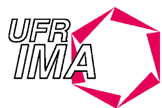

Gestion hybride de la mémoire dynamique dans les systèmes Java temps-réel

Nicolas Berthier

Stage de Magistère M1 encadré par Christophe Rippert et Guillaume Salagnac

Laboratoire Vérimag

Septembre 2007

Résumé : Le langage Java offre un avantage considérable en terme de facilité d'utilisation : la gestion transparente de la mémoire dynamique, basée sur l'emploi de ramasse-miettes. Or, les algorithmes couramment utilisés pour implanter ces mécanismes introduisent des temps de pause dans l'exécution des programmes. Le travail de l'équipe dans laquelle j'ai fait mon stage consiste en l'implantation d'un mécanisme de gestion de la mémoire en régions basé sur une analyse statique sur-approximant les durées de vie des objets. Cependant, cette analyse n'est pas efficace sur certains types de programmes, pour lesquels elle engendre des fuites de mémoire pouvant conduire à la saturation de l'espace mémoire des applications. Après une étude des techniques de gestion de la mémoire dynamique dans les systèmes temps-réel et un ensemble d'études de cas, nous avançons deux solutions à ce problème, en utilisant notamment un ramasse-miettes par comptage de références. L'évaluation de nos solutions, après implantation dans une machine virtuelle Java, montre qu'elles permettent d'éviter les fuites de mémoires pour certains types de programmes problématiques.

Mots-clés : temps-réel, ramasse-miettes, gestion de la mémoire en régions

Je remercie Guillaume SALAGNAC *et Christophe* RIPPERT
pour leur accueil et l'aide qu'ils m'ont apportée
pendant la réalisation de mon stage.

Table des matières

I. Introduction **4**
I.A. Contexte général . . . 4
I.A.1. L'environnement du stage. . . . 4
I.A.2. Java et les systèmes embarqués temps-réel. . . . 4
I.A.3. Le travail de l'équipe. . . . 4
I.B. Problématique . . . 5
I.C. Méthodologie . . . 5

II. La gestion de la mémoire dynamique dans les systèmes temps-réel **7**
II.A. Les ramasse-miettes temps-réel . . . 7
II.A.1. Les algorithmes de ramasse-miettes classiques. . . . 7
II.A.2. Le comptage de références temps-réel. . . . 9
II.A.3. Une solution basée sur le ramasse-miettes par marquage-balayage. . . . 9
II.B. Analyses statiques et gestion de la mémoire en régions . . . 10
II.B.1. La gestion de la mémoire en régions. . . . 10
II.B.2. L'utilisation manuelle des régions. . . . 11
II.B.3. Les analyses statiques pour la gestion de la mémoire en régions. . . . 11

III. Présentation des études de cas **13**
III.A.Analyse du programme Health . . . 13
III.B.Analyse du programme Voronoï . . . 17
III.C.Analyse du programme BH . . . 20
III.D.Analyse du décodeur MP3 JLayer . . . 21

IV. Contributions **23**
IV.A.Contributions théoriques . . . 23
IV.A.1.Séparation des phases d'exécution . . . 23
IV.A.2.Gestion de la mémoire en régions et ramasse-miettes par comptage de références 23
IV.B.Contributions pratiques . . . 24
IV.B.1.Outil d'étude de la démographie des objets d'une application . . . 24
IV.B.2.Implémentation d'un ramasse-miettes par comptage de références . . . 24
IV.B.3.Outil de photographie du tas . . . 25
IV.B.4.Mesures d'occupation et de fragmentation du tas . . . 25

V. Validation expérimentale **27**
V.A. Résultats pour le décodeur MP3 JLayer . . . 27
V.B. À propos de Health et Voronoï . . . 28
V.C. Résultats pour le programme BH . . . 28

VI. Conclusions et perspectives **31**

Bibliographie **32**

I. Introduction

I.A. Contexte général

I.A.1. L'environnement du stage.

Le laboratoire Vérimag est une unité de recherche de la fédération IMAG[1] sous la tutelle du CNRS[2], de l'université Joseph Fourier et de l'INPG[3]. Les travaux qui y sont menés concernent principalement les systèmes et logiciels embarqués (langages synchrones, vérification et test de programmes, modélisation, etc.).

L'équipe dans laquelle j'ai effectué mon stage travaille plus particulièrement sur les aspects théoriques et pratiques de la conception de systèmes et logiciels critiques (temps-réel, embarqués, etc.). Une des composantes de ces travaux porte notamment sur les systèmes Java temps-réel.

I.A.2. Java et les systèmes embarqués temps-réel.

Bien qu'il ait été initialement destiné au développement de programmes utilisant peu de ressources (*e.g.* applets web), le langage Java est aujourd'hui bien plus utilisé pour concevoir des applications pour serveurs ou stations de travail, que pour réaliser des programmes ou des systèmes pour les plates-formes contraintes. Les différentes implantations de l'environnement d'exécution Java standard nécessitent en général des quantités de ressources (*i.e.* mémoire, puissance de calcul, etc.) très supérieures à ce qui est disponible dans les systèmes embarqués contraints comme les cartes à puce ou les capteurs par exemple.

En outre, les systèmes temps-réel imposent des contraintes de temporisation liées au calcul des temps d'exécution au pire cas des différents processus à ordonnancer. Or, la gestion de la mémoire dynamique en Java standard est basée sur l'utilisation d'un ramasse-miettes, ce qui a l'avantage de simplifier considérablement le travail du développeur d'applications, mais pose des problèmes en ce qui concerne les calculs des temps d'exécution (temps de pause de périodicité et de durée non-déterministes liés à l'exécution du ramasse-miettes). Des solutions alternatives ont été suggérées, mais elles ne sont en pratique pas satisfaisantes pour le programmeur : certaines proposent par exemple un mécanisme de désallocation programmée [1], ou encore la suppression de l'allocation dynamique [2].

I.A.3. Le travail de l'équipe.

L'équipe au sein de laquelle j'ai effectué mon stage travaille à l'implantation de mécanismes d'allocation en régions dans une machine virtuelle Java standard destinée aux systèmes embarqués (JITS – pour *Java In The Small* [3]). Le principe de la gestion de la mémoire en régions est basé sur un regroupement des objets selon leur site d'allocation et leur cycle de vie, caractéristiques déterminées (ou plutôt sur-approximées) par une analyse statique du programme [4, 5]. Chaque groupe d'objets ainsi créé constitue une région qui sera alors allouée et désallouée d'un seul bloc à des points déterminés de l'exécution du programme. Les avantages principaux de cette technique sont, d'une part de permettre en théorie de se passer du ramasse-miettes et d'autre part d'implanter un mécanisme de gestion de la mémoire dont les primitives d'allocation et de désallocation s'exécuteront en des temps fixes.

[1] Institut d'Informatique et Mathématiques Appliquées de Grenoble
[2] Centre National de la Recherche Scientifique
[3] Institut Polytechnique de Grenoble

I.B. Problématique

Cependant, la technique d'analyse statique utilisée ne permet pas de rendre le mécanisme d'allocation en régions «autonome» dans tous les cas. En effet, si pour certains programmes ce système gère correctement l'ensemble des objets alloués (*cf* figure I.1 – la courbe en pointillés décrit la variation de l'occupation mémoire lorsque celle-ci est gérée avec un ramasse-miettes ; la courbe pleine représente cette valeur avec l'utilisation des régions), pour d'autres en revanche, l'occupation de la mémoire ne cesse d'augmenter : une proportion de plus en plus importante de l'espace est occupée par des objets *morts* (*i.e.* qui ne sont plus référencés), mais que le système actuel ne libère pas (*cf* figure I.2 page suivante).

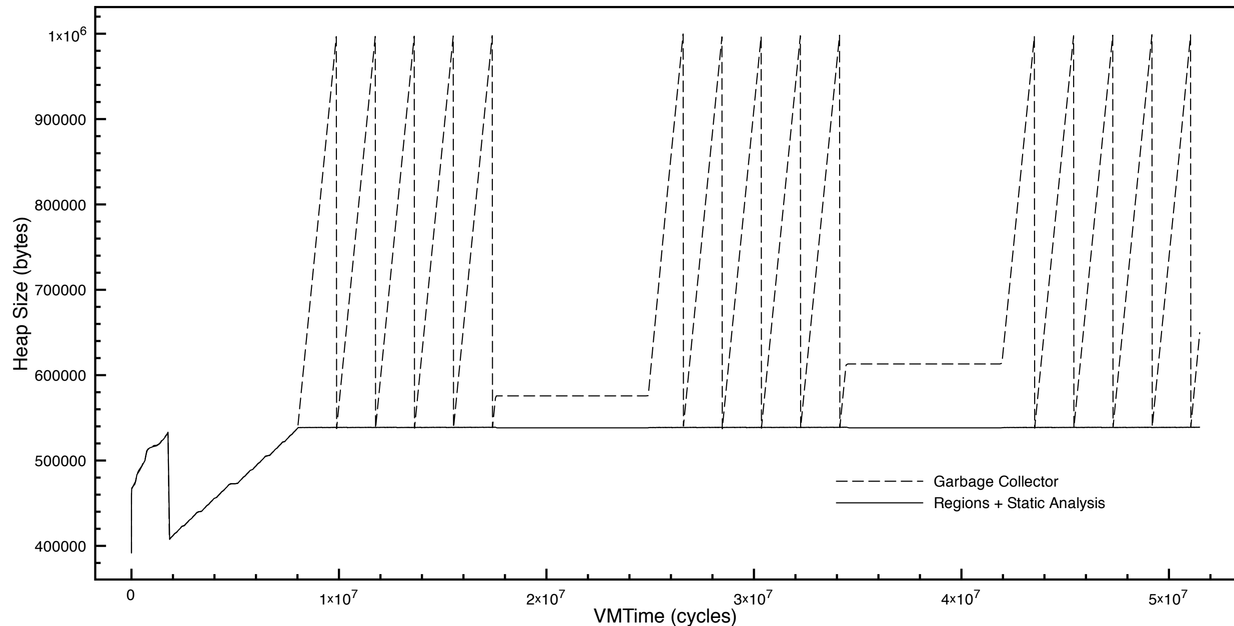

FIG. *I.1: Occupation mémoire pour un programme géré correctement par le mécanisme d'allocation en régions actuel : l'application s'exécute dans un espace mémoire de taille presque constante.*

Le travail à réaliser consiste donc à rechercher une solution permettant de compléter la gestion de la mémoire en régions actuellement implantée de manière à gérer correctement ces applications problématiques.

I.C. Méthodologie

Dans le but d'apporter une réponse au problème exposé précédemment, le travail réalisé est réparti en plusieurs phases.

La première partie présentera une étude des différentes techniques de gestion de la mémoire dynamique dans les systèmes temps-réel. Cet état de l'art permet d'énumérer les divers aspects liés à ce domaine : les problèmes récurrents et leurs solutions, les avantages et inconvénients de telle technique ou de telle autre, etc.

Dans un second temps, une étude approfondie de différents programmes problématiques sera effectuée afin d'expliquer les problèmes rencontrés par la méthode actuelle et ainsi de déterminer quelles adaptations du gestionnaire de mémoire en régions ou encore de l'analyse statique actuelle pourraient améliorer le comportement de ces applications particulières.

