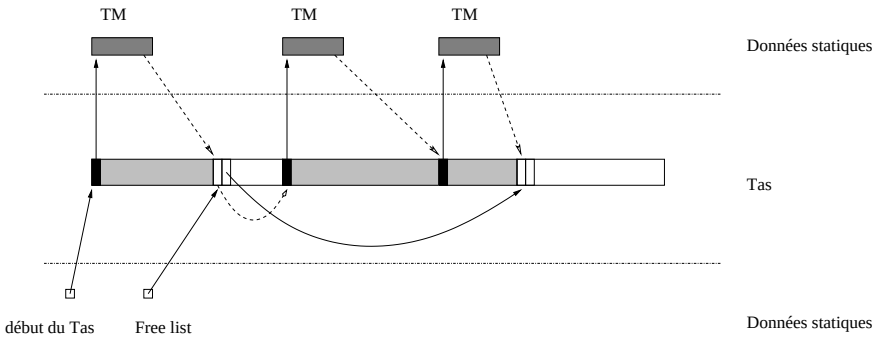

FIG. 6.1 – Accès au bloc suivant pour un langage classique

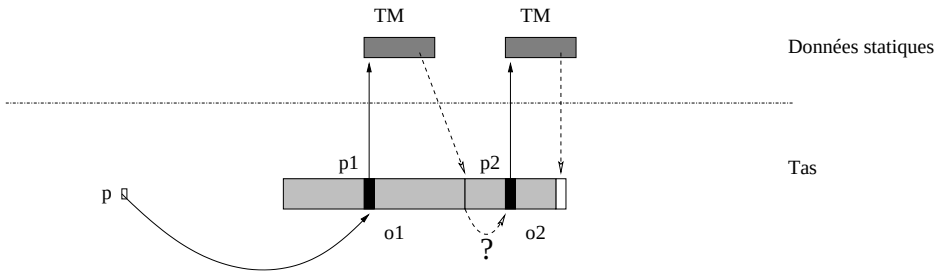

FIG. 6.2 – Accès au bloc suivant et problème pour la coloration

6.2 Présentation générale de nos architectures

Nous allons commencer par présenter les architectures des objets, des blocs libres, des tables de méthodes et des *free list* car elles sont invariantes suivant nos différents modèles. Nous présenterons la suite en deux parties :

- Premièrement une stratégie où le plus gros objet du programme peut tenir dans une page en présentant d'abord une version basée uniquement sur le *Mark and Sweep.* Dans cette version, nous présentons volontairement un modèle récursif afin de bien exposer le principe. Ensuite nous présentons un modèle basé sur le *Mostly-Copying Collector* et sur le *Mark and Sweep.* Nous en donnons ici une version non récursive.
- Deuxièmement nous verrons 2 modèles dans lesquels le plus gros objet du programme dépasse la taille d'une page. Le premier, en considérant que la plus grande partie négative possible de la coloration bi-directionnelle peut tenir dans une page et le deuxième, en considérant que non.

6.3 Architecture

6.3.1 Architecture d'un objet

Le problème qui se pose est de savoir comment passer de bloc en bloc lors de la phase de *sweep* sans provoquer d'*overheads* supplémentaires dans les objets. Nous pouvons voir l'architecture d'un objet à la figure 6.3. L'idée est d'utiliser les tables de méthodes en y stockant un delta pour aller à la fin de l'objet et ainsi se trouver au début du bloc suivant. Ceci serait facilement réalisable dans un langage où tout pointeur sur un objet pointe au début de l'objet (figure 6.1). Le problème de la coloration est que le pointeur pointe dans le cas général au milieu de l'objet (figure 6.2).

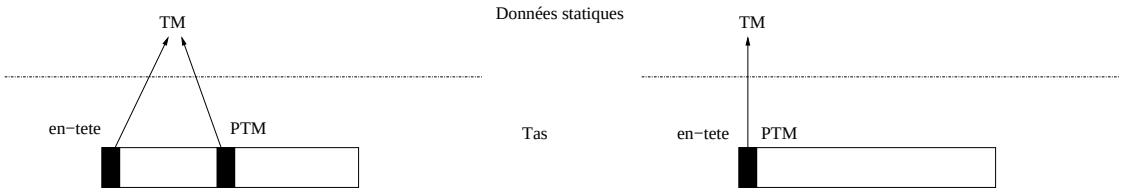

FIG. 6.3 – Architecture d'un objet

On rajoute une en-tête au début de l'objet

Nous avons donc choisi de rajouter une en-tête à l'objet. Cette en-tête est un pointeur sur la même table de méthodes. Une autre alternative était de mettre pour en-tête la taille de l'objet avec pour objectif d'accéder directement au bloc suivant sans passer par une indirection dans la table de méthodes. Le problème est, qu'avec cette solution, l'*overhead* supplémentaire est toujours de 1 mot mémoire. Or, avec la solution que nous avons choisi, il y a un cas particulier dans lequel on a pas d'*overhead* supplémentaire. Ce cas particulier est celui où il n'y a pas de deltas négatifs. En effet, dans ce cas là, il n'y a qu'un seul pointeur sur la table de méthodes, le pointeur d'en-tête.

On aligne les objets sur le mot.

L'adresse d'un objet est donc un multiple de 4 ce qui implique que les 2 bits de poids faibles sont à 0.

Aspect du mot de l'en-tête d'un objet

1. Le premier bit de poids faible est le bit d'état. (0 occupé)
2. Le deuxième bit de poids faible est le bit de marquage. (ie 0 non marqué, 1 marqué).

La valeur des bits d'état et de marquage est choisie pour chacun d'eux judicieusement à 0 par défaut (ie Les tables de méthodes étant alignées sur le mot). En effet, si on se trouve dans le cas particulier décrit précédemment, il n'y a qu'un seul pointeur sur la table de méthodes. L'idée est de ne pas avoir à appliquer un masque pour trouver l'adresse de la table de méthodes afin de ne pas ralentir l'exécution, par exemple lors de l'accès à la table de méthodes pour la liaison tardive.

6.3.2 Architecture d'un bloc libre

Un bloc libre a une taille au moins égale à un mot mémoire à cause de l'alignement des objets sur le mot. Un bloc libre de taille supérieure ou égale à 2 mots mémoire (bloc libre classique) va être constitué d'une en-tête formée de 2 mots. Le premier mot pour la taille du bloc libre et le second un pointeur de *free list*. Un bloc libre de taille 1 mot mémoire ne va avoir une en-tête composée que d'un mot. La figure 6.4 montre l'architecture d'un bloc libre.

Aspect du premier mot de l'en-tête

1. Le premier bit de poids faible est le bit d'état (ie 1 libre).
2. Les autres bits vont permettre de coder la taille du bloc libre.

C'est l'alignement sur le mot qui peut provoquer la création de blocs libres de taille de 1 mot mémoire. La figure 6.5 en est l'illustration. Nous pouvons voir l'architecture d'un bloc libre à la figure 6.4.

Remarque 1 : Le fait qu'il puisse y avoir des blocs libres de 1 mot mémoire n'est pas un problème puisqu'on va utiliser la fusion des blocs libres contigus.

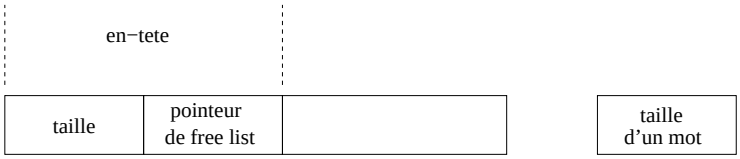

FIG. 6.4 – Architecture d'un bloc libre.

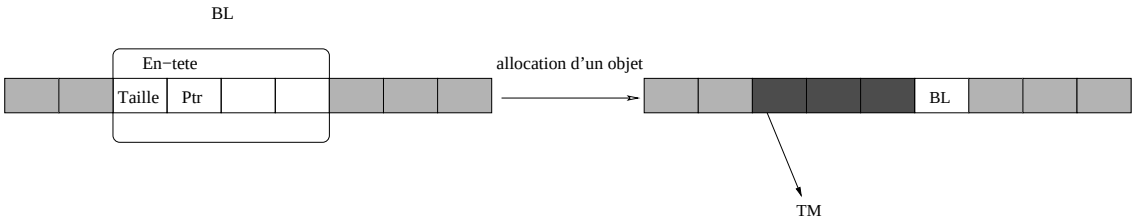

FIG. 6.5 – Implication de l'alignement sur le mot

6.3.3 Architecture des tables de méthodes (TM)

Les tables de méthodes sont rangées les unes à la suite des autres à des adresses alignées sur le mot. Une table de méthodes va contenir des informations pour le GC.

Tables de méthodes classiques

Lors de la phase de marquage, il va falloir savoir si un objet est déjà marqué ou pas. Dans le cas général, un pointeur sur un objet pointe au milieu de l'objet et pas directement sur l'en-tête. Il faut donc pouvoir se rendre sur l'en-tête. Cette information (d-) va s'obtenir dans la table de méthodes. Comme exposé précédemment, la phase de *sweep* va imposer de pouvoir passer de bloc en bloc. Cette information (d+) sera également stockée dans la table de méthodes. Enfin, un tableau d'offsets indique la localisation des pointeurs dans l'objet.

