Un GC temps réel semi-compactant

Danny Dubé, Marc Feeley, Manuel Serrano

Département d'informatique et de recherche opérationnelle,
Université de Montréal,
C.P. 6128, succursale Centre-ville, Montréal, H3C 3J7
{dube, feeley, serrano}@iro.umontreal.ca

Résumé

Nous avons développé un nouvel algorithme de GC temps réel "dur". Notre algorithme place une limite supérieure sur le temps d'exécution de chaque primitive d'allocation ou d'accès aux objets, selon leur nature. Il s'agit d'un GC compactant. Nous l'avons implanté et avons vérifié qu'une application à allocation intensive tourne seulement environ 3 fois plus lentement avec ce GC qu'avec un GC copiant à deux semi-espaces.

1. Introduction

Nous présentons un nouveau GC (Glaneur de Cellules) temps réel dérivé des GC marque-et-compacte. La présentation sera divisée de la façon suivante: nous définissons ce que nous entendons par GC temps réel; nous introduisons la technique sous sa forme bloquante; nous montrons les transformations à effectuer pour la rendre temps réel; nous évaluons ses performances par rapport à un GC copiant à deux semi-espaces; et, pour conclure, nous faisons une brève discussion de la technique présentée et des ses variantes possibles.

La technique est présentée dans le contexte où la taille du tas est fixée à l'avance et ne change pas au cours de l'exécution.

2. GC temps réel

2.1. Définition

Un grand nombre de techniques de GC sont décrites par Wilson ([Wil92]). Les grandes classes de GC sont: le comptage de références; les GC bloquants; les GC temps réel; les GC générationnels; les GC parallèles; les GC distribués; les GC en milieu non coopératif. Ici, nous nous intéressons aux GC temps réel. Un GC temps réel ([Bak78, DLM+78, Wil92]) est un GC incrémentiel qui garantit une certaine borne supérieure sur le temps pris par l'exécution des opérations primitives d'allocation et d'accès des objets. Dans un GC incrémentiel, le cycle du GC ne se fait pas de façon atomique à l'épuisement de la mémoire libre, mais plutôt est découpé en plusieurs parties qui peuvent être faites au gré des allocations, par exemple. Un GC temps réel fait encore plus: chaque opération primitive sur les objets alloués prend un temps du même ordre que ce que l'on observerait s'il n'y avait pas de travail de collecte qui était fait.

Décrivons de façon plus précise ce que cela veut dire. Nous supposons que les objets doivent être initialisés à leur création. Dans un système qui n'est pas doté d'un GC temps réel, une allocation dure au plus $kn+c$ unités de temps s'il n'y a pas de travail de collecte qui est effectué. k et c sont des (petites) constantes et n est la longueur de l'objet qui est alloué. La borne tient pour n'importe quel objet. Nous considérons donc que dans un système doté d'un GC temps réel, l'allocation du même objet dure au plus $k'n + c'$ unités de temps. Les accès aux champs des objets doivent respecter une règle similaire. Si un accès à un champ demande l'exécution d'au plus a instructions dans le premier système, il doit en demander au plus a' dans le second (a et a' étant des constantes).

2.2. Les difficultés des GC temps réel

La difficulté principale dans un système avec GC temps réel est la coordination entre le travail du GC et celui de l'application.

2.2.1. Les mutations faites par l'application

Un premier problème survient quand le GC est dans la phase de marquage et que l'application change les champs des objets. La situation à éviter est celle où l'application cache un objet vivant aux yeux du GC à l'aide de quelques mutations sur des objets.

Il existe différentes techniques pour éviter cette situation. On implante ce qu'on appelle des barrières en lecture et/ou en écriture, qui sont en fait quelques instructions supplémentaires exécutées lors de la lecture et/ou de l'écriture pour éviter de fausser le travail effectué par le GC.

Le fait de commencer à désigner des objets comme étant retenus *avant* le manque d'espace libre occasionne un conservatisme plus grand chez un GC temps réel que chez un GC bloquant. Certains des objets marqués comme étant retenus seraient peut-être morts avant l'épuisement de l'espace libre. Le choix des barrières a normalement une influence sur le degré de conservatisme du GC ([Wil92]).

2.2.2. Les grands objets

Un autre problème survient si l'on a des objets de grande taille dans un système qui déplace les objets. Pour rester temps réel, le GC ne peut se permettre de déplacer de façon atomique des objets arbitrairement longs. Il doit donc déplacer ces objets par petits morceaux. Le GC doit informer d'une façon ou d'une autre l'application du fait qu'un objet est présentement coupé en deux.