Enfin, une solution hybride combinant la gestion de la mémoire en régions actuellement proposée par l'équipe et l'utilisation de ramasse-miettes adaptés aux contraintes des systèmes temps-réel sera

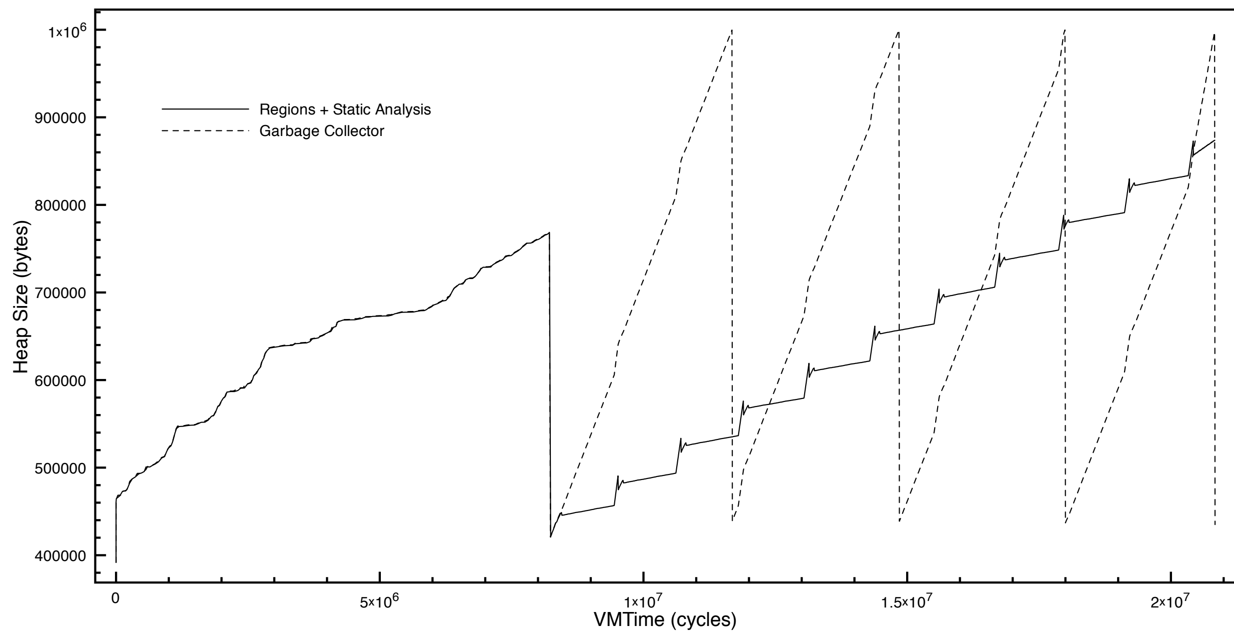

Fig. *I.2: Occupation mémoire pour un programme problématique : une portion de la mémoire n'est jamais récupérée.*

avancée, en tenant compte des résultats des études précédentes. L'objectif de cette phase de travail est de déterminer une solution réaliste adaptée au problème qui nous concerne. Cette nouvelle technique sera finalement implémentée dans JITS, puis évaluée grâce à des expérimentations sur plusieurs applications.

II. La gestion de la mémoire dynamique dans les systèmes temps-réel

Le problème de la gestion de la mémoire dynamique dans les systèmes temps-réel a déjà fait l'objet de nombreuses recherches et un grand nombre de solutions ont déjà été proposées. L'étude de celles-ci permet de mieux appréhender les avantages et inconvénients de chacune, afin de cerner dans quelle mesure elles pourraient être utilisée ou adaptée dans le cas qui nous intéresse.

Il existe différents types de solutions au problème de la gestion de la mémoire dynamique dans les systèmes temps-réel. Parmi celles-ci, deux grandes familles peuvent être observées. Un premier ensemble d'approches regroupe les méthodes consistant à adapter des ramasse-miettes classiques de manière à satisfaire les contraintes de ce type de système. La seconde famille est celle des solutions utilisant des analyses statiques et la gestion de la mémoire en régions.

Cette étude des solutions existantes se focalisera dans un premier temps sur les ramasse-miettes temps-réel. Les approches employant des analyses statiques et la gestion de la mémoire en régions seront étudiées ensuite.

II.A. Les ramasse-miettes temps-réel

La gestion automatique de la mémoire dynamique peut être assurée à l'exécution par des mécanismes dits de *ramasse-miettes*. Leur rôle est de récupérer la mémoire occupée par les objets dits *morts* ou *inaccessibles*.

Après une rapide présentation des principaux algorithmes de ramasse-miettes, deux adaptations aux systèmes temps-réel seront présentées.

II.A.1. Les algorithmes de ramasse-miettes classiques.

Il existe plusieurs classes d'algorithmes principales permettant la gestion automatique de la mémoire dynamique [6]. Le ramasse-miettes par comptage de références, puis celui par marquage-balayage et, enfin, celui par recopie seront décrits.

Le ramasse-miettes par comptage de références.

La technique dite du «comptage de références» consiste à associer à chaque objet un compteur, représentant le nombre de références existantes sur cet objet. Le compteur d'un objet est incrémenté lors de l'apparition d'une nouvelle référence sur cet objet ; il est décrémenté lorsqu'une de ces références disparaît : si le compteur devient nul, l'objet n'est plus référencé et la mémoire qu'il occupe peut alors être récupérée.

Cet algorithme présente l'avantage d'être très simple à mettre en œuvre : il suffit pour cela d'ajouter un champ à tous les objets, et de détecter l'apparition, la modification, ou la suppression d'une référence (une *barrière en écriture* est une portion de code s'exécutant à chaque écriture de référence ; le surcoût occasionné par ce genre de mécanisme peut généralement être très réduit [7]). De plus, il peut permettre une allocation en temps prédictible, puisque la mémoire est récupérée dès la mort d'un objet. En outre, la libération se faisant récursivement, elle a un coût d'exécution au pire cas proportionnel au nombre d'objets du tas (libération de tout son contenu). Une adaptation présentée plus loin propose une réduction de ce coût.

En revanche, il présente une limitation importante : il est inopérant sur les données cycliques. La figure II.1 illustre cet inconvénient majeur, qui impose alors d'associer ce type de ramasse-miettes avec un autre mécanisme, destiné à libérer la mémoire de ce genre de structures. En outre, cette

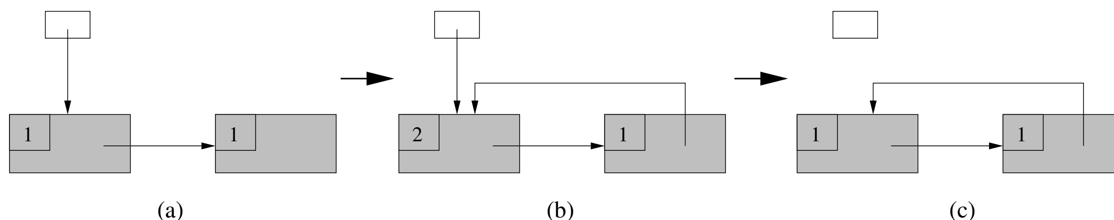

Fig. *II.1: Illustration du problème de la gestion des cycles par un ramasse-miettes de type comptage de références. Deux éléments d'une structure cyclique ont initialement leur compteur à 1 (ils ne sont référencés qu'une seule fois chacun) (a). Le cycle est alors construit : le premier élément est référencé deux fois (b). Enfin, l'unique pointeur sur la structure complète disparaît, mais les compteurs ne sont pas nuls (c) : la place occupée par les deux éléments est définitivement perdue.*

méthode de récupération de la mémoire n'empêche pas la fragmentation du tas : une allocation peut échouer alors que la quantité de mémoire libre est suffisante, mais qu'elle est trop fragmentée. Enfin, la place occupée en mémoire par le compteur peut avoir un impact sur l'occupation de la mémoire, ressource souvent rare dans les systèmes qui nous intéressent.

Le ramasse-miettes par marquage-balayage.

Une autre technique consiste à exécuter un algorithme de libération de mémoire lorsqu'une tentative d'allocation échoue (*i.e.* il n'y a plus de zone de mémoire inoccupée de taille suffisante pour contenir le nouvel objet).

Un ramasse-miettes par marquage-balayage consiste à différencier l'ensemble des *objets accessibles* du reste du tas. Il s'effectue à partir d'un état initial dans lequel aucun élément n'est marqué. Ces objets sont alors parcourus et marqués récursivement à partir des références dites *racines* (pile d'exécution, etc.). Enfin, ceux qui n'ont pas été visités sont considérés comme inaccessibles, et la mémoire qu'ils occupent est récupérée.

Contrairement à l'algorithme par comptage de références, celui-ci ne nécessite pas de barrière en écriture. Il permet de plus la récupération des structures cycliques. Le surcoût en taille mémoire nécessaire peut également être réduit, puisqu'un seul bit suffit à stocker l'information de marquage d'un objet.

En revanche, le temps au pire cas de l'allocation devient beaucoup plus important, dans la mesure où il nécessite au moins deux parcours de l'ensemble des objets. Ce genre de mécanisme peut alors occasionner des pauses de durées considérables pendant l'exécution d'un programme. Enfin, il n'empêche pas la fragmentation de la mémoire.

Le ramasse-miettes par recopie.

L'algorithme de ramasse-miettes par recopie est basé sur un partitionnement du tas en deux zones de tailles égales. Les allocations se font en incrémentant un pointeur dans une première zone (donc en temps constant), jusqu'à rencontrer la fin de celle-ci. L'algorithme se déclenche alors, parcourant les objets de la première zone et les recopiant dans la seconde. Enfin, le rôle des deux zones est interverti.

Un inconvénient à cette recopie est la nécessité d'utiliser des *barrières en lecture* (*i.e.* mécanisme d'indirection exécuté lors de chaque lecture d'une référence), qui ont pour particularité d'être assez coûteuses en temps d'exécution. En effet, le déplacement des objets par le ramasse-miettes pendant l'exécution du programme implique au moins un degré d'indirection lors de la lecture

d'une référence. Hormis cet aspect, cet algorithme présente à peu près les mêmes propriétés qu'un ramasse-miettes par marquage-balayage. Il élimine toutefois la fragmentation de la mémoire, en contrepartie d'un doublement de l'espace mémoire nécessaire à l'exécution d'un même programme.

Des versions améliorées de ces différents mécanismes existent déjà, mais les avancées sur un point se font souvent au détriment d'un autre. Par exemple, Hudson et Moss [8] ont proposé un algorithme par recopie réduisant la taille de la mémoire supplémentaire nécessaire, mais cet algorithme devient très peu efficace s'il doit traiter de grandes structures cycliques.

Parmi les différents ramasse-miettes temps-réel étudiés, deux seront décrits en détail : un premier basé sur le principe du comptage de références, et un second qui est une solution combinant d'autres ramasse-miettes classiques. En effet, ils ont paru regrouper l'ensemble des aspects liés au type de solution recherché.

II.A.2. Le comptage de références temps-réel.

Ritzau et Fritzson [9] proposent d'utiliser un ramasse-miettes basé sur le principe du comptage de références. Comme nous l'avons vu précédemment, un tel algorithme est assez simple, mais ne gère pas la fragmentation de la mémoire, ni les structures de données cycliques. Le coût de la libération peut de plus devenir trop important pour les systèmes que l'on considère. Les adaptations proposées visent principalement à éliminer ces problèmes.

Cette solution propose de diviser tous les objets en blocs de même taille. Cela a pour avantage de simplifier la gestion de l'espace mémoire. Il existe différentes techniques pour connecter les divers blocs d'un même objet (liste chaînée de blocs, indexation, etc.), mais il est en revanche nécessaire que l'accès à un attribut ou à un élément d'un tableau se fasse en un temps toujours prédictible. *e.g.* les tableaux sont ici gérés comme des arbres de blocs : l'accès à un élément se fait alors en un temps logarithmique (par rapport à la taille du tableau). Enfin, on peut noter que Siebert [10] a déterminé qu'une taille de bloc de 64 octets est généralement un bon choix pour la majorité des applications Java.

De plus, en ce qui concerne le coût au pire cas de la libération, la parade employée est l'utilisation d'un ramasse-miettes par comptage de références dit *différé*. Il consiste à ajouter les objets à libérer dans une liste, dont les premiers éléments sont traités lorsqu'une allocation échoue.

Par ailleurs, la division des objets élimine les problèmes liés à la fragmentation externe non gérée par les ramasse-miettes de ce type : la perte de mémoire se fait dans les zones occupées, lorsque le dernier bloc d'un objet est trop petit pour remplir une zone complète. Ce déplacement de problème peut cependant être considéré comme une avancée, dans la mesure où la fragmentation interne de la mémoire est, contrairement à la fragmentation externe, prédictible. On peut effectivement prévoir ses conséquences, puisque l'on connaît la taille des objets à la compilation. La perte de mémoire occasionnée peut donc être évaluée à l'avance, en fonction des proportions des objets qui seront créés.

Enfin, la libération des structures cycliques est assurée par un ramasse-miettes secondaire (par marquage-balayage). Cette technique occasionne un surcoût en temps d'exécution qui peut néanmoins être supprimé si la structure des objets de l'application ne comporte pas de cycles.