La figure 6.6 montre le détail d'une table de méthodes. d- permet d'obtenir le début de l'objet. d+ permet d'obtenir la fin de l'objet. Le tableau d'offsets permet d'obtenir un pointeur sur chaque champ de type pointeur. Afin de ne pas ralentir l'exécution normale du programme, ce tableau est placé après les données classiques. En effet, le compilateur sait où commence les données classiques (ie au 5ème mot mémoire) alors que si le tableau d'offset est placé avant, cela nécessite un calcul pour connaître le début des données classiques.

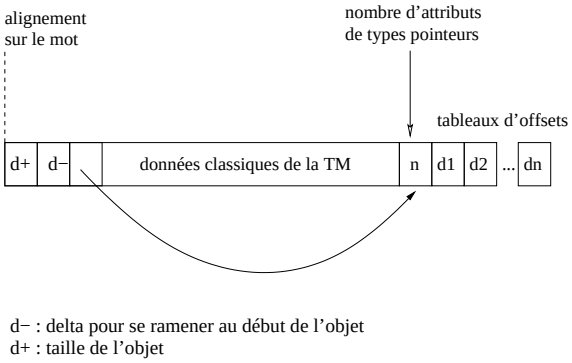

FIG. 6.6 – Aspect d'une table de méthodes

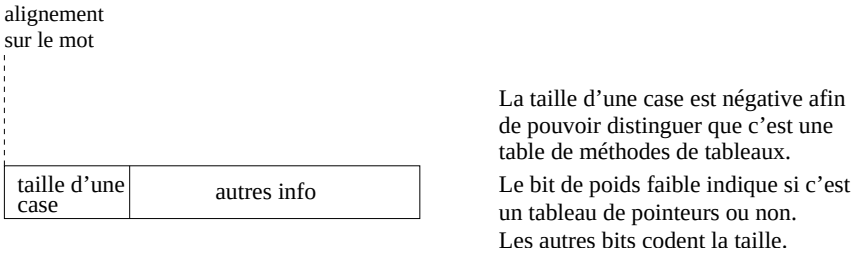

FIG. 6.7 – Table de méthodes de tableaux

Tables de méthodes de tableaux

La taille du tableau étant variable, on ne peut pas la stocker dans la table de méthodes. On l'a stocke donc dans l'objet juste après le pointeur qui pointe sur la table de méthodes. Or le problème est qu'il faut que le GC puisse savoir où récupérer la taille de l'objet. Afin de reconnaître une table de méthodes de tableau, le premier mot va être négatif. D'autre part, il faut pouvoir savoir si le tableau contient des pointeurs ou non. Pour ce faire, on utilise le bit de poids faible. Le premier contient donc la taille d'une case codée avec les autres bits. La figure 6.7 montre l'aspect général d'une table de méthodes de tableau.

6.3.4 les free lists

Nous allons utiliser plusieurs *free lists* par intervalle de taille de bloc libre afin de rendre l'allocation en temps quasi constant. Ces *free lists* vont être construites lors de la phase de *sweep*. Durant cette phase, nous fusionnerons au préalable les blocs libres contigus afin de diminuer la fragmentation du tas. Nous définissons les intervalles suivant les puissances de 2. En d'autres termes, la nième *free list* va couvrir l'intervalle $[2^n; 2^{n+1} - 1]$. Lorsque nous voulons allouer un bloc de taille k, il est facile d'obtenir la *free list* correspondante en récupérant son indice comme suit : $E[log_2(k)] + 1$. Une fois la *free list* récupérée, on utilise un First-Fit pour allouer. Dans le cas où il n'y a pas de possibilités d'allouer l'objet dans la *free list* d'indice e, on essaye dans la *free list* d'indice $e + 1$ et ainsi de suite. Une fois l'objet alloué, il faut insérer le résidu (s'il y en a un) dans la *free list* correspondante (dans le cas où le résidu est de taille strictement supérieure à un mot mémoire) et marquer le bit d'en-tête de la page correspondant. Cependant, ce n'est pas si simple. Lors de la première allocation, il n'y a qu'un seul bloc libre ce qui signifie que toutes les free lists sont à NULL sauf celle-ci. Ceci implique qu'une allocation pour un objet de petite taille va provoquer un parcours séquentiel du tableau d'indices avant de trouver la bonne *free list*. Ceci risque de se produire pour toutes les premières allocations. La solution que nous proposons consiste à ne pas mettre NULL pour une *free list* vide mais l'indice dans le tableau de la première *free list* non vide. On remarquera qu'il est facile de distinguer un petit entier d'une adresse du tas. La figure 6.8 montre notre organisation des *free lists*.

6.3.5 Les invariants

1. Un bloc est soit libre soit occupé.
2. Tout bloc est aligné sur le mot.
3. Un bloc libre est de taille au moins égale à 1 mot mémoire.
4. Un bloc qui suit un bloc libre est un bloc occupé (du à la fusion).
5. Un bloc qui suit un bloc occupé est soit un bloc occupé soit un bloc libre.
6. Le bit de poids faible du premier mot de l'en-tête d'un bloc est le bit d'état. (ie 1 libre ou 0 occupé).
7. Lors de l'exécution du mutateur, le bit de poids fort du pointeur d'en-tête d'un bloc occupé est à 0 et le bit suivant est à 0.

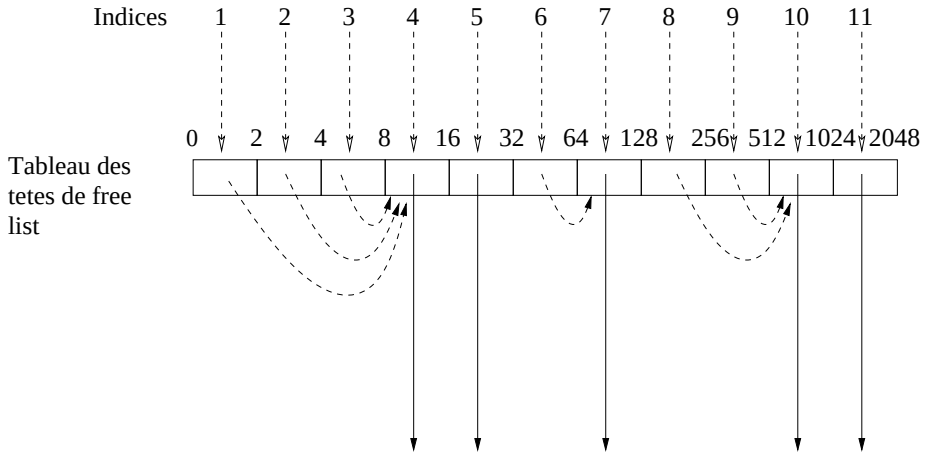

FIG. 6.8 – *free lists*

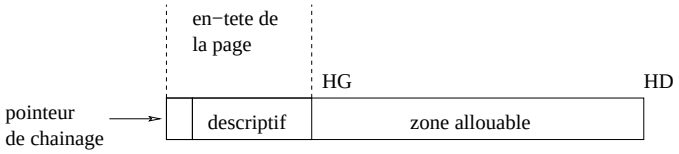

FIG. 6.9 – Architecture d'une page 1

8. Un champ de type pointeur d'un vrai objet pointe (s'il n'est pas NULL) toujours sur un vrai objet.
9. Un vrai pointeur est soit à NULL soit pointe au bon endroit dans l'objet.
10. La plus petite fragmentation possible est de 1 mot mémoire.

6.4 Stratégies où une page peut contenir le plus gros objet du programme

Dans cette partie nous partons de l'hypothèse que tout objet peut être stocké dans une page.

6.4.1 Architecture de base d'une page

Une page va être constituée d'une en-tête et d'une zone allouable. La figure 6.9 montre l'architecture d'une page. L'en-tête se divise en deux parties. La première est un pointeur de chaînage de pages (les pages allouées vont être chaînées entre elles). Quant à la deuxième c'est la partie descriptive[1] de la zone allouable. Ainsi, le premier des mots de l'en-tête descriptive va permettre de décrire les 32 premiers mots de la zone allouable. Le deuxième va décrire les 32 autres et ainsi dessuite. En d'autres termes, l'en-tête va nous permettre de connaître le mot où est un objet dans la page.

Remarque 2 : La première page allouée le sera à une adresse multiple de la taille de la page. Comme on ne fera qu'allouer ou désallouer des pages de même taille, les autres respecterons cet alignement. Cet alignement permet de retrouver la bonne page contenant un objet à partir de l'adresse de celui-ci de façon triviale.

[1] Nous l'appellerons l'en-tête descriptive.

Calcul du nombre de mots descriptifs composant l'en-tête d'une page

Avec un mot d'en-tête, on peut couvrir 32 mots, donc 128 octets. $nb_{mot} = \frac{taille_{page}}{128}$.

Calcul de la zone allouable d'une page

1. $H_D = adresse_{page} + taille_{page}$
2. $H_G = adresse_{page} + nb_{mot} * 4$

Récupération de l'adresse du début d'une page à partir d'une adresse adr alignée sur le mot

$adresse_{page} = E[\frac{adr}{taille_{page}}] * taille_{page}$

Récupération du bit de localisation d'objets correspondant à une adresse adr alignée sur le mot à partir de l'adresse d'une page

L'idée est de récupérer le i^{me} bit de l'en-tête descriptive de la page correspondant à adr, puis d'aller chercher l'adresse du mot d'en-tête dans lequel il se trouve. On récupère le i^{me} bit (on commence à 0) de la façon suivante : $i = \frac{H_G - adr}{4}$. On récupère le j^{me} mot de l'en-tête descriptive (en commençant à 0) comme suit : $j = E[\frac{i}{32}]$. On récupère alors son adresse et la position du bit dans ce mot.