2.2.3. S'assurer de conserver de l'espace libre

Un dernier problème est celui de s'assurer que le GC aura terminé son cycle avant que la zone libre ne se soit épuisée. Il faut alors déterminer une vitesse de travail du GC suffisamment élevée pour qu'il puisse toujours finir son cycle à temps. Le "temps" dont nous parlons ici est l'allocation. La vitesse de travail du GC est donc le nombre de cases élémentaires traitées par case élémentaire allouée.

3. La technique de base (bloquante)

Notre GC est de type marque-et-compacte ([Mor78, CN83, Wil92]). La première phase consiste à trouver et marquer tous les objets accessibles directement ou indirectement depuis les racines de l'application. La deuxième phase consiste à regrouper tous les objets accessibles en une région contiguë du tas et tout l'espace libre en une autre région contiguë. Ainsi, il n'y a pas de fragmentation qui se crée dans le tas. La phase de compactage nécessite une mise à jour des pointeurs afin de refléter la nouvelle position des objets. Nous verrons en détail ces étapes un peu plus loin.

3.1. Disposition du tas

Nous supposons au départ que le tas est divisé en deux régions: celle des identificateurs et celle des objets et de la pile de marquage. La taille des régions ne peut pas être choisie arbitrairement, comme on le verra un peu plus loin.

Un identificateur sert de pointeur vers un objet dans la deuxième région. Généralement, les identificateurs ne sont pas tous utilisés. Les identificateurs inutilisés sont chaînés en une liste d'identificateurs libres. L'identificateur assigné à un objet demeure le même durant toute la vie de l'objet. La région des identificateurs n'est pas compactée. Comme tous les identificateurs ont la même taille, ceci ne cause pas de fragmentation, mais simplement une dispersion des identificateurs vivants. Le fait de ne pas compacter cette région explique le nom de GC "semi-compactant".

La région des objets et de la pile de marquage ne comporte pas de séparation fixe entre ses deux parties. Les objets sont placés un à la suite de l'autre à partir des adresses basses. La pile grossit à partir des adresses hautes. Chaque objet a un champ supplémentaire qui sert à contenir l'adresse de son identificateur. Nous appelons ce champ le pointeur arrière.

Lors de l'allocation d'un objet, un identificateur est réservé dans la première région, l'espace nécessaire est réservé dans la deuxième région, les champs de l'objet sont initialisés, l'identificateur reçoit l'adresse du nouvel objet et l'adresse de l'identificateur est retournée à l'application. L'espace réservé dans la deuxième région est constitué des champs de l'objet, incluant son pointeur arrière, et d'une case mémoire pour la pile. L'application doit toujours accéder aux champs de l'objet via l'identificateur. L'idée d'utiliser des identificateurs a été inspirée de la technique de Brooks ([Bro84]). Sa technique fonctionne avec deux semi-espaces. Celle-ci ajoute un champ supplémentaire devant chaque objet. Ce champ sert à indiquer la position de l'objet et permet donc d'assurer le lien avec celui-ci lorsqu'il change de semi-espace.

La pile n'est utilisée que durant la phase de marquage, mais un espace à son effet est réservé en permanence. Nous jugeons qu'il est préférable que le GC gère lui-même une pile pour le marquage. Premièrement, la pile d'exécution de l'application risque de ne pas être assez grande pour permettre de marquer récursivement tous les objets d'une structure de données très profonde. Deuxièmement, les informations stockées par le GC dans sa propre pile sont beaucoup plus compactes que celles stockées dans la pile d'exécution.

La figure 1 montre un exemple où le tas ne contient que deux paires. Celles-ci forment la liste `(1 2)`. On peut remarquer que le premier champ de chaque paire est le pointeur arrière vers leur identificateur respectif. A l'extrême droite du tas se trouvent les deux cases réservées pour la pile. On peut voir aussi que les identificateurs qui ne sont pas utilisés forment une liste d'identificateurs libres. A chaque allocation, les pointeurs d'allocation et de pile se rapprochent l'un de l'autre.

3.2. Déclenchement du GC

Le GC doit être déclenché lorsque l'application demande l'allocation d'un objet pour lequel il ne reste plus assez d'espace. Il n'est pas nécessaire de vérifier s'il reste un identificateur libre si l'on fixe une taille suffisamment grande pour la région des identificateurs. En effet, supposons que le plus petit objet alloué n'ait qu'un champ. Alors, lors de l'allocation d'un tel objet, on réserve un identificateur, un pointeur arrière, le champ et une case pour la pile. Ce type d'objet étant le plus petit, on sait qu'au maximum, le quart du tas sera occupé par des identificateurs. Donc, en prenant la région des identificateurs trois fois plus petite que celle des objets, on élimine ce test. Il ne reste que le test pour vérifier s'il y a assez d'espace pour l'objet dans la deuxième région.