II.A.3. Une solution basée sur le ramasse-miettes par marquage-balayage.

Une autre solution intéressante est proposée par Bacon *et al.* [11], inspirés de Siebert [12]. C'est un algorithme fonctionnant principalement par marquage-balayage et secondé par un mécanisme de recopie destiné à éliminer la fragmentation.

Premièrement, cette approche se base sur l'hypothèse que la plupart des programmes satisfont une propriété dite de «localité de taille des objets» dans le but de réduire la fragmentation externe (*i.e.* les tailles des objets qui ont été alloués fréquemment sont assez proche de celles de ceux qui seront alloués dans le futur). En effet, la mémoire est ici partitionnée en pages de taille fixée ;

chacune est ensuite divisée en blocs de taille spécifique. Cette organisation de la mémoire permet de maintenir des listes distinctes regroupant les blocs libres de même taille. La stratégie d'allocation choisie place un objet de taille donnée dans la plus petite zone qui peut le contenir (allocation en *best-fit*) afin de réduire la fragmentation interne. Les différentes tailles de blocs sont également choisies dans ce but.

En outre, ces mécanismes ne permettent pas d'éliminer totalement les problèmes de fragmentation externe. En effet, une page peut rester très peu occupée et empêcher tout de même l'allocation d'un objet plus gros. C'est pourquoi une technique de recopie est employée afin de libérer les pages les moins occupées. Cette solution impose donc l'emploi de *barrières en lecture*. L'argument avancé contre les surcoûts occasionné par ce genre de barrières est qu'il est possible d'optimiser les applications de manière à réduire leur coût [13].

De plus, l'algorithme de ramasse-miettes principalement utilisé est de type marquage-balayage *incrémental* : *i.e.* une phase complète de libération de mémoire (marquage + balayage) est répartie sur plusieurs exécutions du ramasse-miettes. Cela a pour avantage de réduire le temps des pauses occasionnées par son exécution.

Enfin, les stratégies d'ordonnancement de l'application et du ramasse-miettes sont choisies à partir de propriétés calculées sur l'application. Ces caractéristiques sont, d'une part le taux d'allocation du programme, d'autre part le taux de mort des objets de celui-ci. Mann *et al.* [14] ont tenté de proposer une technique d'évaluation de la première, mais l'inconvénient majeur de ces approches réside dans le fait que le taux de mort des objets d'une application est incalculable.

Pour conclure sur les algorithmes de ramasse-miettes temps-réel, on peut noter que Nettles et O'Toole [15] ont proposé une solution basée sur un principe de recopie des objets et d'écoute du comportement de l'application qui a pour avantage de permettre l'élimination des barrières en lecture.

II.B. Analyses statiques et gestion de la mémoire en régions

La seconde famille de solutions au problème de la gestion de la mémoire dynamique dans les systèmes temps-réel est celle des mécanismes «semi-autonomes». C'est par exemple le cas de la gestion de la mémoire en régions combinée aux analyses statiques.

Après une présentation du principe de la gestion de la mémoire en régions, seront évoquées les différentes analyses statiques actuellement utilisées dans le but de prédire le comportement d'une application sur le plan de la mémoire dynamique.

II.B.1. La gestion de la mémoire en régions.

La gestion de la mémoire dynamique en régions repose sur un principe simple : les objets qui ont le même cycle de vie sont alloués dans une même région [16]. L'espace mémoire occupé par une région peut être libéré lorsque tous les objets qu'elle contient sont morts.

Les avantages de cette technique de gestion de la mémoire sont divers. Elle permet d'une part de réduire l'espace nécessaire pour exécuter une application. Effectivement, ce dispositif permet en principe de récupérer la mémoire occupée par les objets morts relativement tôt ; *e.g.* un groupe d'objet (liste, arbre, etc.) dont tous les éléments sont placés dans la même région, peut être libéré en une seule fois lorsque la dernière référence sur cette structure disparaît. De plus, les primitives de gestion de la mémoire peuvent être implémentées de manière à s'exécuter en des temps bornés. Il existe différents types de région, à taille fixe ou variable. Une taille de région fixe implique de pouvoir borner le nombre et la taille des objets qui y seront alloués, alors qu'une taille variable apporte plus de liberté, au prix d'une gestion plus coûteuse.

Par contre, ce procédé implique de pouvoir déterminer les objets qui ont un cycle de vie similaire. On peut confier cette tâche au programmeur ou tenter de l'automatiser grâce à une analyse statique.

II.B.2. L'utilisation manuelle des régions.

Diverses approches ont été proposées dans le but de permettre une gestion manuelle des régions. Certaines se sont basées sur une modification des langages hôtes ; *e.g.* Gay et Aiken [17] ont présenté un système de typage en C permettant la gestion de la mémoire dynamique en régions. Le même genre d'adaptation existe pour le langage Java : la spécification RTSJ[1] [1] propose également une construction manuelle des groupes d'objets. L'opérateur `new` est remplacé par des appels de méthodes spécifiques : l'API[2] est agrémentée de classes spécifiant les comportements des différents types de région (région à temps d'allocation constant, à taille limitée, etc.).

Le principal problème lié à cette solution est la relative difficulté d'écriture des applications : il faut prévoir, à chaque allocation d'un nouvel objet, dans quelle région il devra être placé, ce qui n'est pas forcément plus simple que de déterminer le moment où la mémoire occupée par un objet peut être libérée. La destruction d'une région doit également être planifiée. En outre, ce changement de gestion implique la réécriture de l'ensemble de la librairie Java standard, ainsi que la modification des modèles de programmation communément utilisés [18].

En outre, l'approche de Zhao *et al.* [19], qui est une extension de la précédente, propose une modification du langage Java en ajoutant des annotations ainsi qu'une redéfinition de la notion de paquetage dans le but de rendre l'utilisation des régions plus sûre. Cette proposition s'éloigne donc trop des concepts à la base du langage Java pour être utilisable en pratique.

Enfin, Basanta-Val *et al.* [20] a également étendu la spécification RTSJ en y ajoutant des régions d'un nouveau type, permettant une libération transparente de certains objets.

II.B.3. Les analyses statiques pour la gestion de la mémoire en régions.

Puisque l'utilisation manuelle des régions s'avère difficile, des techniques ont été proposées pour automatiser leur manipulation. Il existe deux principaux types d'analyse statique dont les résultats sont utilisables afin de gérer la mémoire dynamique.

Analyse d'échappement.

Le but d'une analyse d'échappement est de déterminer statiquement (*i.e.* au moment de la compilation du programme) un sous-ensemble des objets qui ne seront plus référencés à la fin de l'exécution d'une méthode [21, 22, 23, 24]. Elle permet donc de savoir si un objet peut être placé dans la pile d'exécution (*i.e.* il meurt avant la fin de l'exécution de la méthode).

Par ailleurs, Sălcianu et Rinard [25] ont proposé une analyse statique exprimant la *pureté* des méthodes d'une application (une méthode est dite *pure* si elle n'a pas d'effet de bord au niveau du tas). On peut donc également apprendre par cette analyse si les allocations faites pendant l'exécution d'une méthode peuvent l'être dans la pile d'exécution (*i.e.* la méthode est pure).

Synthèse de régions.

Pour aller plus loin, plusieurs travaux [26, 27] proposent d'utiliser une analyse statique pour transformer le programme et lui faire utiliser un système de gestion de mémoire en régions. L'objectif de ces analyses est donc de déterminer des ensembles de *sites d'allocation* produisant des objets au cycle de vie similaire, de façon à les regrouper dans une même région.

L'avantage de ces techniques est qu'elles sont complètement automatiques. En revanche, certains types de programmes posent problème : lorsque les structure de données sont trop complexes pour l'analyse, les transformations sont imprécises et n'exploitent pas efficacement les régions. Par exemple, une région peut regrouper une structure vivant très longtemps et des objets *auxiliaires* à durée de vie très courte (itérateurs, etc), qui s'accumulent alors sans pouvoir être désalloués.

[1] *Real-Time Specification for Java* – Spécification temps-réel pour Java
[2] *Applications Programming Interface* – Interface de programmation d'applications

Pour pallier à ces inconvénients, nous proposons [4, 5] de ne pas utiliser une approche totalement automatique, mais d'impliquer le programmeur dans le travail de gestion mémoire en lui faisant remonter les résultats de l'analyse. Ainsi il peut, à l'avance, prévoir le comportement du programme pour éviter les mauvaises surprises. L'analyse proposée est assez simple, pour pouvoir être exécutée efficacement dans un environnement de développement interactif, et pour produire des résultats intelligibles pour le programmeur.

En termes de gestion mémoire, cette solution a des caractéristiques similaires aux autres travaux évoqués [26, 27] : suivant les programmes, l'utilisation automatique de régions peut avoir des effets très positifs, ou conduire à une utilisation déplorable de la mémoire. La principale différence réside dans la participation du programmeur : lorsque l'outil d'analyse rencontre un programme «problématique», c'est-à-dire dont la structure se prête mal à la transformation vers l'utilisation de régions, il soulève le problème dès la phase de compilation. Ainsi, le développement peut être guidé vers des programmes plus appropriés à cette approche.

III. Présentation des études de cas

La seconde phase du travail a consisté à analyser différentes applications problématiques. Les programmes qui ont été choisis pour cela sont, d'une part, trois programmes de la suite de benchmarks JOlden[1], d'autre part, le décodeur MP3 JLayer[2]. Les premiers font partie d'un ensemble de dix applications qui ont l'avantage d'utiliser les modèles de programmation courants (*e.g.* polymorphisme, récursivité, forte utilisation de la mémoire dynamique, etc.) : c'est pourquoi ils représentent bien le genre de programmes Java que la solution à proposer devrait pouvoir gérer. Le dernier, quant à lui, est un programme plus conséquent qui utilise la librairie Java standard de manière intensive. Il est également un bon représentant des applications dites temps-réel dans la mesure où la lecture d'un son demande une certaine fluidité de traitement : l'utilisation d'un ramasse-miettes classique (avec pauses) produirait certainement un son saccadé.

Chacun de ces programmes a été dans un premier temps analysé «à la main», c'est-à-dire en examinant son code source en tenant compte des familles créées par l'analyse statique actuelle. Ce premier examen a permis de mettre en évidence le fonctionnement global de l'application et ses parties potentiellement problématiques par rapport à la gestion en régions. Dans un second temps, les variations de son occupation mémoire lorsque celle-ci est gérée par le système actuellement proposé et avec un ramasse-miettes classique ont été comparées. Ceci a permis de confirmer et préciser les observations précédentes. Enfin, une simulation du programme a été analysée plus finement à l'aide d'un outil réalisé dans le cadre d'un stage de première année de Magistère [28]. Il permet de connaître le profil de chaque site d'allocation d'une application en terme de cycle de vie des objets alloués.

III.A. Analyse du programme Health

Le programme Health effectue une simulation du système de gestion des hôpitaux Colombien. Il crée tout d'abord un arbre d'instances de la classe Village, chaque noeud représentant une ville, possédant un hôpital (instance de la classe Hospital), lui-même contenant des listes de patients. Un certain nombre d'itérations de la simulation sont ensuite exécutées, créant des patients, les déplaçant d'hôpital en hôpital, etc.

En ce qui concerne les sites d'allocation indiqués problématiques par l'analyse inférant les familles d'objets, ils peuvent se diviser en deux catégories : ceux qui sont dans des méthodes récursives appelées pendant la création des structures initiales (instanciation de Village et Hospital) et ceux qui allouent des objets pendant la boucle de simulation (création et ajout de patients dans les listes).

Après l'analyse manuelle de l'application, une description de son comportement effectif par rapport au système de gestion de la mémoire en régions va être présentée. Elle sera suivie par une étude des statistiques sur la démographie des objets alloués.

Comportement avec la gestion en régions actuelle

Le graphe de la figure III.1 page suivante présente deux courbes issues de deux simulations du programme Health sur les mêmes données. On peut dans un premier temps remarquer que le système de gestion de la mémoire dynamique actuel n'est pas efficace : une ou plusieurs régions se

[1] http://www-ali.cs.umass.edu/DaCapo/benchmarks.html

[2] http://www.javazoom.net/javalayer/javalayer.html

peuplent de plus en plus et l'espace occupé par les objets qu'elles contiennent n'est jamais récupéré. En outre, la courbe de l'exécution du programme avec utilisation d'un ramasse-miettes indique que de moins en moins d'objets meurent (les «dents» qui apparaissent sur cette courbe sont dues à la configuration du ramasse-miettes : il est déclenché chaque fois que la taille de la mémoire occupée dépasse 400000 octets).