6.4.2 Structure de données permettant de savoir si une page est allouée

Lors de l'exécution du programme, nous allons allouer des pages et en rendre au système. Lors de l'analyse de la pile, nous allons avoir des racines candidates. Il est alors primordial de savoir si une racine candidate pointe ou non à l'intérieur d'une page allouée. Pour ce faire, nous allons utiliser une structure de données qui va couvrir l'ensemble des pages allouables. Nous allons donc nous donner une limite dans l'espace d'adressage virtuel. Cette structure de données va contenir des mots dont chaque bit correspond à une page adressable. Si le bit est à 1, cela signifie que la page est allouée si non il est à zéro. Nous pouvons en voir une représentation graphique à la figure 6.10.

Récupération du bit correspondant à partir de l'adresse d'une page

1. $i = \frac{adr_{page} - H}{taille_{page}}$
2. $j = E[\frac{i}{32}]$

On calcule l'adresse du mot.

6.4.3 L'allocation

L'allocation va se réaliser de la façon suivante. On va d'abord examiner la *free list* adéquate. Si l'allocation ne peut se faire, on fait une demande d'allocation de page. Si le taux de pages allouées est dépassé on déclenche le GC, sinon la page est allouée et est chaînée aux autres.

6.4.4 Détermination d'une racine

1. On regarde si elle correspond à un alignement sur le mot.
2. On regarde si elle se trouve dans les limites du tas.
3. On regarde si elle correspond à une page allouée.
4. On regarde si elle correspond à la zone allouable de cette page.
5. On récupère ensuite le bit de localisation d'objets correspondant et on regarde s'il est à 1 (ie la racine candidate pointe au bon endroit d'un objet).

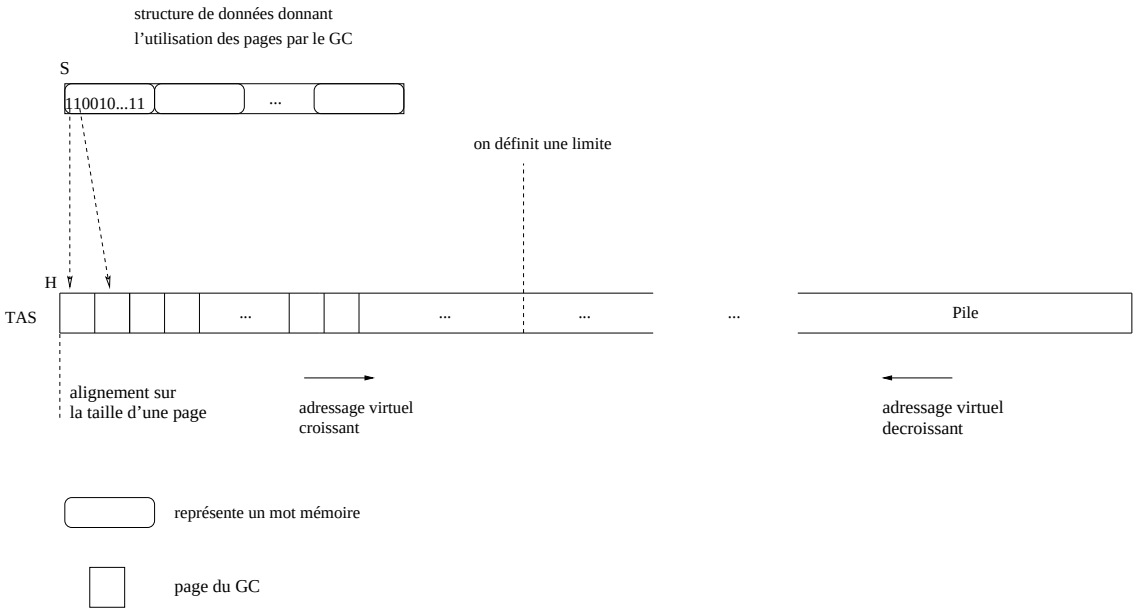

FIG. 6.10 – Structure de données permettant de savoir si une page est allouée

6.4.5 Un simple Mark and Sweep

architecture d'une page

C'est la même que l'architecture de base que l'on a présentée au 6.4.1.

Liste de pages

On a besoin de la liste des pages occupées.

Le marquage

La phase de marquage va commencer par un scannage de la pile afin de déterminer les racines. Chaque mot va être analysé et considéré comme une racine potentielle. Une racine candidate va passer les 4 tests du 6.4.3. Si la racine candidate passe avec succès ces 4 tests, alors elle est considérée comme une racine ; ensuite, on déclenche une phase de marquage en profondeur à partir de celle-ci. Les figures 6.11 et 6.12 illustrent la phase de marquage.

Remarque 3 : La racine qui permet de faire un marquage en profondeur peut être une *"misidentification"*.

Remarque 4 : Lors d'un parcours en profondeur, on ne va pas re-tester à chaque fois si l'objet pointé est un vrai objet. (invariant 9 du 6.3.5).

Récupération du bit de marquage d'en-tête de l'objet

Soit p un pointeur sur cet objet et ptr le mot qui est pointé par p. ptr est donc le pointeur sur la table de méthodes (invariant 9 du 6.3.5). On regarde si ptr correspond à une adresse de TM. Si c'est le cas on va dans la TM et on récupère d- pour se ramener à l'en-tête de l'objet.

Remarque 5 : d- peut être égal à zéro.