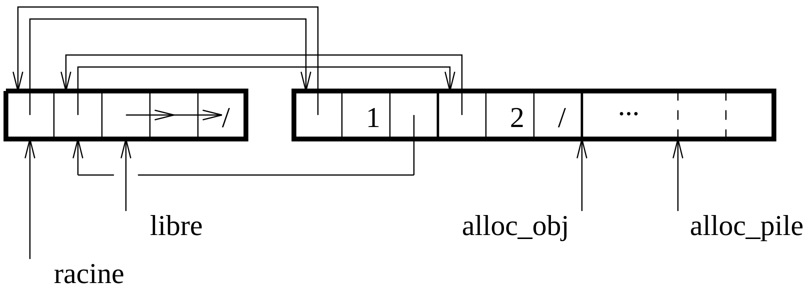

Figure 1: Illustration du tas contenant la liste `(1 2)`

Légende: racine: une racine de l'application
libre: la tête de la liste des identificateurs libres
alloc_obj: le pointeur d'allocation des objets
alloc_pile: début de la zone réservée pour la pile

3.3. Description de l'algorithme

3.3.1. Le marquage

La première phase du GC est le marquage. L'algorithme commence par marquer tous les objets accessibles directement des racines. Le fait de marquer un objet cause son ajout dans la pile de marquage. Les objets qui sont sur la pile sont des objets dont on dira qu'ils n'ont pas encore été *fouillés* à la recherche de références à des objets non encore marqués. Par la suite, l'algorithme extrait un objet de la pile, le fouille et marque tous les objets nouvellement atteints. Le processus de marquage se poursuit tant que la pile n'est pas vide.

Connaissant la façon de marquer, on peut comprendre pourquoi nous avons choisi de réserver une case de pile par objet alloué. En effet, il est possible que toute la pile sauf une case soit utilisée. Par exemple, un vecteur qui contient des références sur tous les autres objets du tas est une structure de données qui fait en sorte que la pile est utilisée presque complètement.

Nous utilisons une astuce simple et relativement portable pour éviter de réserver de l'espace supplémentaire pour le bit de marquage. Il est raisonnable de supposer que les mots de la machine sont à des adresses paires. On peut donc utiliser le bit le moins significatif du pointeur arrière comme bit de marquage. Il serait également possible d'utiliser le bit le moins significatif de l'identificateur mais cela serait moins efficace pour les deux raisons suivantes: lors du compactage, le test du bit de marque demanderait une référence de plus à la mémoire; les accès ordinaires aux champs de l'objet demanderaient de masquer le dernier bit de l'identificateur.

3.3.2. Le compactage

La deuxième phase du GC est le compactage. Les objets marqués sont regroupés au début du tas comme s'ils avaient tous glissé vers les adresses basses. Leur ordre est donc préservé. Deux pointeurs partent du début de la région des objets. Le premier est le pointeur de source et sert à rechercher les objets marqués. Le second est le pointeur de destination et indique la nouvelle position de chaque objet marqué. Si l'objet pointé par le pointeur source est marqué, il est déplacé à l'endroit pointé par le pointeur de destination. De plus, son identificateur est mis à jour pour qu'il pointe vers la nouvelle position de l'objet. Bien sûr, l'identificateur de l'objet est retrouvé rapidement grâce au pointeur arrière. Si l'objet n'est pas marqué, son identificateur est libéré et ajouté à la liste des identificateurs libres. Durant le compactage, les objets vivants sont comptés afin de pouvoir réserver la bonne taille de pile pour le prochain cycle de GC et de mesurer correctement la quantité de mémoire libre.

4. La technique temps réel

Nous allons tout d'abord décrire les barrières en lecture et en écriture qui doivent être ajoutées à la technique de base; ensuite, nous montrerons de quelle façon l'algorithme du GC est découpé; nous parlerons de l'aspect synchronisation entre le GC et l'application; enfin, nous décrirons deux optimisations servant à améliorer l'efficacité du GC.

Il y a certaines contraintes qui doivent être imposées à l'application en plus des barrières pour que cette technique puisse fonctionner. Premièrement, le nombre de racines de l'application doit être assez petit. Deuxièmement, l'application ne doit pas tenter de conserver plus qu'une certaine fraction du tas en objets vivants. Cette dernière contrainte est commune à toutes les techniques de GC temps réel, à l'exception du comptage de références.