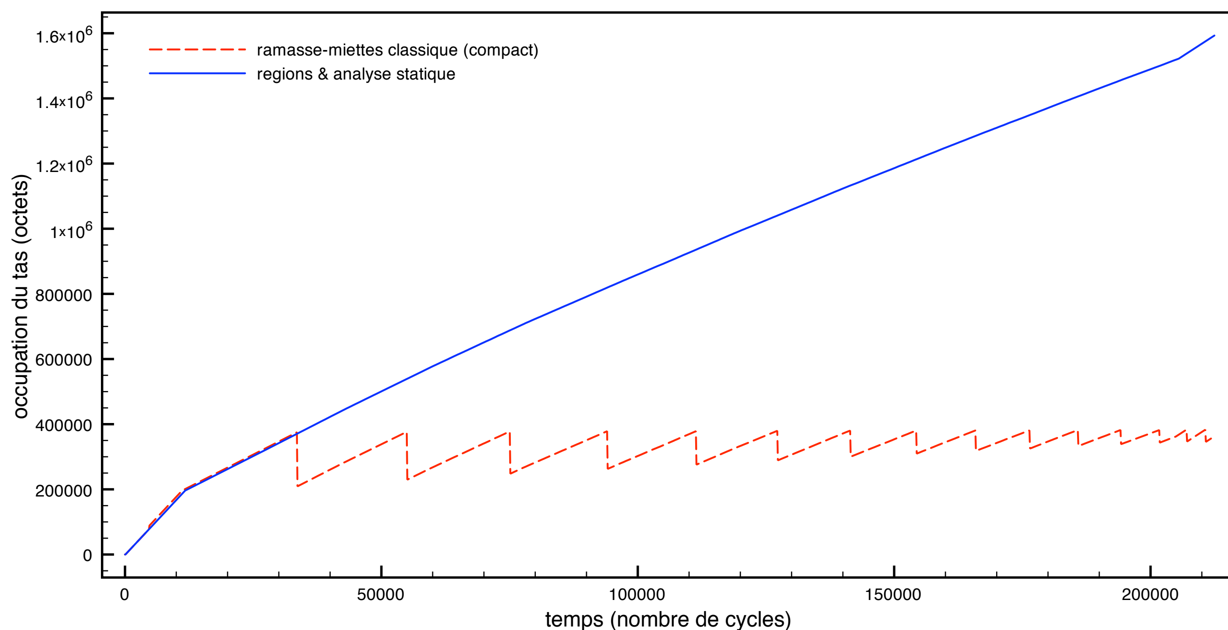

Fig. III.1: *Graphe résumant la variation de l'occupation de la mémoire dynamique par le programme* Health *avec le système de gestion en régions actuel (courbe pleine) et avec l'utilisation d'un ramasse-miettes de type marquage-balayage (courbe en pointillés).*

En ce qui concerne les résultats de l'analyse inférant les familles d'objets, on peut remarquer que quelques sites sont indiqués comme potentiellement problématiques : notamment des sites allouant des itérateurs, des instances de Patient ou des listes. Les méthodes d'affichage ou celle de traitement des arguments de la ligne de commande (Health.parseCmdLine) sont déclarées sûres dans la mesure où elles ne sont pas impliquées dans les régions à problème.

Comportement détaillé

L'analyse détaillée d'une simulation de ce programme dans la plate-forme JITS (*cf* figure III.2 page ci-contre) révèle plusieurs informations intéressantes.

Premièrement, on peut constater qu'il y a effectivement deux phases «distinctes» pendant l'exécution du programme : la première pendant la création et l'initialisation des structures (*i.e.* pendant les 5000 premiers cycles) ; la seconde pendant la simulation (*i.e.* le reste de l'exécution).

En outre, hormis pour la méthode Health.parseCmdLine, tous les objets créés pendant la première étape de l'exécution vivent jusqu'à la fin de celle-ci. Ce comportement est aisément explicable par la logique du programme : ces structures initiales sont permanentes et non modifiées. En ce qui concerne la méthode de traitement des arguments de la ligne de commande, ses sites peuvent être considérés comme gérés correctement puisque l'analyse inférant les régions les place tous dans une petite famille.

De plus, le site qui alloue le plus d'objets construit des instances de la classe java.util.Enumeration. Elles servent donc à parcourir les listes de l'application et ont des durées de vie très courtes. Deux sites (Hospital.checkPatientsAssess(LVillage ;)LList ; :0 et Village.simulate()LList ; :1) créent également des objets qui meurent tôt, alors que la méthode List.add alloue des objets durables.

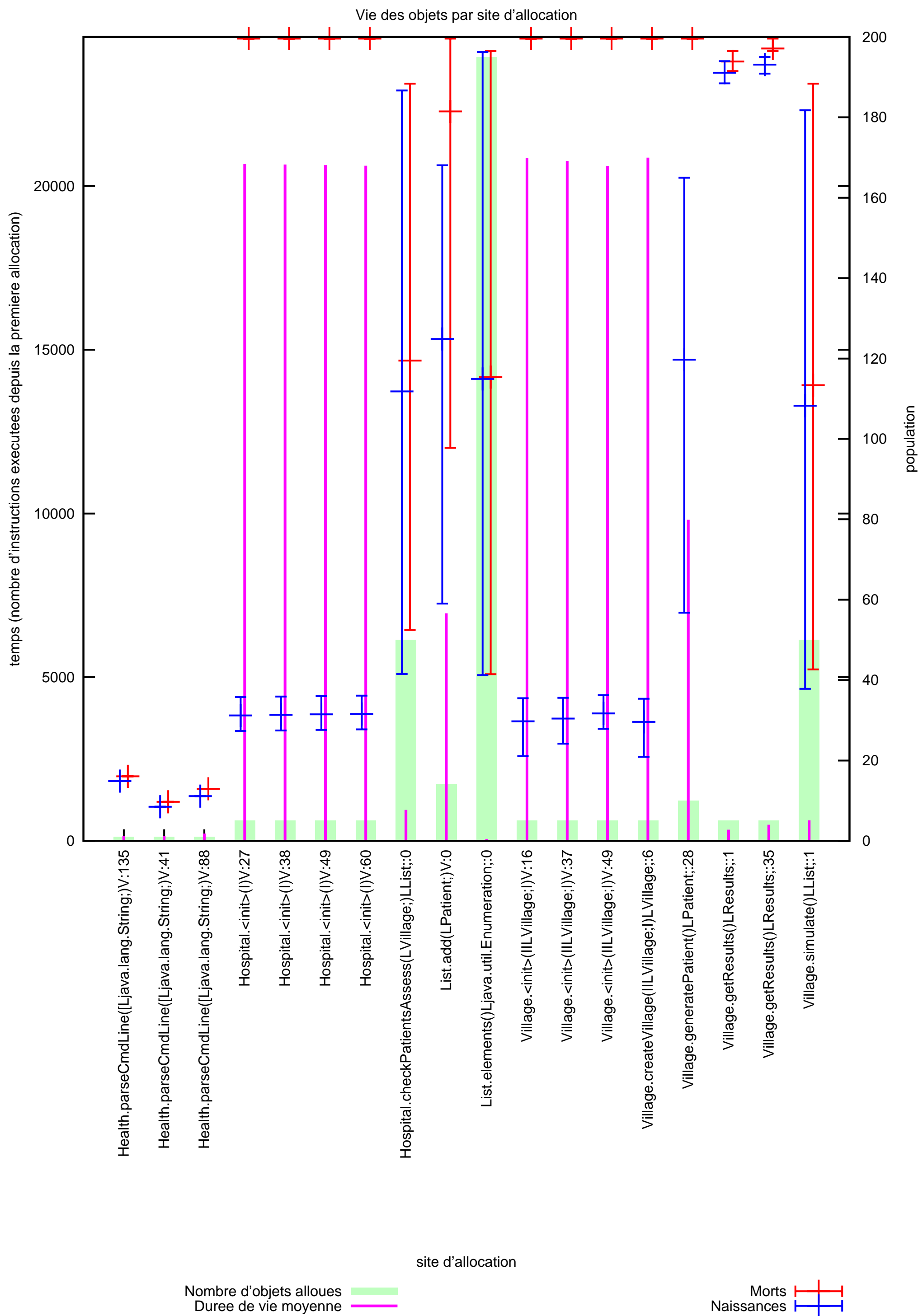

Fig. *III.2: Graphe représentant, pour chaque site d'allocation de l'application* Health*, l'intervalle de temps de naissance des objets alloués, l'intervalle de temps de mort de ceux-ci, leur durée de vie moyenne ainsi que leur effectif total.*

Enfin, la méthode Village.generatePatient alloue des objets qui vivront jusqu'à la fin de l'exécution pendant la simulation.

Cette analyse permet de mettre en avant un certain nombre de caractéristiques du programme Health qui peuvent être exploitée afin d'améliorer la gestion de sa mémoire.

Dans un premier temps, il apparaît intéressant de séparer l'exécution en deux phases. En effet, la première partie comporte des sites «à risque», mais ceux-ci servent à l'initialisation des structures sur lesquelles le programme travaillera. Comme tous les objets alloués dans la méthode Health.parseCmdLine sont placés dans la même région par l'analyse statique, l'ensemble des autres objets créés pendant cette étape pourraient l'être dans une région globale vivant jusqu'à la fin de l'exécution. Il serait également utile d'y placer les objets créés par la méthode Village.generatePatient.

De plus, on a vu précédemment que les objets auxiliaires pouvaient le plus souvent être traités par une analyse d'échappement : le site allouant des instances de la classe java.util.Enumeration peut donc certainement allouer ses objets dans la pile d'exécution.

Enfin, les sites restants sont placés dans la même famille par l'analyse statique. On peut alors constater que des objets à durée de vie courte côtoieront des objets vivant plus longtemps. Les régions associées à cette famille (une seule pour cette application) pourraient certainement bénéficier d'un ramasse-miettes interne.

III.B. Analyse du programme Voronoï

Cette application construit une triangulation de Delaunay sur un graphe de sommets (classe Vertex) généré aléatoirement. Cette construction implique la création, la modification et la suppression d'arêtes (instances de Edge). La construction de la triangulation se fait récursivement par la méthode Vertex.buildDelaunay, appelant elle-même Edge.makeEdge et Edge.doMerge.

De même que l'application Health pouvait se diviser en deux parties distinctes, celle-ci, après une phase de création du graphe de sommets, appelle la méthode de triangulation.

Comportement avec la gestion en régions actuelle

Le graphe de la figure III.3 présente deux courbes issues de deux simulations du programme Voronoï sur les mêmes données. On peut remarquer que la quantité de mémoire nécessaire à l'application croît avec le temps. En effet, la courbe représentant l'occupation mémoire lorsqu'un ramasse-miettes est utilisé révèle que la machine virtuelle augmente toujours la taille du tas : ceci est dû à l'algorithme utilisé, qui créée au moins une nouvelle arête à chaque appel de la méthode Vertex.buildDelaunay. Les arêtes supprimées ou remplacées peuplent toujours la mémoire lorsque celle-ci est gérée à l'aide des régions.