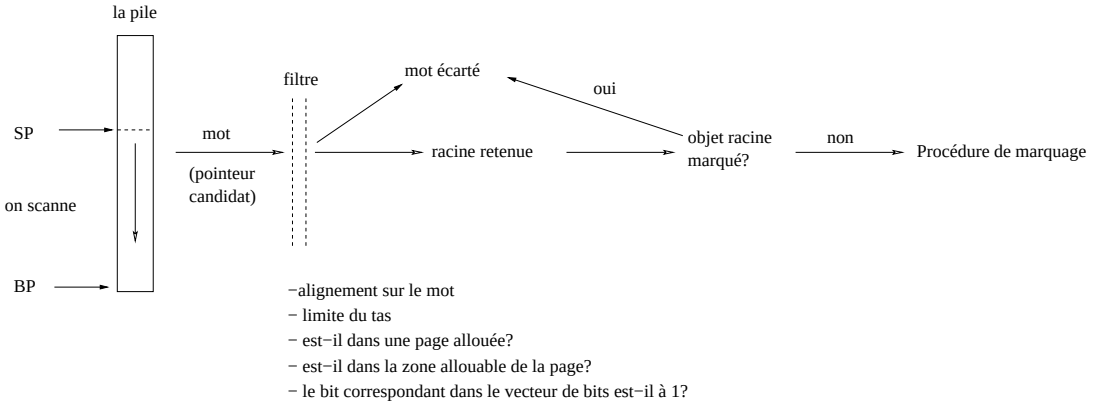

FIG. 6.11 – Détermination des racines et exploration

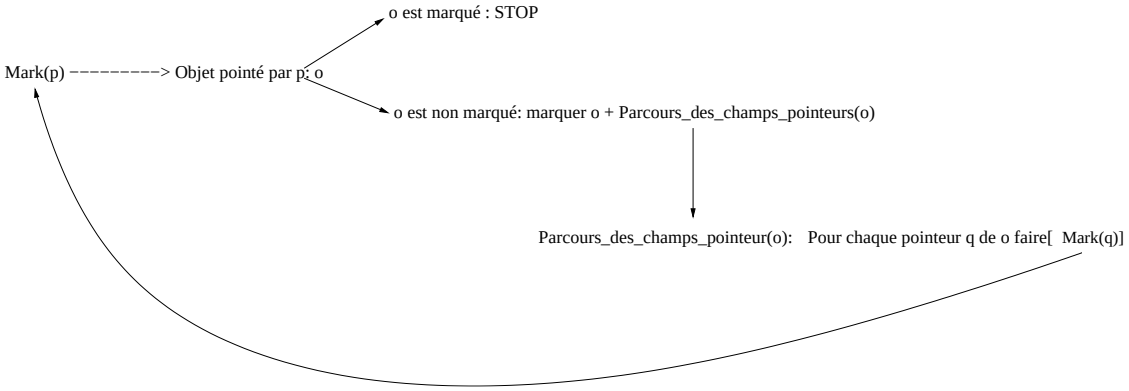

FIG. 6.12 – Marquage

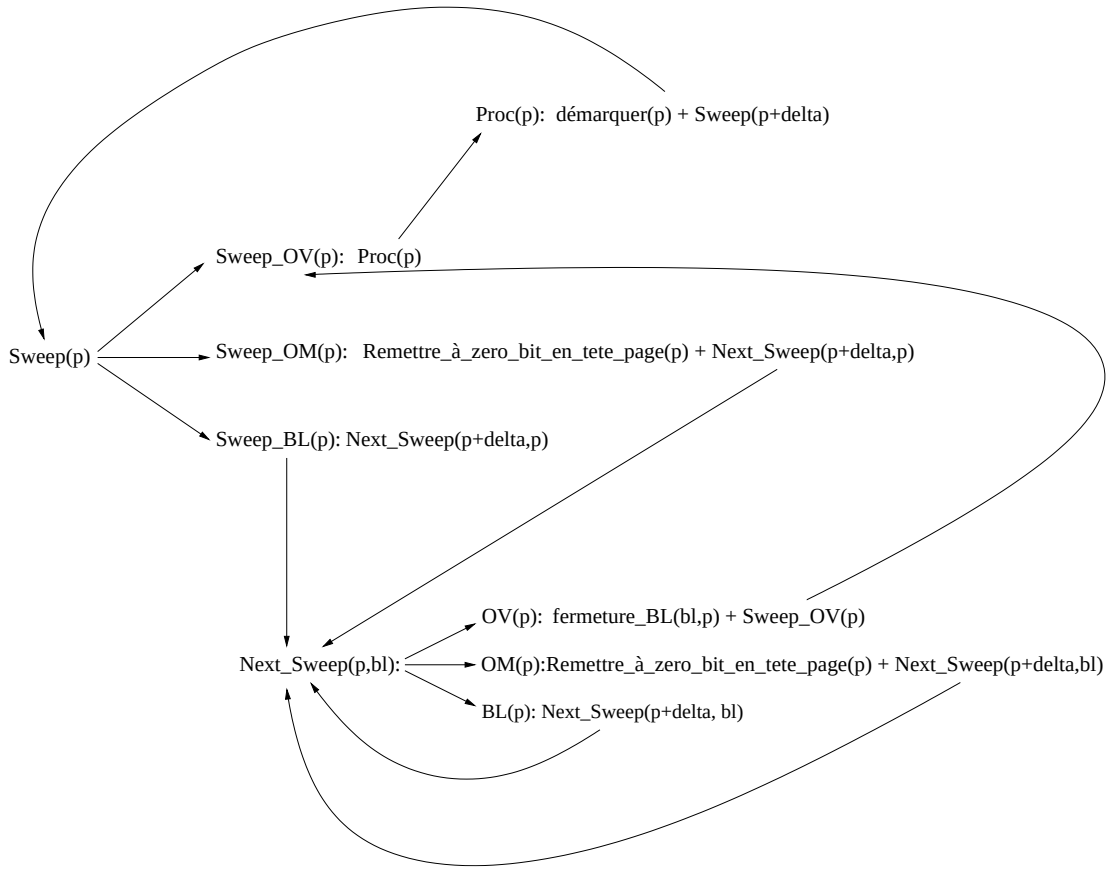

FIG. 6.13 – Modélisation du *sweep*

Remarque 6 : Si ptr ne correspond pas à une adresse de TM c'est parce qu'on est dans le cas particulier où le pointeur d'en-tête se confond avec le pointeur du milieu (ie cas d'un objet sans deltas négatifs ou d'un tableau) et où l'objet est déjà marqué.

Invariant 1 : Tout objet marqué est un vrai objet.

Le balayage

Le balayage s'effectue en scannant le contenu de chaque page allouée. On en profite pour rendre au système les pages qui ne contiennent plus d'objets. La figure 6.13 illustre le *sweep*.

Remarque 7 : On appellera cette version le MS (*Mark and Sweep*).

6.4.6 Un Mostly-Copying Collector hybride pour la coloration bi-directionnelle

Architecture d'une page

On rajoute un deuxième pointeur de chaînage entre le premier et l'en-tête descriptive. Le bit de poids faible du premier pointeur de chaînage indiquera si la page est *«pinned»* ou non. Voir figure 6.14.

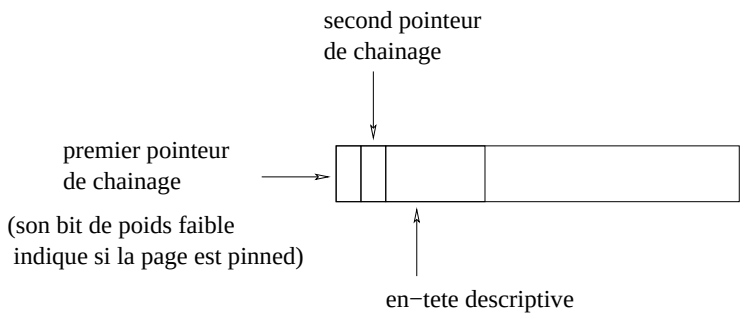

FIG. 6.14 – Architecture d'une page 2

Structures de données utilisées

1. liste des pages copiables.
2. liste des pages non copiables (ie liste des pages *"pinned"*).
3. liste des nouvelles pages. ie utilisées pour la copie.
4. *mark queue* (liste explicite).
5. *cheney queue* (liste implicite).
6. plusieurs *free lists*.

Avant que le GC ne se déclenche, toutes les pages sont dans la liste des pages copiables. La liste est doublement chaînée.

1ère phase

Le GC commence par une phase qui collecte les racines. Dès qu'une racine est identifiée, on *«pin»* sa page que l'on place dans la liste des pages non copiables (elle sort trivialement de la liste des pages copiables). On marque également l'objet comme atteint et on place son adresse dans la *mark queue*. Au final, on obtient la liste des pages non copiables et les adresses des racines se trouvent dans la *mark queue*. Voir figure 6.15.

2ème phase

Tant que la *mark queue* et que la *cheney queue* ne sont pas vides, on extrait un élément de l'une des deux files. On traite alors tous les enfants de cet objet. Pour chaque enfant on regarde d'abord s'il se trouve dans une page *"pinned"*. Si c'est le cas, on regarde s'il est marqué. Si oui, on ne fait rien. Sinon, on le marque et on place son adresse dans la *mark queue*. Si la page n'est pas *"pinned"*, on regarde si l'objet est marqué. Si oui, cela veut dire qu'il a été copié et qu'un *forward pointer* a été inséré juste après l'en-tête. Sinon, on le marque, on le copie et on met un *forward pointer* dans l'original pointant vers la copie. En ce qui concerne la copie, elle s'effectue dans des pages que l'on alloue. En d'autres termes, pour le premier objet que l'on copie, on alloue une page. Pour les autres copies, on les copie dans cette page tant qu'il y a de la place. Dès qu'il n'y a plus suffisamment de place, on créé un bloc libre et on alloue une nouvelle page. A l'issue de cette phase, tous les objets vivants ont été explorés et on obtient la liste des nouvelles pages. Voir figure 6.16.

3ème phase

On part de la liste des pages non copiables et on réalise un *sweep* (avec fusion des blocs libres contigus) dans chacune d'elles. On construit alors les free lists. Voir figure 6.17.

Remarque 8 : Lors de cette phase on démarquera les pages et on démarquera les objets vivants.

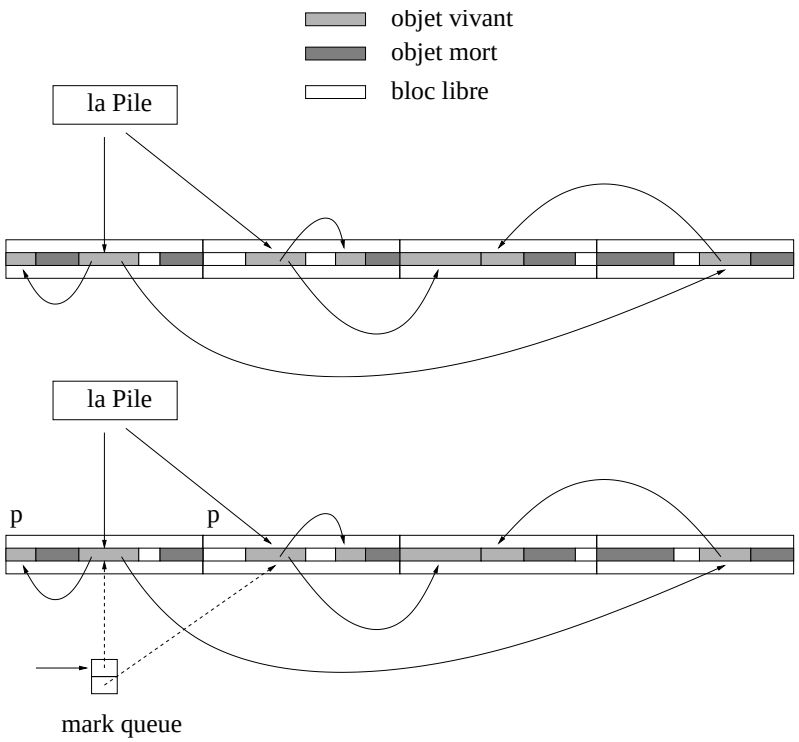

FIG. 6.15 – La 1ère phase

4ème phase

On parcourt la liste des pages copiables et on rend au système chaque page. Une fois ce traitement effectué, il ne reste plus qu'à concaténer la liste des nouvelles pages à la fin de la liste des pages non copiables (ceci se fait par une simple affectation de pointeur). Pour finir, la liste des pages non copiables devient la liste des pages copiables.Voir figure 6.17.

6.5 Stratégies où le plus gros objet peut dépasser une page

6.5.1 Architecture de base d'une page

Une page peut se présenter sous 3 formes différentes.