4.1. Les barrières en lecture et en écriture

4.1.1. L'indirection via l'identificateur

La barrière la plus importante est déjà incluse dans la technique de base, c'est l'utilisation des identificateurs. En effet, ces identificateurs permettent au GC de déplacer un objet et de faire suivre "toutes" les références à cet objet en une seule affectation, donc de façon atomique.

4.1.2. Barrière en écriture

Nous utilisons une barrière en écriture durant le marquage pour éviter que le GC ne collecte des objets accessibles. A l'instant où un objet A non marqué est stocké dans un objet B déjà marqué, A est lui-même marqué. Cette barrière en écriture est similaire à celle proposée dans [DLM+78], et est appelée barrière en écriture à mise à jour incrémentielle (*incremental update write barrier*). Cette barrière est moins conservatrice qu'une barrière en lecture ([Bak78, Bro84, Nil88, Bak92, NS92]), qui ne laisse accéder l'application à aucun objet non marqué, ou qu'une barrière en écriture de type *snapshot-at-beginning* ([Yua90]), qui conserve tout objet accessible au début du cycle de GC. Il y a une autre barrière en écriture de type *incremental update* qui est décrite dans [Ste75]. L'action de cette barrière consiste à noter que l'objet déjà fouillé qui vient de recevoir l'objet non marqué devra être fouillé à nouveau ultérieurement. Cette barrière est encore moins conservative que celle que nous avons adoptée ([Wil92]), mais elle est plus difficile à implanter. En pratique, le fait d'être moins conservateur cause la survie de moins d'objets inaccessibles à la fin du cycle.

4.1.3. Barrière pour les longs objets

Il y a une dernière barrière en lecture et en écriture qui doit être en place pour les longs objets. La définition de long objet peut varier d'une implantation à l'autre. Lorsqu'un long objet est en cours de copiage, le début de l'objet se trouve à la nouvelle position et le reste se trouve à la position originale. Durant un tel copiage, le GC déclare l'identificateur du long objet. La barrière consiste simplement à comparer l'identificateur de l'objet accédé à celui qui est déclaré. En cas d'égalité, il faut vérifier dans laquelle des deux parties se trouve la case qui doit être accédée.

4.2. L'algorithme du GC

4.2.1. La banque de temps

Avant de décrire l'algorithme du GC, il est important de parler de la *banque de temps* du GC. L'idée consiste à maintenir un compteur qui indique la quantité de travail que peut effectuer le GC. A chaque allocation, une certaine quantité de temps est ajoutée à la banque, le GC travaille tant que sa banque est positive et ensuite l'objet demandé est réellement alloué. Nous verrons dans la prochaine sous-section comment le temps est géré lors des allocations et lors du travail du GC.

Les opérations avec un astérisque sont des opérations qui coûtent du temps au GC, c'est-à-dire qui enlèvent du temps de sa banque.

4.2.2. L'algorithme proprement dit

L'algorithme est décrit à la façon d'un automate fini. C'est à chaque “transition” qu'un test est fait pour vérifier que la banque de temps n'est pas épuisée. L'état initial est l'état 1.

1. Préparer le cycle du GC. Déclarer à l'application que le GC est en phase de marquage. Aller à 4.
2. S'il n'y a pas d'objet sur la pile de marquage, aller à 4. Si l'objet à fouiller est long, aller à 3. Sinon, fouiller* l'objet et aller à 2.
3. Fouiller* autant de champs que possible dans l'objet. S'il a été fouillé au complet, aller à 2. Sinon, aller à 3.
4. Marquer les objets directement accessibles depuis les racines. Si la pile est vide, aller à 5. Sinon, aller à 2.
5. Fin du marquage. Début du compactage. Aller à 6.
6. S'il n'y a plus d'objets à déplacer, aller à 9. Sinon, aller à 7.
7. Si l'objet n'est pas marqué, libérer* son identificateur, sauter par-dessus l'objet et aller à 6. Si l'objet est petit, le déplacer*, changer la valeur de son identificateur et aller à 6. Sinon, changer* la valeur de son identificateur, déclarer que l'objet est en déplacement et aller à 8.
8. Déplacer* autant de cases de l'objet que possible. Mettre à jour le point de coupure déclaré à l'application. Si l'objet est complètement déplacé, cesser de le déclarer coupé et aller à 6. Sinon, aller à 8.
9. Clore le cycle du GC. Aller à 1.

4.2.3. Observations

L'algorithme nécessite en plus un test spécial lorsque le tas est vide. En effet, il ne coûte aucun temps au GC pour faire un cycle sur un tas vide. Un moyen simple d'éviter le test consiste à introduire un objet toujours vivant dans le tas dès son initialisation.