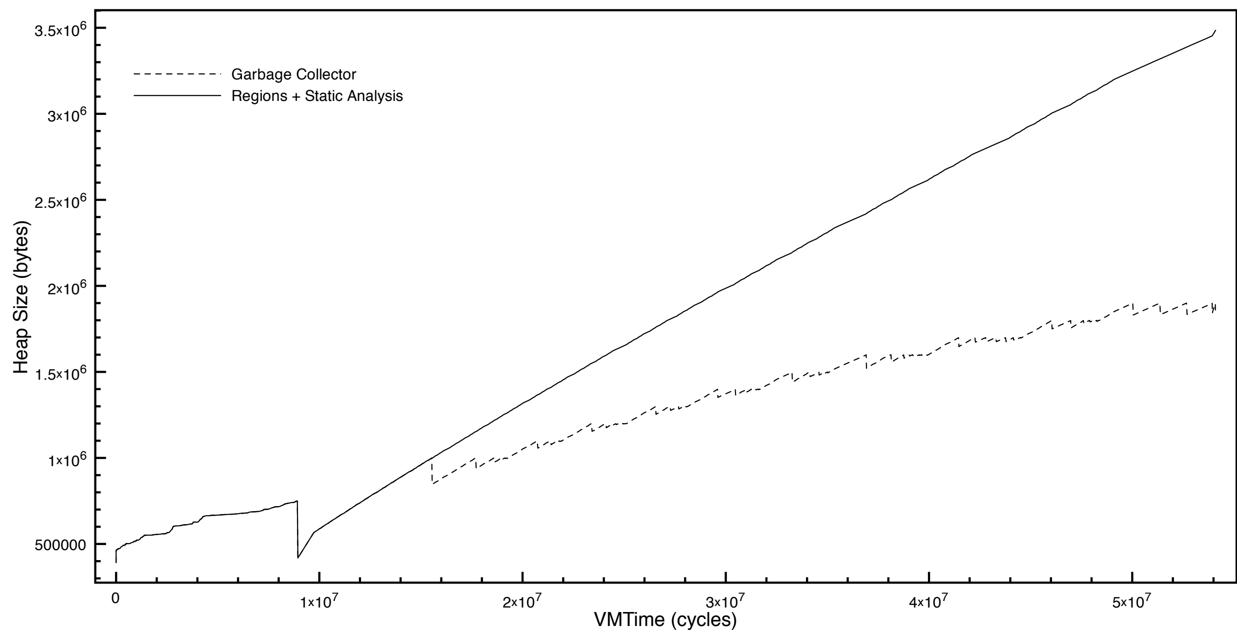

Fig. *III.3: Graphe résumant la variation de l'occupation de la mémoire dynamique par le programme Voronoï avec le système de gestion en régions actuel (courbe pleine) et avec l'utilisation d'un ramasse-miettes de type marquage-balayage (courbe en pointillés).*

Enfin, l'analyse statique regroupe l'ensemble des points et des arêtes dans une même famille, qui est de plus la seule à présenter des sites d'allocation potentiellement problématiques.

Comportement détaillé

La figure III.4 page suivante présente la démographie des objets pendant une exécution du programme Voronoï.

Comme pour le programme Health, quelques sites d'allocation des méthodes Voronoi.main et Voronoi.parseCmdLine n'allouent chacun qu'un unique objet qui meurt rapidement. À l'inverse, tous les points sont créés pendant la première partie de l'exécution et vivent jusqu'à la fin de celle-ci (méthode Vertex.createPoints).

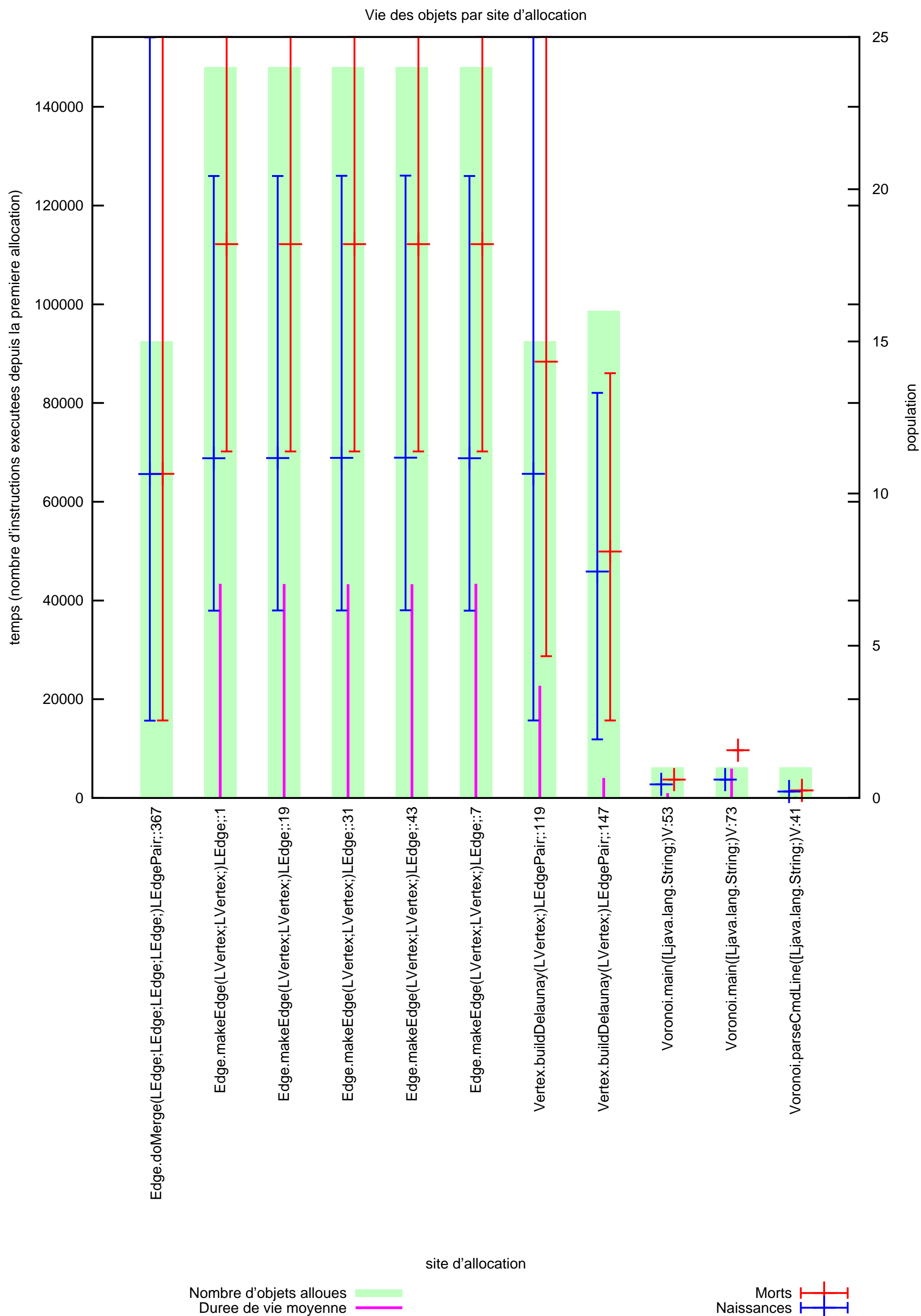

FIG. *III.4: Graphe représentant, pour chaque site d'allocation de l'application* Voronoï, *l'intervalle de temps de naissance des objets alloués, l'intervalle de temps de mort de ceux-ci, leur durée de vie moyenne ainsi que leur effectif total.*

En outre, tous les sites actifs pendant la seconde phase allouent constamment des objets à durée de vie relativement longue, excepté celui de la méthode Edge.doMerge. Aussi, on peut remarquer

que tous ces sites ont des objets qui viendront peupler la région qui contient les instances de la classe Vertex (il n'y en a qu'une seule concernée pour cette application), même après leur mort.

De même que pour l'application Health, l'ensemble des structures initiales (les sommets dans ce cas) peuvent être alloués dans une région vivant jusqu'à la fin de l'exécution.

En outre, l'examen manuel du programme et la faible durée de vie des objets alloués par le site d'allocation de la méthode Edge.doMerge indique que ceux-ci pourraient certainement être placés dans la pile d'exécution.

Enfin, le reste des sites problématiques étant directement impliqué dans la construction des objets composant le graphe, une stratégie différente devrait être adoptée. Par exemple, l'utilisation d'une région encapsulant un ramasse-miettes pourrait être efficace dans ce cas.

III.C. Analyse du programme BH

L'application BH construit initialement un ensemble de corps célestes (instances de Body), et simule l'effet de la gravité entre eux. Chaque itération de la simulation crée un arbre dont chaque nœud est un regroupement de corps permettant d'approximer le calcul des forces en jeu entre ces corps.

Les sites d'allocation indiqués problématiques par l'analyse concernent majoritairement la création de l'arbre de calcul. Effectivement, cet arbre est recréé à chaque pas de simulation et est placé dans la même région que les corps célestes. Il se comporte donc comme un objet *auxiliaire* alloué dans une boucle et pointant sur une structure à durée de vie longue.

Comportement avec la gestion en régions actuelle

La comparaison du comportement de ce programme au niveau de l'occupation du tas lorsque celui-ci est géré avec le système des régions ou avec un ramasse-miettes classique apporte plusieurs informations (*cf* figure III.5). D'une part, la quantité de données utiles reste relativement constante : en effet, chaque exécution du ramasse-miettes classique récupère approximativement la même quantité de mémoire. De plus, le système de gestion de la mémoire en régions actuel ne gère pas correctement une partie des sites d'allocation : comme prévu par l'analyse, une ou plusieurs régions se peuplent d'objets à durées de vie très différentes.

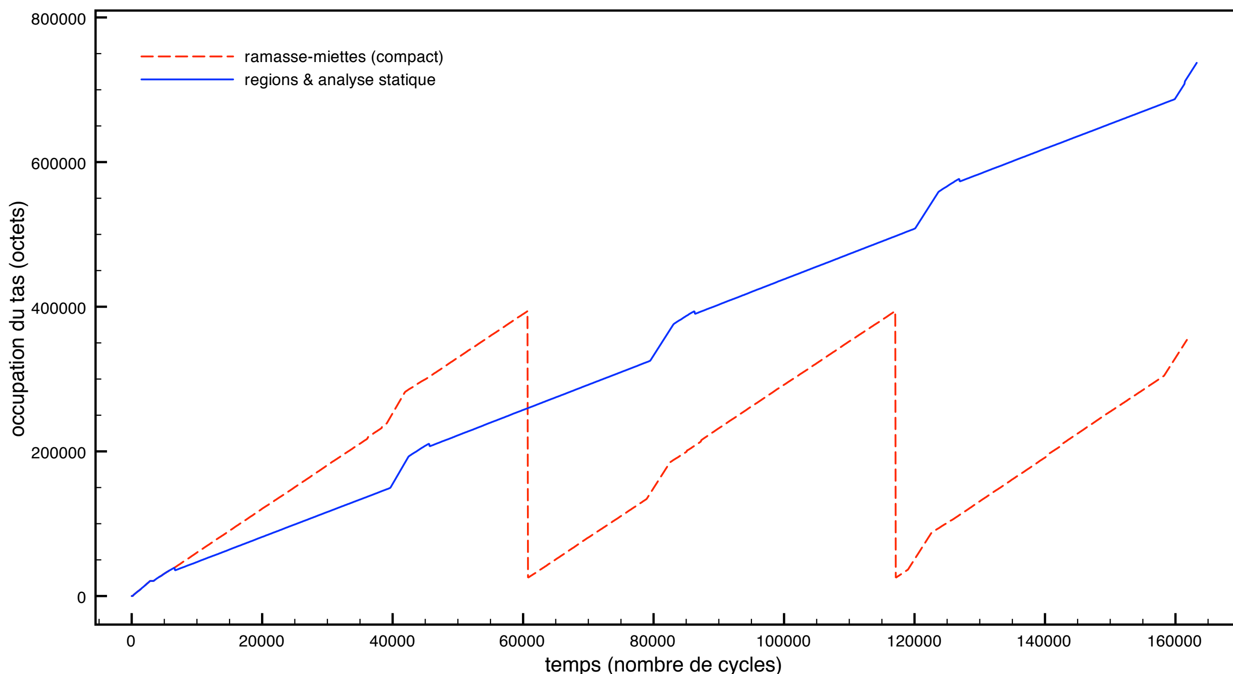

Fig. *III.5: Variation de l'occupation de la mémoire dynamique par le programme* BH *avec le système de gestion en régions actuel et avec l'utilisation d'un ramasse-miettes de type marquage-balayage.*

Comportement détaillé

La figure III.6 page 22 présente la démographie des objets pendant une exécution du programme BH.

Premièrement, on peut remarquer sur ce graphique que les instances de la classe Body sont effectivement créées au début de l'exécution et vivent jusqu'à la fin de celle-ci (site du constructeur

de la classe Tree et Tree.createTestData(I)V :57). Cependant, elles référencent aussi des objets créés pendant cette phase, mais qui meurent tôt, pendant une itération de la simulation.