1. Elle contient des petits objets.
2. Elle peut faire partie d'un gros objet.
3. Elle peut être la première page (page d'en-tête) d'un gros objet. Dans ce cas, on rajoute le nombre de pages utilisées juste après le pointeur de chaînage.

6.5.2 L'allocation

Pour un petit objet, l'allocation se fait de la même manière que dans la version 6.4.5. Par contre, pour un gros objet, on demande au système de nous allouer le nombre de pages consécutives nécessaires pour accueillir l'objet et on chaîne sa page d'en-tête à la liste des gros objets.

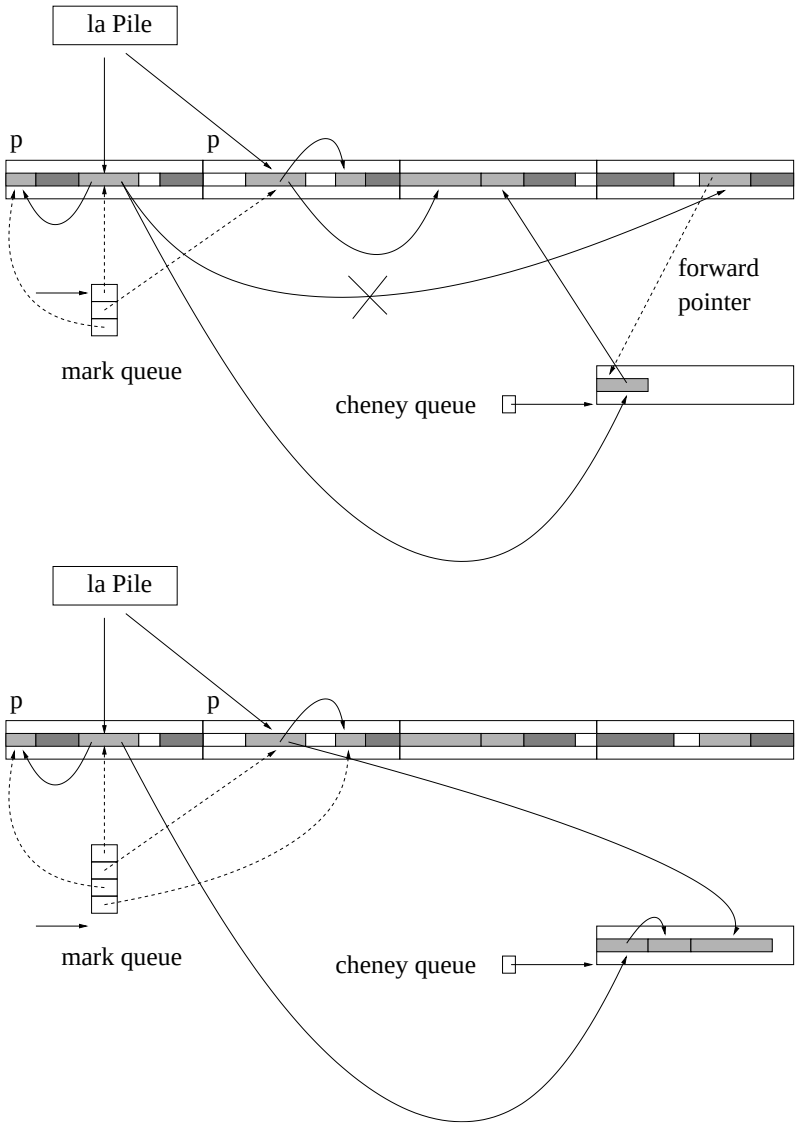

FIG. 6.16 – La deuxième phase

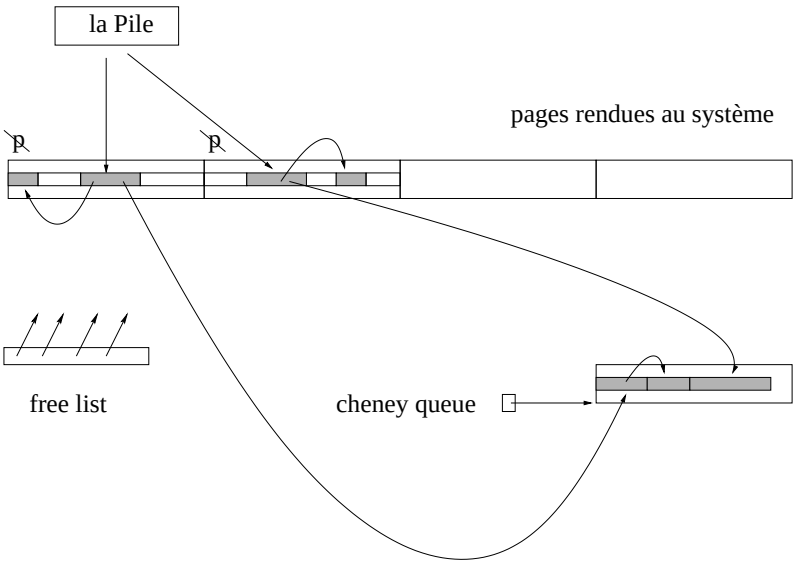

FIG. 6.17 – Phase 3 et 4

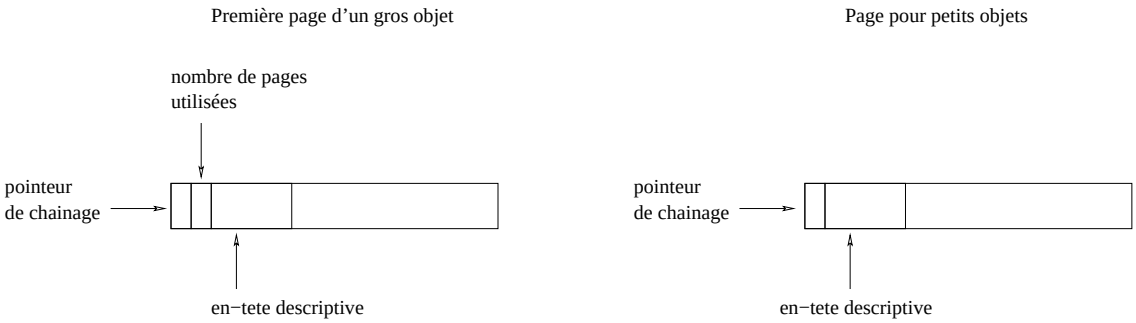

FIG. 6.18 – Architecture des pages 3

6.5.3 Modèle où la partie négative d'un objet peut toujours rentrer dans une page

Nous faisons l'hypothèse que la partie négative d'un objet peut tenir dans une seule page afin que le pointeur sur l'objet pointe dans la page d'en-tête de l'objet. Afin de savoir si une page est allouée, on utilise la même architecture qu'on a présentée au 6.4.2. Cependant le traitement est différent pour un gros objet. Lorsqu'on alloue des pages pour un gros objet, seul le bit correspondant à la première page de l'objet est mis à 1, les autre restant à 0. La figure 6.18 illustre cela.

Le Mark and Sweep

On commence par une phase de marquage qui est identique à la version 6.4.5. Puis on fait un *sweep* comme dans la version 6.4.5. On finit par un parcours de la liste des gros objets. Lorsqu' un objet est non marqué on rend les pages qu'il occupe au système et on le sort de la liste. Lorsqu'un objet est marqué, on le laisse dans la liste et on le démarque.

Le Mostly-Copying Collector hybride

Nous allons copier les petits objets atteints lors de la phase d'exploration se trouvant dans des pages non marquées. Il nous faut donc un moyen de distinguer les pages d'en-têtes des gros

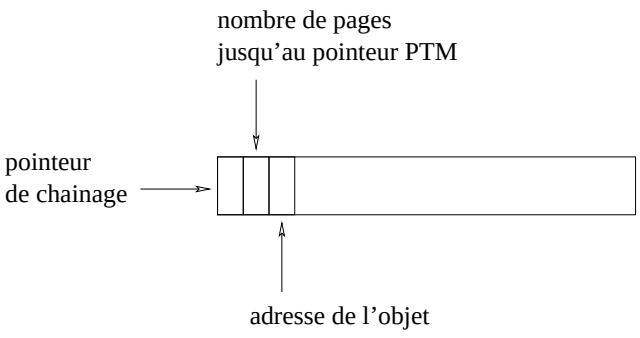

FIG. 6.19 – MSMC2. Page d'en-tête d'un gros objet

objets des autres pages. Comme nous sommes déjà obligés de passer par la table de méthodes pour déterminer si c'est un tableau ou non, le plus simple est alors d'analyser la taille de l'objet afin de savoir s'il rentre ou pas dans une page. On réalise les mêmes phases qu'au 6.4.6 avec un traitement à part pour les gros objets. Lorsqu'on rencontre un gros objet on le marque et on place son adresse dans la mark queue. On rajoute une phase finale dans laquelle on réalise un *sweep* sur la liste des gros objets.

Remarque 9 : Par la suite on appelera cette version le MSMC1 (*Mark and Sweep Mostly-Copying* version 1)

6.5.4 Modèle où la partie négative d'un objet ne se trouve pas forcément sur la première page

Dans cette solution, on s'inspire de la méthode de Boehm en utilisant un tableau d'entiers[2]. Lorsqu'on alloue des pages pour un gros objet, seule la case du tableau correspondante à la page où se trouve le pointeur sur la table de méthodes est modifié. On y met le nombre de pages permettant de se ramener à la première page de l'objet. Cette méthode permet de distinguer de suite que la page fait partie d'un gros objet. Ce qui nous permet de réduire l'en-tête descriptive de la première page à l'adresse de l'objet (voir figure 6.19).