L'état qui fouille les racines peut être atteint plus d'une fois par cycle. Ceci est nécessaire à cause du fait que des allocations et des changements dans les racines se font lors des pauses du marquage. De plus, il est nécessaire de fouiller toutes les racines d'un coup. Sinon, il n'est pas possible de savoir s'il n'y a vraiment plus d'objets à marquer. Ceci explique pourquoi il faut que l'application n'ait qu'un petit nombre de racines. Malgré les allocations faites par l'application durant le marquage, le rythme de marquage du GC est réglé de façon à assurer la terminaison du marquage.

Il est possible d'éliminer la limite sur le nombre de racines. On peut avoir un nombre non borné de racines et les fouiller de façon incrémentielle. Toutefois, ceci nécessite la mise en place d'une barrière en écriture sur les racines. Toute application qui utilise une pile ou un grand nombre de variables globales risque d'avoir à prendre une telle mesure.

Nous avons préféré que cette fouille des racines soit atomique et ne coûte aucun temps au GC. Nous aurions pu considérer que cette opération avait un coût. Auquel cas, nous aurions un contrôle plus serré sur les bornes de temps d'exécution, mais aussi l'analyse du rythme de travail du GC serait plus complexe (pour l'analyse, voir plus loin).

Pour ce qui est du compactage, les modifications dans la version temps réel sont assez simples. Là encore, le rythme de compactage est suffisamment élevé pour permettre au pointeur de source de rejoindre le pointeur d'allocation. A la fin du compactage, le pointeur d'allocation est modifié pour indiquer la même position que le pointeur de destination.

Afin que le GC ne conserve pas trop d'objets inaccessibles, nous avons choisi d'allouer les objets non marqués pendant le marquage. Ceci laisse une chance aux objets de courte durée de devenir inaccessibles *avant* que le GC ne les marque. Par exemple, un groupe d'objets fraîchement alloués peut être détenu temporairement par une racine et abandonné ensuite. Si les racines ne sont pas fouillées durant ce temps, ces objets risquent de ne jamais être atteints par le marquage. Durant le compactage, toutefois, les objets doivent être alloués marqués, étant donné que le GC ne peut avoir aucune preuve qu'il sont devenus inaccessibles avant d'avoir fait un nouveau marquage.

4.3. Synchronisation du GC avec l'application

4.3.1. Contrainte sur la quantité d'objets vivants

A tout moment, l'application ne doit pas tenter de conserver plus qu'une certaine fraction du tas en objets accessibles. En fait, plus le tas contient d'objets accessibles, plus le temps en pire cas pris par les opérations primitives est grand. Si on calcule pour l'application le pourcentage d'occupation maximale du tas en objets alloués accessibles, on peut alors calculer les bornes de temps sur chaque opération primitive.

4.3.2. Comptabilité du temps du GC

La façon de compter le temps lors des allocations et du travail du GC a une grande importance pour assurer un bon fonctionnement du GC. La vitesse du GC est réglée par un paramètre que nous appelons le *ratio*. A chaque allocation, nous ajoutons à la banque de temps du GC le produit entre le ratio et l'espace demandé pour le stockage de l'objet. Nous avons décidé que l'espace qui compterait serait celui requis par les champs de l'objet, le pointeur arrière et la case réservée pour la pile. Bref, c'est l'espace qui est consommé par un objet dans la région des objets et de la pile de marquage. De façon analogue, le temps qui est requis pour *fouiller* un objet (et non pas pour le marquer) est l'espace pris par cet objet dans la région en question. Pareillement, pour le traitement d'un objet durant la phase de compactage. Que l'objet soit accessible ou non, son traitement consomme une quantité de temps égale à l'espace qu'il occupe dans la région des objets. Nous ne comptons pas l'espace pris par l'identificateur étant donné que nous supposons que le nombre d'identificateurs a été choisi de façon à ce qu'ils ne puissent être épuisés (du moins, sans épuisement de l'espace dans la région des objets).

Cette *comptabilité* du temps a été adoptée parce qu'elle simplifie beaucoup le calcul du ratio. En effet, le ratio indique, d'une certaine façon, combien de cases mémoire sont traitées pour chaque case allouée. Ainsi, il est simple de compter combien d'unités de temps il faut pour la phase de compactage. Chaque objet doit être traité une et une seule fois. L'ensemble des objets à traiter est constitué de ceux existant au début du cycle du GC, ceux alloués durant le marquage et ceux alloués durant le compactage. Le temps requis pour cette phase est donc l'espace qu'occuperont ces objets à la fin de la phase. On peut calculer aussi combien d'unités de temps sont requises par le marquage. Chaque objet doit être traité (fouillé) *au plus* une fois. L'ensemble des objets qui sont possiblement à considérer est constitué de ceux qui existaient au début du cycle, plus ceux qui ont été alloués durant le marquage.