De plus, beaucoup d'objets éphémères sont alloués pendant la phase de simulation. Le constructeur de MathVector ou la méthode MathVector.clone par exemple, allouent beaucoup d'objets pendant toute l'exécution. On remarque également l'allocation d'objets auxiliaires pendant la phase de simulation (méthode Body.elementsRev).

Tout comme les deux programmes précédents, celui-ci crée une structure initiale complexe vivante jusqu'à la fin de l'exécution.

En outre, certains sites problématiques sont très sollicités et contribuent à rendre le mécanisme de gestion de la mémoire en régions plutôt inefficace sur ce programme.

III.D. Analyse du décodeur MP3 JLayer

L'analyse manuelle et l'examen des caractéristiques des sites d'allocation du programme JLayer ont permis de confirmer que les sites d'allocation indiqués comme problématiques par l'analyse statique actuelle présentaient des similitudes avec ceux des programmes précédemment étudiés. De plus, cette application se divise en deux phases d'exécution, la première initialisant les structures du programme (ouverture du fichier MP3, lecture de son entête, création de grands tableaux utilisés pour accélérer la décompression et les calculs, etc.) : elle alloue beaucoup d'objets à durée de vie longue. La seconde partie est la lecture proprement dite : une boucle allouant tous ses objets dans une région recréée à chaque itération : celle-ci n'est pas indiquée problématique par l'analyse statique.

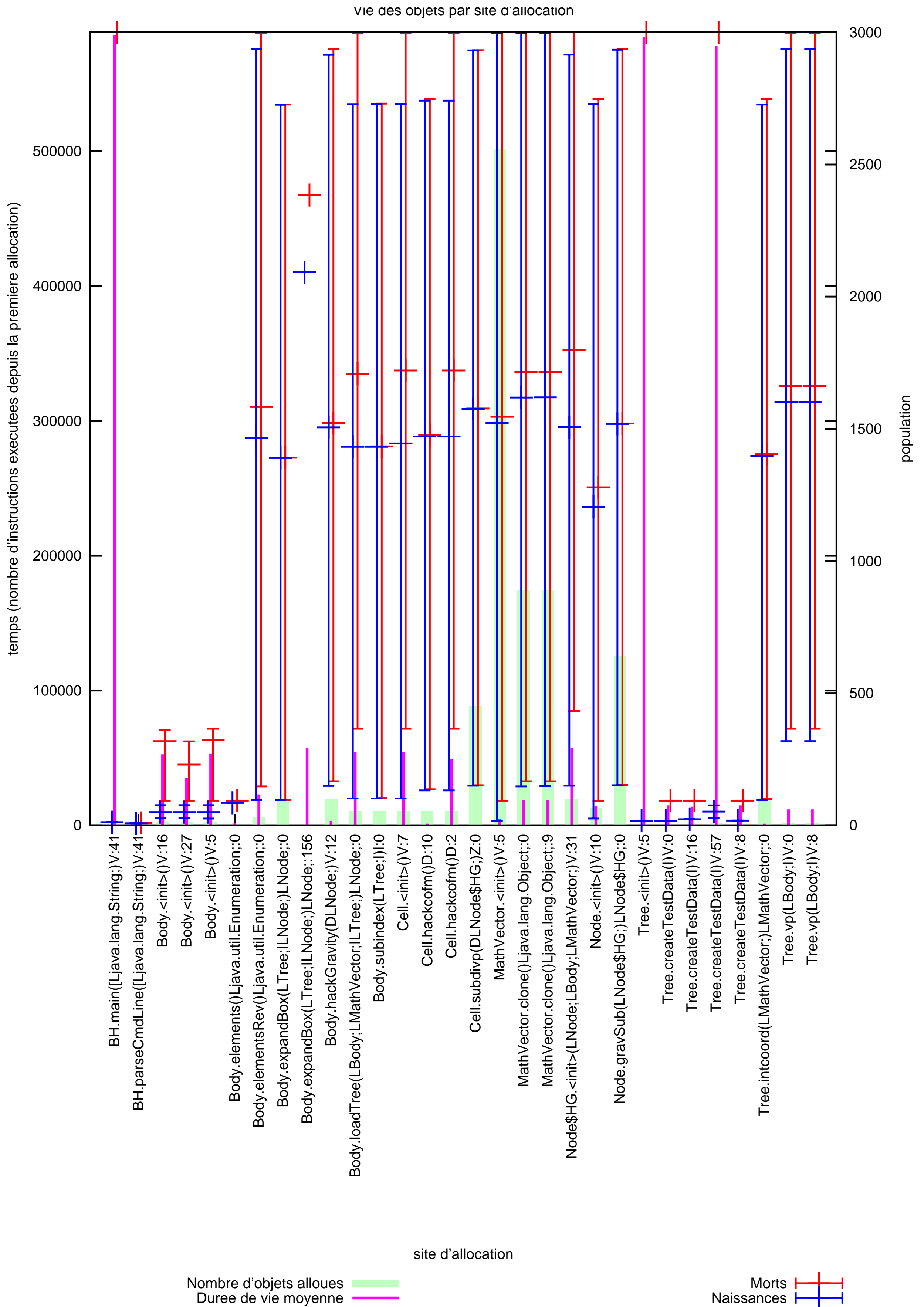

FIG. III.6: *Représentation, pour chaque site d'allocation de l'application* BH*, l'intervalle de temps de naissance des objets alloués, l'intervalle de temps de mort de ceux-ci, leur durée de vie moyenne ainsi que leur effectif total.*

IV. Contributions

Dans le but de proposer une solution au problème posé, nous avons envisagé deux pistes : la première vise à réduire l'ensemble des objets à gérer sous contrainte temps-réel et la seconde a pour objectif d'améliorer l'efficacité du mécanisme de gestion en régions actuel. Nous les avons évaluées après les avoir implémentées dans la plate-forme JITS.

IV.A. Contributions théoriques

IV.A.1. Séparation des phases d'exécution

Le premier point qui ressort des études précédentes est que les applications visées présentent un profil similaire. En effet, elles commencent toutes par initialiser un ensemble de structures qui vivent souvent jusqu'à la fin de l'exécution. On peut donc en déduire que tous les objets alloués durant cette première période peuvent être gérés par un système ne respectant pas nécessairement les contraintes des systèmes temps-réel.

Aussi, il est envisageable d'exécuter cette phase, éventuellement hors-ligne (*i.e.* avant le déploiement de l'application), en gérant les allocations à l'aide d'un ramasse-miettes classique. Il suffit ensuite de forcer l'élimination de tous les objets morts, puis de continuer l'exécution avec le système de gestion en régions. Ainsi, les sites d'allocation indiqués problématiques par l'analyse statique peuvent être correctement traités lorsqu'ils allouent des objets dans ce contexte.

IV.A.2. Gestion de la mémoire en régions et ramasse-miettes par comptage de références

Après avoir traité une portion des allocations problématiques, restait à gérer le syndrome des régions qui se peuplent d'objets morts pendant l'exécution de la phase principale de l'application. Pour cela, nous proposons d'employer un modèle hybride de gestion de la mémoire dynamique : le système des régions, secondé d'un ramasse-miettes respectant les contraintes des systèmes temps réels. Ce dernier peut être un mécanisme de comptage de références qui a l'avantage d'être facilement combinable avec le modèle des régions. En outre, il se plie bien aux contraintes des systèmes temps-réel (*cf* section II.A.2 page 9).

De plus, l'implication du développeur du programme dans le choix du système de gestion de la mémoire [5] permet «a priori» d'éliminer le problème des cycles. En effet, si l'application se révèle être incompatible avec le système des régions secondé du ramasse-miettes par comptage de références parce qu'elle manipule des structures de données trop complexes, alors on peut considérer ce programme comme incompatible avec le modèle proposé.

Cette solution consiste donc à remplacer les allocations dans des régions problématiques par une allocation dans un espace géré par un ramasse-miettes de type comptage de références. Le surcoût occasionné par ce mécanisme est alors réduit si une grande proportion d'objets est allouée dans des régions. De plus, cette solution permet aussi de traiter les éventuels objets alloués pendant la phase d'initialisation qui meurent durant l'exécution proprement dite de l'application, s'ils sont alloués dans cet espace.

IV.B. Contributions pratiques

Afin d'expérimenter les propositions précédentes, j'ai effectué plusieurs implémentations dans la plate-forme JITS.

IV.B.1. Outil d'étude de la démographie des objets d'une application

Le premier outil utilisé avait été réalisé lors de mon stage de première année de Magistère [28]. Il a pour objectif d'analyser dynamiquement la démographie des objets d'une application Java en scrutant toutes les naissances et morts des objets qu'elle alloue. Un graphique résumant, pour chaque site d'allocation, les cycles de vie des objets qu'il a créés peut ensuite être déduit. Il représente pour chacun le nombre d'allocations et la durée de vie moyenne des objets alloués. Y figurent aussi l'intervalle de temps entre la première et la dernière allocation ainsi que celle entre la première et la dernière mort des objets que chaque site a créés.

Il est alors possible, notamment lorsqu'un site est indiqué potentiellement problématique par l'analyse statique, de savoir pendant quelle phase de l'exécution il alloue des objets, à quelle fréquence, et si ceux-ci ont globalement une durée de vie très différente de celle des autres sites de la région associée.

IV.B.2. Implémentation d'un ramasse-miettes par comptage de références

La solution à mettre en œuvre étant basée sur l'utilisation d'un ramasse-miettes par comptage de références, une partie conséquente du travail a consisté en son implémentation dans la machine virtuelle native de JITS.

Le principe de ce type de ramasse-miettes est basé sur un mécanisme de barrières en écriture et de gestion de compteurs (*cf* section II.A.1 page 7). Plutôt que d'utiliser un espace restreint dans l'entête de chaque objet, j'ai choisi de considérer ce compteur comme un champ ajouté aux objets dont le nombre de références doit être compté. La raison principale de ce choix est que l'objectif de cette implémentation n'est pas la recherche de performances particulières, mais plus l'évaluation de la solution hybride de gestion de la mémoire en terme d'occupation du tas. De plus, les transformations à apporter pour placer ce compteur dans les entêtes sont dépendantes des programmes : le choix de sa précision dépend du nombre maximal qu'il peut atteindre.

En outre, les barrières en écriture à considérer dans le cas de la machine virtuelle Java sont les *bytecodes* comme putfield ou astore, ou encore le retour et l'appel d'une méthode. Les modifications sur les compteurs des objets concernés sont effectuées en conséquence par les traitants de ces barrières.

Enfin, le ramasse-miettes effectivement ajouté à la machine virtuelle libère immédiatement les objets dont le compteur devient nul. Cet aspect n'est pas conforme à la spécification d'un mécanisme compatible avec les contraintes temps-réel (cas de la libération d'une structure de données volumineuse). Cependant, l'implémentation actuelle peut aisément être modifiée de manière à ajouter les objets à libérer dans une liste, parcourue par portions de taille fixe ou encore lorsqu'une allocation échoue. Cette technique permet d'éviter les parcours de coûts imprédictibles des grandes structures lors de la libération.

L'objectif final étant de combiner ce mécanisme avec le système de gestion de la mémoire en régions déjà implémenté, les contraintes importantes étaient d'assurer la compatibilité des deux systèmes.

Enfin, pour aider à l'implémentation et à la vérification de ce ramasse-miettes, j'ai choisi d'ajouter deux outils connexes à la machine virtuelle.

IV.B.3. Outil de photographie du tas

J'ai implémenté ce premier outil dans la machine virtuelle native de JITS afin de générer une image du tas à un instant donné de l'exécution des applications. Un graphe orienté représentant l'ensemble des objets du tas avec l'état de leur compteur de références est produit, permettant ainsi d'évaluer la forme globale des structures manipulées à tout moment de l'exécution du programme.

La figure IV.1 montre un exemple de graphe résultant pour le programme BH. On y remarque la liste doublement chaînée de corps célestes, ainsi que les données propres à chacun.