Remarque 10 : Par la suite on appellera cette version le MSMC2 (*Mark and Sweep Mostly-Copying* version 2)

6.6 Conclusion

On s'est placé dans le cas où il n'y a pas d'allocations d'objets conservatifs. Nous avons fait 2 tableaux comparatifs entre Boehm, le MCC et notre MSMC1 et un tableau concernant les cas d'utilisations de nos modèles MS, MSMC1 et MSMC2. Le tableau 6.1 compare les 3 GC au niveau des structures de données utilisées et des *overheads* produits. Le tableau 6.2 fait une comparaison autour de l'allocation. Le tableau 6.3 concerne nos modèles.

Discussion

En ce qui concerne l'allocation, le MCC est le plus rapide puisqu'il utilise une allocation linéaire. Cependant, ce système perd de la place dans certaines pages (les pages qui étaient non copiables lors de la collecte). De notre côté, notre allocation semble tout de même rapide et nous utilisons les trous laissés par le *sweep* pour des allocations futures. Nous combattons le problème

[2] Boehm utilise un GC purement conservatif. Il est obligé de mettre une correspondance (un entier) pour chaque page composant le gros objet vers la page d'en-tête.

	Boehm	MCC	MSMC1
Type de GC	Purement conservatif	Semi-conservatif	Semi-conservatif
modification de haut niveau du compilateur	non	non	oui
Structures de données	Pages contenant des objets de même taille.	Gros blocs contenant des pages qui, elles, contiennent des objets de tailles différentes.	Pages contenant des objets de tailles différentes.
Certificat exact pour les objets	Simple calcul pour trouver le début du bloc dans la page correspondante et vérification qu'un objet est alloué à cet emplacement. (temps constant)	Parcours séquentiel de la page correspondante jusqu'à déterminer si la valeur est bien l'adresse d'un vrai objet. (O(nombre de blocs de la page))	Utilisation du vecteur de bits. (temps constant)
En-têtes de pages	1 mot mémoire pour la taille, 1 mot mémoire pour un pointeur vers une table qui indique (entre autre) quel objets sont alloués et une *bitmask* pour le mark and sweep.	1 mot dans lequel chaque bit a une signification. Par exemple indiquant si la page est pinned, *black listed*, ancienne ...	Un vecteur de bit où chaque bit correspond à 1 mot de la zone allouable de la page.
Overhead dû à la fragmentation	*Overhead* possible s'il y a beaucoup d'objets de taille différente. L'*overhead* sur un emplacement dans la page sera la différence entre la taille stockable et la taille réelle de l'objet que l'on a alloué.	oui dans les pages non copiables.	non
Overhead dû à l'alignement	oui Alignement sur le double mot. Dans le cas où le découpage de la page en bloc de même taille ne tombe pas juste et où le dernier bloc ne peut rester dans la page.	oui pour les gros objets.	oui pour les gros objets.
Overhead (dus au GC) dans les objets	non	1 mot d'en-tête contenant une *bit-mask* qui permet d'identifier les champs de types pointeurs et la taille de l'objet. (pour un gros objet il faudra utiliser plusieurs mots pour les *bit-mask*)	non dans le cas où le delta négatif de la coloration bidirectionnelle est égal à zéro. Dans l'autre cas il est de 1 mot. A noter que la taille ainsi que la localisation des champs de types pointeurs sont stockés dans la table de méthodes de l'objet.

TAB. 6.1 – Tableau récapitulatif des pages.

	Boehm	MCC	MSMC1
Nombre d'allocations élevé	Plus couteûse que le MCC. Il faut sélectionner la bonne *free list.*	Très rapide. Allocation linéaire.	Le système de *free list* par tranche de taille suivant les puissances de 2 permet de calculer la bonne *free list* par un simple calcul. De plus, le fait de pointer sur la *free list* supérieure dans le cas où elle est vide permet de penser que ce système est rapide. (moins que MCC mais plus que Boehm)
Nombre d'allocations d'objets de courte durée de vie élevé	Handicapant car il y a beaucoup d'objets à parcourir lors du *sweep*.	Performant. (grace à la copie)	Performant. (grace à la copie)
Nombre d'allocations d'objets de longue durée de vie élevé	Performant.	Handicapé à cause d'un grand nombre d'objets à copier.	Mêmes problèmes que pour le MCC.
Nombre d'allocations d'objets de taille différente élevé	Provoque des *overhead* dans les pages.	Aucun problème.	Aucun problème.
Beaucoup de pages atteintes par des racines	Pas génant.	Peu de copies donc handicapant pour les prochaînes allocations.	Pas génant.

TAB. 6.2 – Tableau récapitulatif concernant les allocations

	Cas d'utilisations	Les pages
MS	beaucoup d'objets de grande taille.	Très peu de pages, mais utilisation de pages de plusieurs mégas.
MSMC1	Cas où l'on manipule beaucoup de tableaux de grande taille et peu de gros objets non tableaux et où les objets ont une courte durée de vie.	Pages de quelques Ko (multiple de 1024 o).
MSMC2	Cas où l'on manipule des objets non tableaux de très grande taille et où les objets ont une courte durée de vie.	Pages de quelques Ko (multiple de 1024 o).

TAB. 6.3 – Cas d'utilisations de nos modèles.

de la fragmentation par la copie et par la fusion pour les pages non copiables (*sweeping*). Côté technique pour éliminer les faux pointeurs, la technique du MCC n'est pas en temps constant alors qu'avec le vecteur de bit, on est en temps constant et celui-ci ne représente que 3% de la taille de la page. De plus, l'*overhead* du au GC dans les objets peut être réduit à zéro dans notre modèle alors que le MCC est toujours au moins de 1 et peut être plus pour un gros objet. Concernant nos modèles et le traitement des gros objets, les gros objets peuvent provoquer des *overheads* d'alignement puisque la dernière page d'un gros objet peut ne pas être remplie en totalité par ses champs. En outre, il est à noter que dans le MSMC1 et dans MSMC2 qui utilisent la copie, on ne copie pas les gros objets [3] ce qui est un gros avantage.

Perspectives

Une optimisation intéressante côté compilateur serait de tenir compte du fait que le GC n'ajoute pas d'*overhead* dans les objets lorsque le delta est négatif. Ceci en modifiant l'heuristique de coloration. Côté évolution du modèle et de l'implémentation, il serait intéressant d'ajouter le principe de la liste noire afin de faire baisser le taux de «misidentification» et de faire une version générationnelle pour optimiser le temps de collecte.

[3] On rappelle que la version de base du GC par Recopie copie les gros objets ce qui est handicapant dans le cas où ils sont nombreux.

Chapitre 7

Conclusion du mémoire

Nous avons fait un état de l'art des techniques existantes pouvant nous intéresser dans le contexte de notre recherche et nous en avons adapté certaines à la coloration. Il est clair que les techniques de GC purement conservatives marchent pour la coloration. Mais nous avons choisi d'utiliser des techniques semi-conservatives afin d'améliorer l'efficacité du GC et notamment l'adaptation d'un *Mostly-Copying Collector* et d'un *Mark and Sweep*. Le *Mark and Sweep* est utilisé pour traiter les petits objets qui se trouvent dans des pages non copiables et pour traiter les gros objets. La copie est utilisée pour les objets qui se trouvent dans des pages copiables. Lors de l'exécution du mutateur, nous maintenons 2 listes implicites : la liste des gros objets et la liste des pages copiables. Lorsque le GC se déclenche, nous introduisons la liste des nouvelles pages et la liste des pages non copiables. L'invariant principal concernant la gestion de la mémoire est le suivant : les éléments qui se trouvent dans les zones allouables des pages allouées sont soit des objets (blocs occupés) soit des blocs libres. L'*overhead* par objet peut varier de 1 à 2 mots mémoire. En effet, la coloration nous impose d'utiliser un mot d'en-tête afin de pouvoir passer au bloc suivant lors de la phase de *sweep* et un mot pour le pointeur PTM du à la coloration.[1]Cependant, dans le cas où la partie négative de la coloration bi-directionnelle est nulle, le mot d'en-tête et le pointeur PTM sont confondus ce qui permet alors de n'avoir plus qu'un seul overhead dans l'objet. En ce qui concerne les blocs libres, ils ont besoin d'avoir une en-tête de 2 mots mémoire puisqu'ils sont gérés par free lists. Or l'alignement sur le mot des objets peut provoquer la création de blocs libres de 1 mot mémoire. Plutôt que d'aligner tous les objets sur le double mot, nous avons choisi de créer dans ce cas précis des blocs libres de 1 mot mémoire qui ne sont pas gérés par free lists. Nous optimisons leur disparition avec la fusion des blocs libres contigus lors de la phase de sweep. La localisation des pointeurs dans les objets, ne provoque pas d'*overhead* supplémentaire puisque nous utilisons les tables de méthodes pour les localiser. Le certificat exact que nous utilisons pour repérer les objets en mémoire est un vecteur de bits par page qui s'adapte très bien à la coloration. De plus, il ne représente que 3% dans la page ce qui est négligeable. A la fin d'un GC, les pages copiables sont rendues au système et la concaténation des pages non copiables et des nouvelles pages forment les pages copiables. Côté allocation, nous distinguons le traitement des gros objets et celui des petits objets. Pour les petits objets, nous réalisons une gestion de la mémoire par plusieurs *free lists* suivant les intervalles de taille avec une amélioration qui consiste à pointer soit sur le premier bloc libre (si la *free list* est non vide) soit sur l'offset de la prochaine *free list* non vide. Cette méthode a pour but de se rapprocher des performances de l'allocation linéaire tout en gardant une gestion de la mémoire par free list afin de ne pas perdre d'espace dans les pages non copiables. Quant aux gros objets, ils sont alloués à part en demandant simplement au système de nous allouer le nombre de pages contiguës nécessaires. Nous avons fait une implémentation d'un des modèles et nous avons réalisé des simulations aléatoires. Nous avons également dressé des graphiques comparatifs[2] dans des situations particulières. On peut remarquer que dans une

[1] On rappelle que le PTM est le pointeur qui pointe sur la table de méthodes de l'objet et que, dans le cas général, il se trouve au milieu de l'objet.