4.3.3. Calcul du ratio

Nous allons maintenant expliquer la façon de calculer un ratio. Il est important de noter que le temps requis pour faire le marquage ne peut être plus long que celui requis pour faire le compactage. Aussi, il faut rappeler que l'application ne doit jamais conserver plus qu'une certaine fraction du tas en objets vivants. Soit $\alpha, 0 < \alpha < 1$, la fraction maximale occupée par les objets vivants. Nous allons montrer que si nous choisissons un ratio R égal à $\frac{5+3\alpha}{2-2\alpha}$, l'occupation du tas au début de chaque cycle sera au maximum $\frac{1+\alpha}{2}$.

Preuve par induction. *Base.* Au départ, le tas est complètement vide, donc l'occupation du tas est au plus $\frac{1+\alpha}{2}$. *Pas.* Il faut d'abord montrer que le cycle du GC aura eu le temps de se terminer avant que l'espace libre ne se soit épuisé. Par hypothèse d'induction, au maximum $\frac{1+\alpha}{2}$ du tas est occupé, au début du cycle. Soit v, $0 \leq v \leq \alpha$, la fraction du tas occupée par les objets vivants et m, $0 \leq v + m \leq \frac{1+\alpha}{2}$, la fraction du tas occupé par les objets morts. (Par abus de langage, nous parlons de l'occupation du tas, mais, en fait, c'est de l'occupation de la région des objets dont nous parlons réellement.) Clairement, au moins $\frac{1-\alpha}{2}$ du tas est libre. Après l'allocation de $\frac{1-\alpha}{2}$ du tas, nous prétendons que le GC a reçu assez de temps pour compléter son cycle. Après l'allocation de

$\frac{1-\alpha}{2}$ du tas, grâce au ratio, le GC a reçu du temps pour traiter $\frac{5+3\alpha}{4}$ du tas. Le temps sera distribué en $\frac{1+3\alpha}{4}$ pour le marquage et 1 pour le compactage.

Le temps requis en pire cas pour le marquage est celui demandé si tous les objets vivants au début sont marqués et tous les objets alloués durant le marquage sont marqués. Comme le marquage n'est jamais plus long que le compactage, au plus $\frac{1-\alpha}{4}$ du tas est alloué durant le marquage. Le temps en pire cas requis par le marquage est donc $v + \frac{1-\alpha}{4} \leq \frac{1+3\alpha}{4}$.

Le temps requis en pire cas pour le compactage est celui demandé pour traiter tous les objets existants au début et tous les objets alloués durant le cycle. Ce qui représente un temps de $v + m + \frac{1-\alpha}{2} \leq 1$. Le cycle du GC peut se terminer avant que l'espace libre ne soit épuisé. Du moins, pour ce cycle.

Il reste à montrer qu'à la fin de ce cycle, il y aura au maximum $\frac{1+\alpha}{2}$ du tas qui sera occupé. En pire cas, les objets qui peuvent survivre au cycle sont ceux qui étaient vivants au début et ceux alloués durant le cycle. Il y en avait v au début. Au plus $\frac{1-\alpha}{2}$ ont été alloués durant le cycle. Donc, l'occupation au début du prochain cycle sera au plus $v + \frac{1-\alpha}{2} \leq \frac{1+\alpha}{2}$. Ce qui termine la preuve.

Bien entendu, dans une implantation concrète, il est préférable de choisir un ratio entier. Il suffit de prendre le plus petit entier supérieur ou égal à $\frac{5+3\alpha}{2-2\alpha}$.

4.4. Deux améliorations

Il est certain qu'un GC temps réel impose une pénalité sur la performance globale d'un système. Nous proposons deux améliorations qui permettent d'accroître l'efficacité du GC.

La première est une amélioration pour réduire le temps pris par le compactage. Normalement, un objet alloué durant le compactage l'est à la position du pointeur d'allocation, à la suite du dernier objet alloué. De plus, il est alloué marqué. Et, éventuellement, le pointeur de source du compactage parvient à cet objet et le déplace à sa nouvelle position. Là où on peut faire une économie, c'est lorsqu'il y a assez d'espace entre le pointeur de source et le pointeur de destination pour loger l'objet. En l'allouant immédiatement parmi les objets déplacés, on évite de le déplacer inutilement. La figure 2 illustre cette amélioration. Au lieu d'allouer l'objet marqué (en noir) à l'endroit pointé par *alloc*, il est alloué non marqué (en blanc) à l'endroit pointé par *dest*.