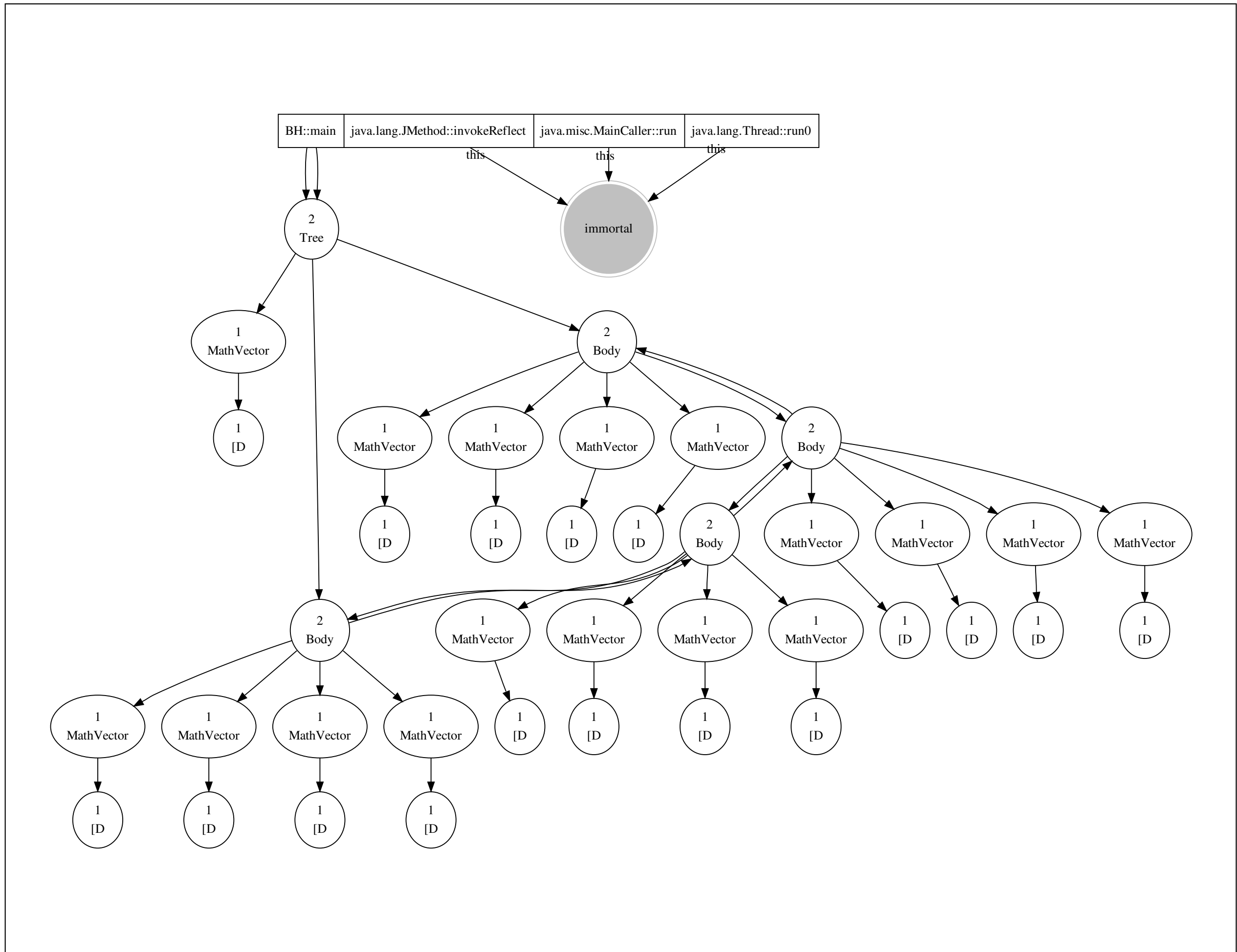

Fig. *IV.1: Graphe représentant le tas manipulé par l'application BH, exécutée avec quatre corps célestes. La pile d'exécution y est également représentée. Les objets immortels (instances de java.lang.Class, etc.) sont regroupés dans le nœud "immortal".*

IV.B.4. Mesures d'occupation et de fragmentation du tas

Afin d'évaluer l'efficacité des mécanismes de gestion de la mémoire implémentés, j'ai ensuite instrumenté de la machine virtuelle. Les variations d'occupation réelle du tas, ainsi que le surcoût lié à l'utilisation de pages par le mécanisme des régions [5], peuvent ainsi être extraites pendant l'exécution des programmes.

La figure IV.2 page suivante illustre un exemple des résultats obtenus. On peut remarquer sur ces courbes que la fragmentation interne au sein des pages reste globalement constante.

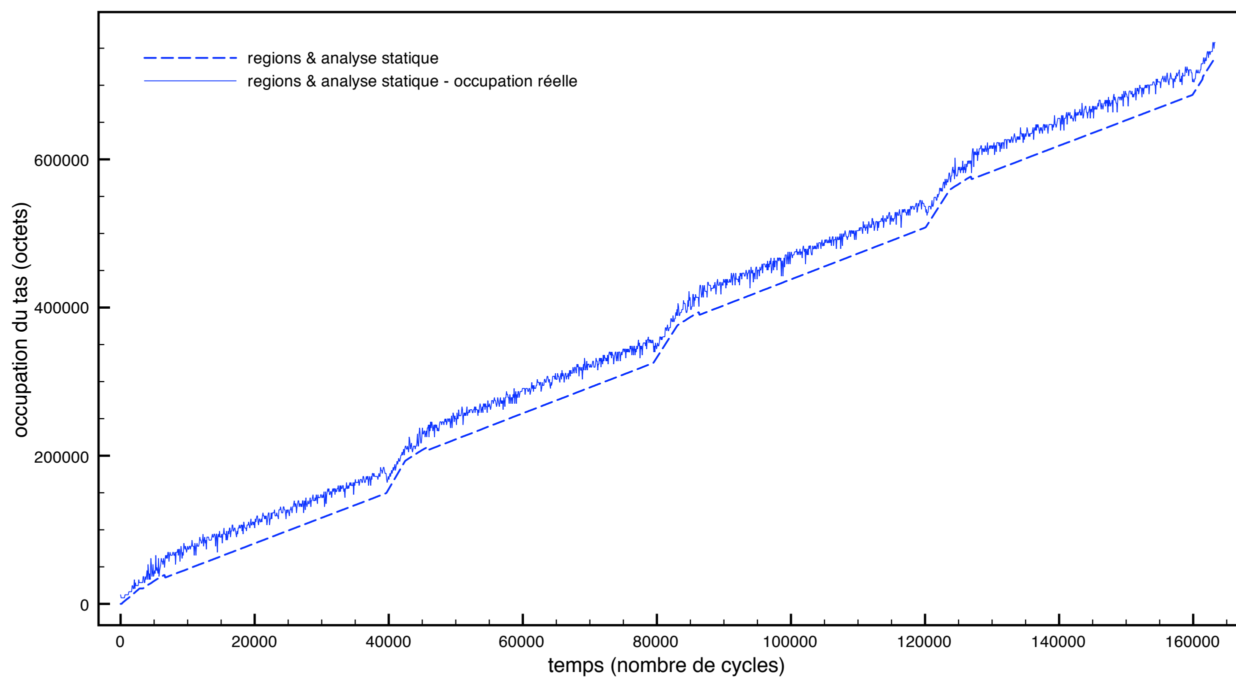

Fig. *IV.2: Représentation de la variation de l'occupation réelle (pages occupées) et de l'occupation utile (taille occupée par les objets) du tas par le programme* BH*, lorsque sa mémoire est gérée avec le système des régions uniquement.*

V. Validation expérimentale

Parmi les études de cas qui ont été présentées figuraient quatre programmes intéressants : ils comportaient des sites d'allocations indiqués problématiques par l'analyse statique. Parmi ceux-ci, nous avons traité complètement deux programmes avec les solutions proposées.

V.A. Résultats pour le décodeur MP3 JLayer

La première proposition nous suffit à traiter ce programme, en raison de son architecture et de son comportement au niveau de la mémoire dynamique. En effet, ce décodeur alloue la majorité des objets qu'il utilise pendant sa phase d'initialisation, et le reste est placé dans des régions à durée de vie courte.

Les courbes de la figure V.1 illustrent le comportement en terme d'occupation mémoire de ce programme avec un ramasse-miettes de type marquage-balayage et avec le système actuellement proposé. Il en ressort que, par rapport à l'utilisation d'un ramasse-miettes classique qui provoquerait des pauses pendant la lecture d'un son (marches descendantes dues aux exécutions périodiques du ramasse-miettes), le modèle en régions rend le niveau d'utilisation du tas globalement constant.

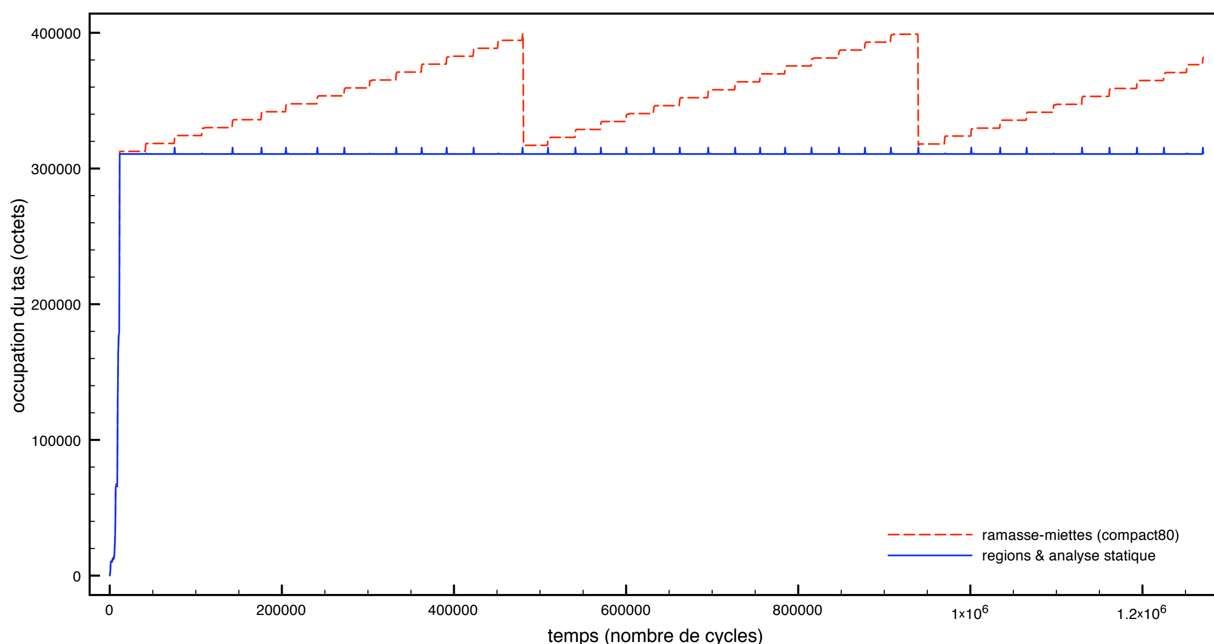

Fig. *V.1: Variations de l'occupation du tas par le décodeur MP3 JLayer avec un ramasse-miettes classique et le système de gestion de la mémoire en régions.*

On peut en déduire que, pour ce programme, les avertissements de l'analyse statique peuvent être ignorés par le développeur, puisqu'ils concernent tous la phase d'initialisation des structures.

V.B. À propos de Health et Voronoï

Parmi les programmes étudiés, Health et Voronoï présentaient deux profils assez similaires. Ils travaillent sur un graphe irrégulier, une structure relativement complexe à gérer en terme de libération d'objets. La figure V.2 page ci-contre est la représentation du tas de Voronoï à la fin de son exécution sur trois sommets obtenue avec l'outil présenté section IV.B.3 page 25. On remarque bien la complexité du graphe, comprenant beaucoup de cycles, ainsi que le nombre important de connexions entre les objets. De même, la figure V.3 page 30 illustre la complexité du tas de l'application Health.

L'examen des structures des tas de ces deux programmes montre que l'organisation des données manipulées est trop complexe pour qu'elles puissent être correctement gérées par le système actuel. En outre, l'élimination des objets auxiliaires ne suffit pas à solutionner le problème du peuplement des régions par des objets morts. De plus, l'utilisation d'un ramasse-miettes par comptage de références ne suffit pas en raison de la présence des cycles dans la structure.