[2] Voir annexe B.

situation où l'on alloue 50000 objets avec environ 200 racines, où le nombre moyen de pointeurs par objet est de 8, où le nombre de gros objets est de 11000 et le nombre de petits objets de 39000, le nombre de copies maximum obtenu est de 25000 objets et le temps du GC est de 0.55 ce qui est raisonnable. Dans les autres cas, il reste bien en dessous de la demi-seconde. Par ailleurs, on a étudié le pourcentage du temps utilisé par le collecteur par rapport au temps total (allocateur plus collecteur). Il en résulte qu'en moyenne il tourne autour de 10% avec dans le pire des cas, 34% et 0,6% dans le meilleur. Au final, il ne reste plus qu'à intégrer ce GC dans le compilateur Eiffel pour lequel il a été conçu [12] et faire des tests sur de réelles applications afin de le valider.

Bibliographie

[1] O. Agesen and D. Detlefs. Finding references in java stacks. In *OOPSLA'97, Workshop on Garbage Collection and Memory Management*, 1997.

[2] Giuseppe Attardi and Tito Flagella. A customisable memory management framework. In *ProceedProceedings of the 1994 Usenix C++ Conference*, pages 123–142. Usenix Association, April 1994.

[3] J. F. Bartlett. Compacting garbage collection with ambiguous roots. In *Technical Repport, 88/2*. Digital Equipment Corporation Western Research Laboratory, 1988.

[4] H.-J. Boehm. Conservative gc algorithmic overview.

[5] H.-J. Boehm. Space-efficient conservative garbage collection. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI'93)*, ACM SIGPLAN Notices, 28(6), pages 197–206, 1993.

[6] Hans-J. Boehm. Simple garbage-collector-safety. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 89–98. ACM Press, 1996.

[7] Hans-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–100. ACM Press, 2002.

[8] R. Ducournau. La coloration pour l'implémentation des langages à objets à typage statique. pages 79–98.

[9] R. Ducournau. Implementing statically typed object-oriented programming languages. Rapport de Recherche 02-174, L.I.R.M.M., Montpellier, 2002.

[10] Daniel R. Edelson and Ira Pohl. A copying collector for c++. In *USENIX C++ Conference Proceedings*, pages 85–102, 1991.

[11] J. R. Ellis and D. L. Detlefs. Safe Efficient GC for C++. Technical report, Xerox PARC, Palo Alto, CA, 1993.

[12] J. Privat et R. Ducournau. Intégration d'optimisations globales en compilation séparée des langages à objets. In *LMO'04*, pages 61–74. RSTI-L'objet, 2004.

[13] Ceriel J. H. Jacobs H. E. Bal and K. G. Langendoen. *Compilateur*. Dunod, 2000.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. http ://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html, 1999.

[15] P. Maurette. *Assembleur*. MICRO APPLICATION, 20-22, rue des Petits-Hôtels 75010 PARIS, 2003.

[16] C. Queinnec. *Les Langages Lisp*. InterEditions, 1994.

[17] N. Richer. Stratégies de gestion mémoire dans les Mémoires d'Objets Persistantes Automatiques Partitionnés. Thèse sur les systèmes Informatiques de l'Université de Pierre et Marie Curie -Paris VI, 2002.

[18] Frederick Smith and Greg Morrisett. Mostly-copying collection : A viable alternative to conservative mark-sweep. Technical Report TR97-1644, 12, 1997.

[19] Frederick Smith and Greg Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the first international symposium on Memory management*, pages 68–78. ACM Press, 1998.

[20] E. Spir. Etude et Implémentation d'un Glaneur de Cellules Adaptatif pour Lisp, 1989.

[21] P. R. Wilson. Uniprocessor garbage collection techniques. In *Int. Workshop on Memory Management (IWMM'92)*, LNCS 637. Springer Verlag, 1992.

[22] O. Zendra. Traduction et Optimisation Globale dans les Langages de Classes, 2000.

Annexe A

Le problème des faux objets

Présentation

Nous avons vu précédemment que toutes les techniques utilisent des structures de données pour déterminer la localisation d'un objet et que ces techniques provoquent des *overheads* ou bien qu'elles ne sont pas en temps constant. Cependant il est possible d'obtenir un certificat exact des tables de méthodes (ie avoir la certitude qu'un pointeur pointe au début d'une table de méthodes) et un certificat approché (presque exact) des objets. A partir de là, la probabilité qu'un faux objet vérifie le certificat est très faible. La question que nous nous sommes posée est la suivante :

- Est-il possible de se baser sur ce certificat approché pour se passer de tout certificat exact imposant des structures de données supplémentaires ? Nous avons essayé d'appliquer cela à un simple *Mark and Sweep*.

Le problème :

- les faux objets validés par le certificat exact de tables de méthodes qui ont été marqués à tort doivent être démarqués une fois qu'ils ont été reconnus comme des faux objets. Pour ce faire, notre idée a été d'utiliser une procédure duale de marquage (démarquage).

Le problème des faux objets

Les faux objets peuvent entraîner de grosses erreurs lors de la phase de marquage. Supposons que l'en-tête d'un faux objet soit en fait un entier faisant parti d'un tableau. Le marquage va modifier cet entier. La figure A.1 illustre ce propos.

L'idée va être d'introduire une phase de démarquage (marquage dual) afin de rétablir les mots que l'on avait modifiés à tord.

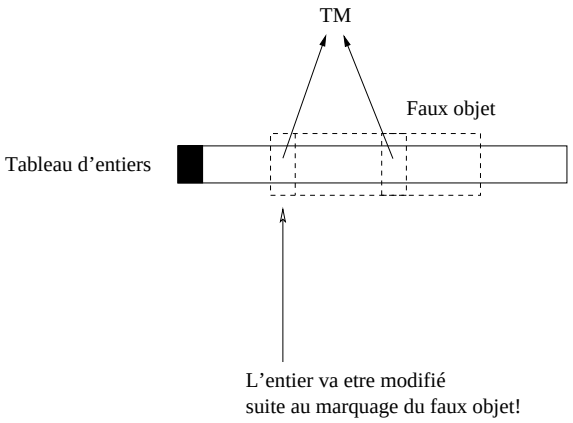

FIG. A.1 – Problème de la modification d'un entier suite à la phase de marquage

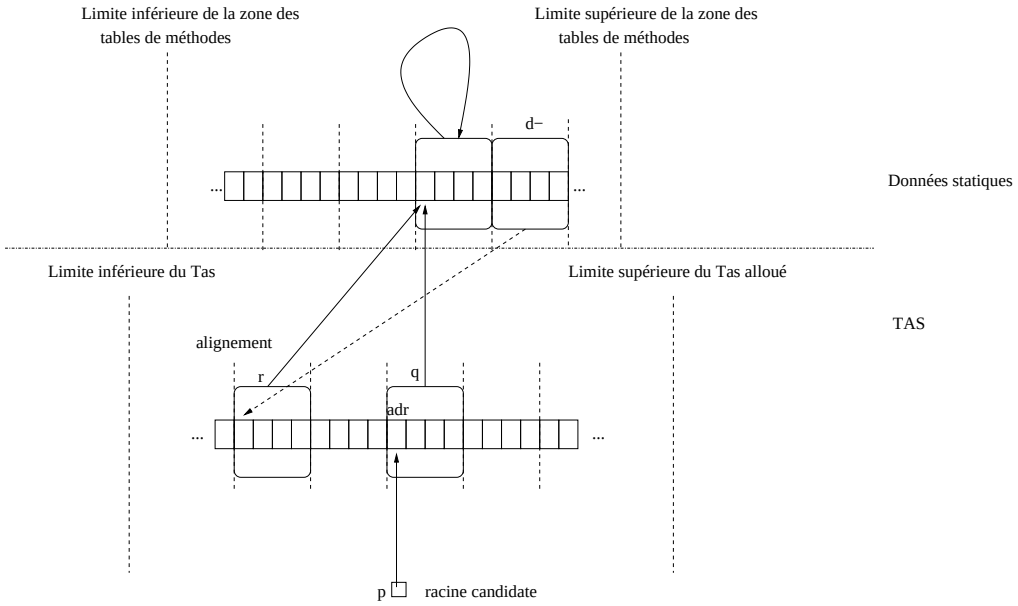

FIG. A.2 – Filtrage d'un pointeur candidat

Certificat exact des tables de méthodes

Les tables de méthodes sont rangées les unes à la suite des autres dans une limite connue et alignées sur le mot avec un pointeur d'en-tête sur lui même.