L'effet de cette première amélioration est de faire terminer les cycles de GC plus tôt. Ce qui permet, en général, d'avoir une occupation du tas moins grande à la fin de chaque cycle. Pour que cette amélioration soit profitable, il faut implanter aussi la suivante: retarder le lancement

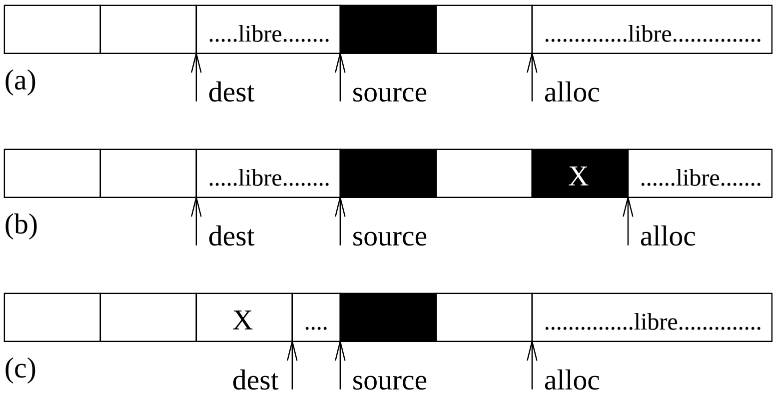

Figure 2: Objet alloué immédiatement parmi les objets déplacés durant le compactage.
(a) Etat original du tas
(b) Allocation de X de la façon habituelle
(c) Allocation de X directement parmi les objets déplacés

du cycle si le tas n'est pas encore assez plein. Comme on a vu à la sous-section précédente, le ratio R est calculé de façon à ce qu'il assure un bon fonctionnement du GC étant donnée une occupation initiale pouvant aller jusqu'à $\frac{1+\alpha}{2}$. Donc, il est inutile de déclencher le nouveau cycle tant que l'occupation est sous ce seuil. La technique pour régler le départ du GC consiste simplement à enlever de la banque de temps du GC la bonne quantité. Par exemple, s'il reste Δ en espace libre avant d'atteindre le seuil où le GC doit obligatoirement se déclencher, on enlève $R\Delta$ de sa banque de temps. Cette dernière amélioration ne nécessite pas que la première amélioration soit implantée.

5. Résultats expérimentaux

Afin d'illustrer les performances de notre technique, nous avons fait quelques tests expérimentaux. La technique de base et la technique temps réel sont mises en comparaison avec le GC copiant à deux semi-espaces de Cheney ([Che70]).

Les tests ont été effectués sur une machine DEC AXP 3000, dotée d'un microprocesseur DEC Alpha tournant à 150 MHz sous OSF/1 version 3.1 et possédant 160 Moctets de mémoire vive. L'application utilisée pour faire les tests était un interprète Scheme exécutant le calcul de la fonction de Fibonacci sur 20 (voir figure 3). L'interprète est plutôt rudimentaire et il utilise une grande quantité d'objets alloués dans le tas pour effectuer son travail. Il constitue donc une application à allocation

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Figure 3: Fonction de Fibonacci

intensive.

Les trois algorithmes de GC ont été testés sur un tas comportant 50 000 cases élémentaires (ou champs). Ils ont été testés sous différents taux d'occupation du tas (la proportion du tas occupée par des objets vivants). L'occupation du tas étant créée artificiellement en commençant par allouer une liste de longueur adéquate avant de lancer le test. Un soin particulier a été porté à la réalisation du taux d'occupation désiré pour ne pas fausser les résultats obtenus. En effet, la taille des objets pour notre GC a été augmentée artificiellement pour que l'allocation de listes ayant le même nombre d'éléments cause le même taux d'occupation dans chaque système. Les objets vivants provenant de l'exécution du test sont très peu nombreux comparativement à la taille de ces listes et n'influent presque pas sur le taux d'occupation.

L'ajustement fait sur la taille des paires permet de comparer les GC à des taux d'occupation égaux avec un nombre égal d'objets, mais du coup notre GC a été désavantagé. Afin de donner une idée des ajustements faits: une paire a 3 champs dans notre implantation du GC de Cheney (le type, le *car*, le *cdr*), ce qui compte pour 6 champs à cause de l'utilisation de deux semi-espaces. Dans notre GC, une paire demande 5 cases en tout. Les paires dans notre technique ont donc reçu un champ supplémentaire.