On peut donc conclure à propos de ces deux programmes que la gestion de la mémoire dynamique dans leur cas est trop complexe pour être faite correctement en respectant les contraintes temps-réel. En outre, il n'est pas non plus envisageable de gérer la mémoire de ces programmes manuellement. On peut même dire qu'ils nécessitent un ramasse-miettes traitant les cycles, donc de type marquage-balayage.

V.C. Résultats pour le programme BH

Le dernier cas intéressant, le programme BH, présente un profil plus adapté à la gestion en régions. En effet, le graphe d'objets manipulé présente une structure moins complexe que les précédents. D'après l'analyse de la structure manipulée par l'application, les seuls cycles qui apparaissent sont dans la liste doublement chaînée de corps célestes (*cf* section III.C page 20), liste dont la durée de vie est celle de l'application. Hélas, cette information ne peut être vérifiée par une analyse de type et ne peut pour le moment qu'être confirmée par le développeur du programme.

Le graphique de la figure V.4 page 30 représente les variations de l'occupation du tas par ce programme avec trois systèmes de gestion de la mémoire dynamique. On remarque dans un premier temps que la nouvelle solution permet de rendre ce niveau globalement constant. La région principale ne croît plus comme avec le système de gestion de la mémoire en régions seul : la mémoire occupée par les objets composants l'arbre de calcul est recyclée par le ramasse-miettes.

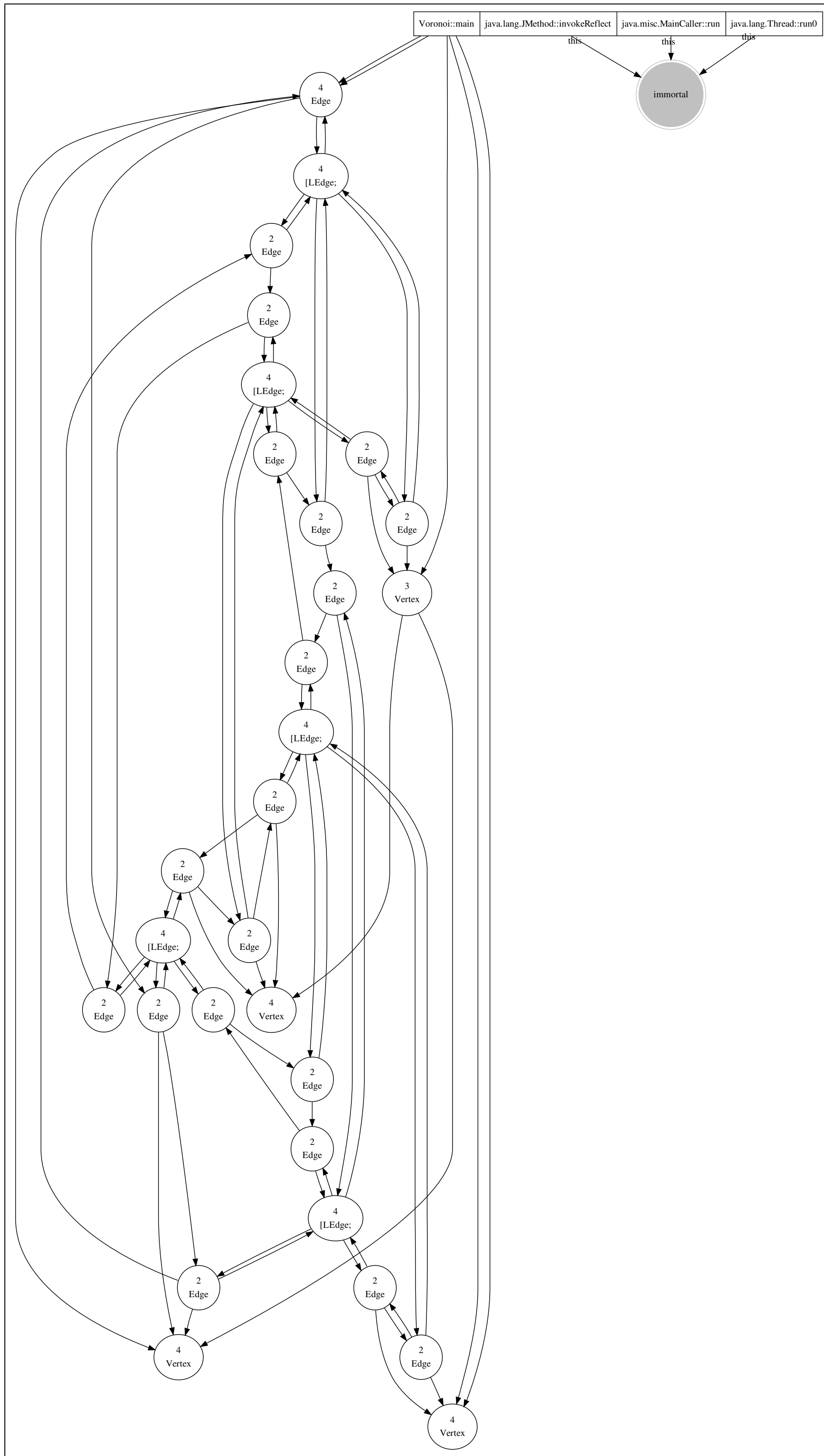

FIG. *V.2: Représentation du tas de l'application* Voronoï *à la fin de l'exécution sur trois sommets.*

FIG. V.3: *Représentation d'une portion de tas de l'application* Health *à la fin de l'exécution.*

FIG. V.4: *Variations de l'occupation de la mémoire dynamique par le programme* BH *avec le système de gestion en régions actuel, avec l'utilisation d'un ramasse-miettes de type marquage-balayage et avec le gestionnaire hybride.*

VI. Conclusions et perspectives

L'étude des différentes solutions existantes au problème de la gestion de la mémoire dynamique a permis de mettre en avant quelques problèmes et les diverses solutions qui ont été proposées pour les résoudre. J'ai ensuite mené un ensemble d'études de cas, révélant des propriétés caractéristiques des programmes à traiter. À partir de ces observations, je propose deux techniques afin d'améliorer la solution proposée par l'équipe.

Ainsi, les objets créés pendant l'initialisation des programmes peuvent être gérés par un ramasse-miettes ne respectant pas les contraintes des systèmes temps-réel. Cette solution élimine une partie des problèmes rencontrés par le mécanisme de gestion de la mémoire en régions, puisque l'exécution de la plupart des programmes à traiter peut se diviser en phases de comportements généralement différents. De plus, l'utilisation d'un ramasse-miettes par comptage de références au sein des régions indiquées problématiques par l'analyse statique permet de traiter le cas du regroupement d'objets à durées de vie très différentes lorsque les structures de données manipulées restent simples (*i.e.* ne contiennent pas de cycles).

J'ai ensuite implémenté ce modèle de gestion de la mémoire dans la machine virtuelle native de JITS, accompagné d'outils annexes de photographie du tas, ainsi que d'évaluation de sa fragmentation. J'ai ensuite mené des expérimentations sur les applications problématiques précédemment présentées. D'une part, cet ensemble d'outils a permis de constater que deux de ces programmes manipulent des données trop complexes pour être gérées par un ramasse-miettes autre que de type marquage-balayage ou recopie (présence de cycles, etc.). Par ailleurs, j'ai vérifié que le lecteur MP3 est correctement géré avec le système de gestion de la mémoire en régions seul. Le programme BH, quant à lui, s'exécute en espace mémoire relativement constant avec le nouveau modèle.

Le résultat obtenu concernant les deux programmes jugés incompatibles avec la gestion en régions peut être nuancé en remarquant que, même manuellement et en connaissant le comportement mémoire d'une application manipulant des graphes irréguliers, la gestion de sa mémoire dynamique est très difficile. Il n'est donc pas envisageable de traiter ces programmes avec un modèle trop généraliste.

Pour poursuivre, une perspective de ce travail serait de prendre le problème à l'envers et d'utiliser un ramasse-miettes par comptage de références globalement, secondé de la gestion de la mémoire en régions pour traiter les cycles. L'étude d'une analyse statique permettant de savoir si une application manipule des données potentiellement cycliques, et replacer ceux-ci dans le contexte des régions peut également être un problème intéressant.

Bibliographie

[1] Gregory Bollella and James Gosling. *The real-time specification for Java*. Java series. Addison-Wesley, Reading, MA, USA, 2000. (cité pages 4 et 11)

[2] Zhiqun Chen. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. (cité page 4)

[3] Gabriel Bizzotto. JITS : Java In The Small. Master's thesis, Université de Lille 1, 2002. (cité page 4)

[4] Guillaume Salagnac, Chaker Nakhli, Christophe Rippert, and Sergio Yovine. Efficient region-based memory management for resource-limited real-time embedded systems. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006), co-located with ECOOP'06*, 2006. (cité pages 4 et 12)

[5] Guillaume Salagnac, Christophe Rippert, and Sergio Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *13th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'07)*, 2007. (cité pages 4, 12, 23 et 25)

[6] Richard Jones and Rafael Lins. *Garbage collection : algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996. (cité page 7)

[7] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance evaluation of write barrier implementation. In *OOPSLA '92 : conference proceedings on Object-oriented programming systems, languages, and applications*, pages 92–109, New York, NY, USA, 1992. ACM Press. (cité page 7)

[8] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *IWMM '92 : Proceedings of the International Workshop on Memory Management*, pages 388–403, London, UK, 1992. Springer-Verlag. (cité page 9)

[9] Tobias Ritzau and Peter Fritzson. Decreasing memory overhead in hard real-time garbage collection. In *EMSOFT '02 : Proceedings of the Second International Conference on Embedded Software*, volume 2491 of *LNCS*, pages 213–226, London, UK, oct 2002. Springer-Verlag. (cité page 9)

[10] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *CASES '00 : Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 9–17, New York, NY, USA, 2000. ACM Press. (cité page 9)

[11] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03 : Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press. (cité page 9)

[12] Fridtjof Siebert. Hard real-time garbage-collection in the jamaica virtual machine. In *RTCSA '99 : Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 96, Washington, DC, USA, 1999. IEEE Computer Society. (cité page 9)

[13] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84 : Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, New York, NY, USA, 1984. ACM Press. (cité page 10)

[14] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *LCTES '05 : Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 193–202, New York, NY, USA, 2005. ACM Press. (cité page 10)

[15] Scott Nettles and James O'Toole. Real-time replication garbage collection. *SIGPLAN Not.*, 28(6) :217–226, 1993. (cité page 10)

[16] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997. (cité page 10)

[17] David Gay and Alex Aiken. Language support for regions. In *PLDI '01 : Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 70–80, New York, NY, USA, 2001. ACM Press. (cité page 11)

[18] Filip Pizlo, Jason M. Fox, David Holmes, and Jan Vitek. Real-time java scoped memory ; design patterns and semantics, 2004. http ://www.ovmj.org/documents.html. (cité page 11)

[19] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *RTSS '04 : Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society. (cité page 11)

[20] Pablo Basanta-Val, Marisol García-Valls, and Iria Estévez-Ayres. Agcmemory : a new real-time java region type for automatic floating garbage recycling. *SIGBED Rev.*, 2(3) :7–12, 2005. (cité page 11)

[21] Bruno Blanchet. Escape analysis for javatm : Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6) :713–775, 2003. (cité page 11)

[22] Alexandru Sălcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP '01 : Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 12–23, New York, NY, USA, 2001. ACM Press. (cité page 11)

[23] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. *SIGPLAN Not.*, 34(10) :1–19, 1999. (cité page 11)

[24] Guillaume Salagnac, Sergio Yovine, and Diego Garbervetsky. Fast escape analysis for region-based memory management. In *Proceedings of Abstract Interpretation of Object-Oriented Languages (AIOOL'05), VMCAI'05*, 2005. (cité page 11)

[25] Alexandru Sălcianu and Martin Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAILTR-949, MIT, may 2004. (cité page 11)

[26] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *ISMM '04 : Proceedings of the 4th international symposium on Memory management*, pages 85–96, New York, NY, USA, 2004. ACM Press. (cité pages 11 et 12)

[27] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *PLDI '04 : Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 243–254, New York, NY, USA, 2004. ACM Press. (cité pages 11 et 12)

[28] Nicolas Berthier. Magistère première année : Analyse de la démographie des objets dans les systèmes Java temps-réel, 2006. (cité pages 13 et 24)