Certificat approché d'objets (figure A.2)

On regarde si la valeur du pointeur candidat vérifie les points suivants :

1. elle correspond à une adresse alignée sur le mot.
2. elle se trouve dans les limites du tas alloué.
3. le mot qu'elle référence vérifie le certificat des TM.
4. soit r le pointeur obtenu après avoir fait le décalage vers la gauche (d-). r doit pointer sur la même table de méthodes.

Le premier gros problème

Un objet mort devient un bloc libre et le champ suivant l'en-tête de l'objet pointe sur une table de méthodes ou sur un autre objet. Dans la figure A.3, l'objet n'est pas marqué et va donc se transformer en bloc libre. Le problème est que le mot se trouvant juste après le pointeur d'en-tête est un pointeur du faux objet. Le pointeur du faux objet est alors écrasé par le pointeur de free list lors de la construction du bloc libre. L'implication est que la phase duale de marquage ne pourra pas arriver à son terme et ainsi ne pourra pas démarquer tous les faux objets issus (uniquement) de ce pointeur.

Le deuxième gros problème

Le problème est de confondre un mot non pointeur dont le bit sera déjà à 1 avant la phase de marquage avec un mot que l'on marquera pendant la phase de marquage. Le problème se produit lors de la phase duale avec le démarquage du mot.

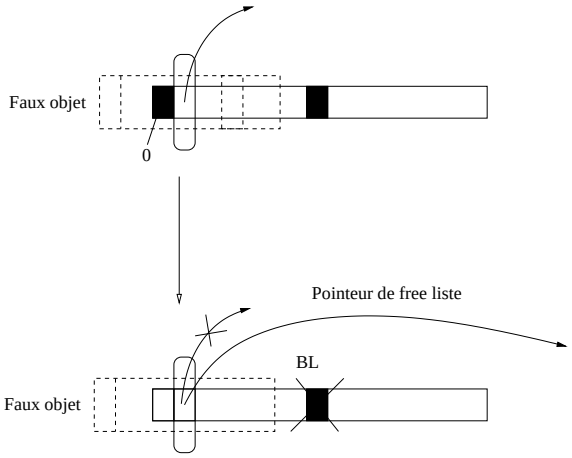

FIG. A.3 – Problème lié à un pointeur d'un objet mort situé juste après l'en-tête de cet objet faisant parti d'un faux objet

Discussion

Nous avons essayé plusieurs solutions pour lutter contre le problème 1 et le problème 2 mais elles se sont révélées impossibles à cause d'autres problèmes ou alors trop coûteuse en temps. Ceci confirme en partie, le fait que ce genre de solutions ne sont pas abordées dans la littérature.

Annexe B

L'implémentation

Présentation

L'implémentation est celle du modèle MSMC2. Nous avons fait des tests en créant des tables de méthodes et en allouant des objets (de l'ordre de 50000) avec un mode aléatoire. Nous avons vérifié que la somme du nombre de petits objets atteints, du nombre de gros objets atteints et du nombre d'objets copiés était égale au nombre d'objets vivants en mémoire. On a vérifié également que tous les objets vivants étaient démarqués, que les pages qui restaient étaient *«dé-pinned»* et que les invariants du 6.3.5 était vérifiés[1].

Expérimentation

Dans cette expérimentation, nous comparons le nombre d'objets copiés par rapport au temps CPU pris par le collecteur. Pour ce faire nous commençons par fixer certains paramètres :

1. Nombre de gros objets alloués : environ 11000.
2. Nombre de petis objets alloués : environ 39000.
3. La taille moyenne des objets (gros et petits confondus) : 250 mots.
4. Nombre de pages moyen par gros objet : 4.
5. Nombre de gros objets morts moyen : 7500.

Nous dressons 2 courbes, la première (figure B1) est le cas où il y a 4 pointeurs par objet et la deuxième (figure B2) est le cas où il y a 8 pointeurs par objet. La première peut être considérée comme linéaire. Le temps de collecte monte au maximum à 0.35 secondes pour 15000 objets copiés et reste principalement inférieur à 0.25 secondes dans les autres cas. La deuxième est aussi assimilable à une courbe linéaire. Dans le pire des cas, le temps de collecte monte jusqu'à 0,55 secondes pour 25000 objets copiés. En recoupant les 2 courbes, on s'aperçoit que le temps de collecte dépend fortement du nombres d'objets copiés ce qui confirme ce qu'on a discuté sur les GC qui utilisent la copie[2].

Pourcentage du temps utilisé par le collecteur

Il s'avère que sur d'autres séries de tests, le pourcentage maximum du temps utilisé par le collecteur est environ de 34% du temps total du programme et au minimum de 0,6%. Le pourcentage moyen tourne autour de 10% ce qui est un bon chiffre pour un GC[22].

[1] Pour calculer le nombre d'objets vivants en mémoire après un GC, on parcourt chaque page copiable (on rappelle qu'à la fin d'un GC les pages vivantes sont mises dans la liste des pages copiables) et compte le nombre d'objets vivants.

[2] Un GC qui pratique de la copie est performant lorsque la durée de vie des objets est très courte puisqu'on a peu d'objets à copier.

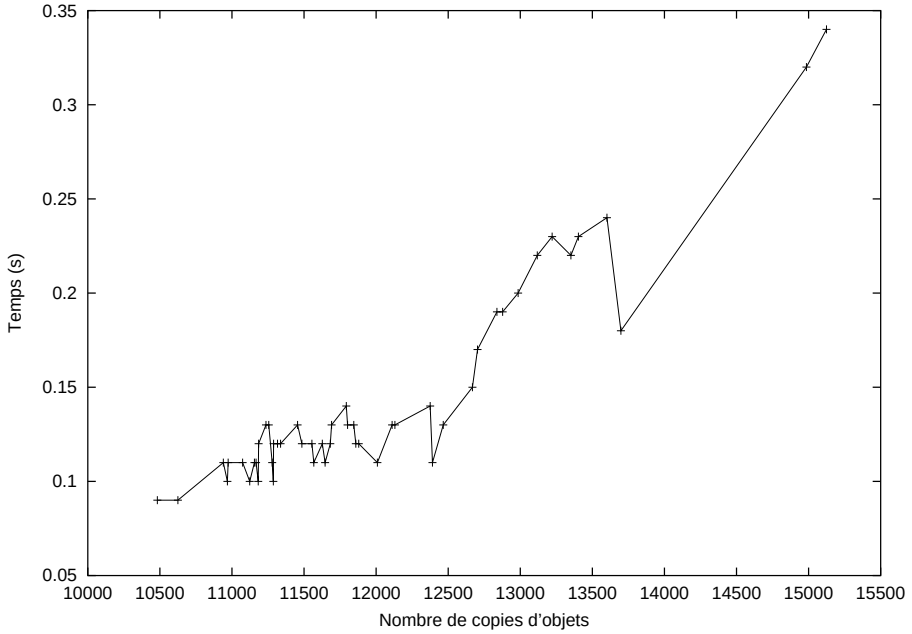

FIG. B.1 – Tests avec 4 pointeurs par objet

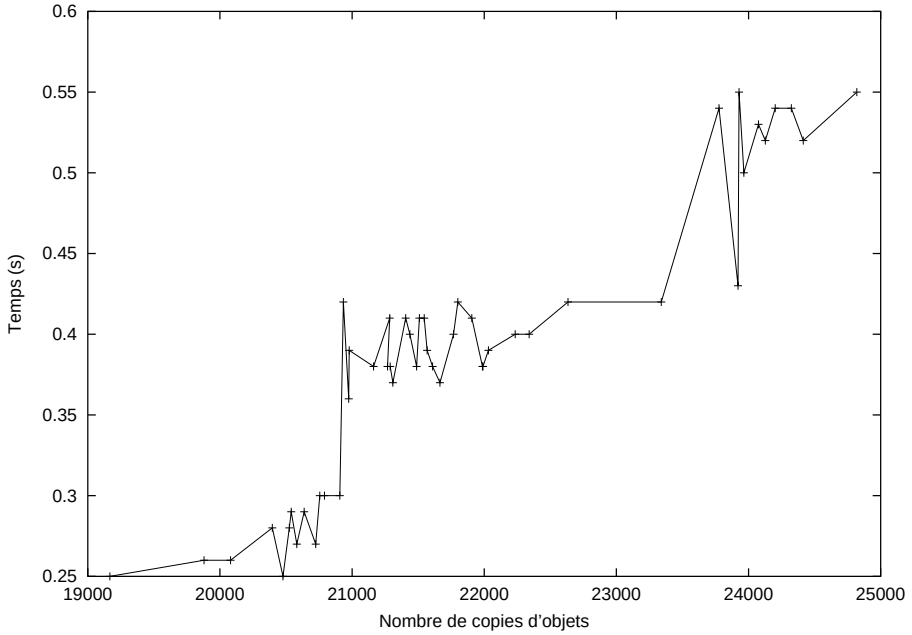

FIG. B.2 – Tests avec 8 pointeurs par objet