La figure 4 résume les résultats obtenus. Pour chacun des GC, les temps moyens pour effectuer une allocation sont indiqués sous différents taux d'occupation. Les temps proviennent de mesures en cycles machine, obtenues à l'aide du "*process cycle counter*" de l'Alpha (compteur de cycles du processus). Ce compteur indique à l'instruction près le temps d'exécution écoulé. Les temps sont relatifs au temps pris par le GC copiant avec un tas vide, celui-ci étant de 94 cycles. Pour le GC temps réel, un écart-type est donné avec la moyenne. Il nous indique à quel point les temps varient d'une allocation à l'autre. Les maxima en temps d'exécution par allocation auraient été intéressants à mesurer. Malheureusement, les interruptions perturbent le compteur et rendent cette mesure inutile. On peut constater que les écarts-type sont assez

Occupation du tas	Temps d'allocation Copiant	non-TR	TR	
0%	1.0	2.2	5.2	$\sigma = 3.5$
5%	1.1	2.4	5.4	$\sigma = 4.4$
10%	1.3	2.6	6.3	$\sigma = 4.9$
15%	1.5	2.8	7.2	$\sigma = 4.9$
20%	1.7	3.1	8.5	$\sigma = 4.7$
25%	1.9	3.4	7.9	$\sigma = 5.6$
30%	2.2	3.7	9.0	$\sigma = 5.0$
35%	2.5	4.2	10.4	$\sigma = 4.8$
40%	2.9	4.7	12.4	$\sigma = 5.4$
45%	3.3	5.1	12.3	$\sigma = 5.5$
50%	3.8	5.7	14.1	$\sigma = 6.2$
55%	4.4	6.5	15.0	$\sigma = 6.6$
60%	5.2	7.5	16.9	$\sigma = 6.6$
65%	6.1	8.8	19.5	$\sigma = 7.5$
70%	7.5	10.5	21.7	$\sigma = 7.6$
75%	9.2	12.9	25.6	$\sigma = 8.1$
80%	12.0	16.2	31.5	$\sigma = 9.8$
85%	16.4	22.3	40.8	$\sigma = 11.4$
90%	25.2	32.6	57.9	$\sigma = 14.6$
95%	50.6	60.6	107.0	$\sigma = 19.2$

Figure 4: Temps d'allocation normalisés pour chacun des trois GC (1 = 94 cycles)

réduits, ce qui laisse croire que les temps pris pour faire une allocation sont réguliers (nous avons observé une distribution normale des temps d'allocation).

L'application complète est ralentie par un facteur d'environ 2 lorsqu'elle utilise le GC de base comparativement au GC copiant. Elle est ralentie d'un facteur d'environ 3 avec le GC temps réel, toujours par rapport au GC copiant. Ce ralentissement est somme toute acceptable, si on prend en compte que l'intention au départ est d'avoir une représentation des données assez compacte. L'utilisation d'objets ayant 3 champs et plus permet de faire une économie d'espace par rapport à un GC à deux semi-espaces. Ce GC est utilisé dans un interprète Scheme compact que nous avons développé pour un micro-contrôleur à 8 bits.

6. Conclusion

Les avantages de notre technique de GC temps réel sont qu'elle compacte les objets du tas, qu'elle est assez économe en espace mémoire pour un GC temps réel compactant, qu'elle ne demande pas de matériel spécialisé et qu'elle est vraiment temps réel.

Nous considérons que le compactage est un avantage appréciable. Typiquement, les applications qui demandent des bornes de temps serrées sur les opérations primitives ne sont pas exécutées sur des systèmes à mémoire virtuelle. Donc, elles ne peuvent se permettre d'allouer un tas énorme qui supporterait la fragmentation ou la présence d'un semi-espace constamment inutilisé.

Elle a les désavantages suivants: le nombre de racines doit être borné (et assez petit) pour permettre au GC d'être véritablement temps réel; il faut trouver la consommation maximale de mémoire d'une application (ou une borne supérieure) pour pouvoir calculer un ratio garantissant un bon fonctionnement; étant compactante, notre technique n'est probablement pas parmi les techniques temps réel les plus efficaces.

Les aspects suivants de la technique sont les plus susceptibles d'avoir un impact sur l'efficacité de notre technique. L'allocation est assez coûteuse. En effet, malgré qu'il s'agisse d'un algorithme compactant, il ne suffit pas d'avancer un pointeur dans le tas pour faire une allocation. Il y a aussi le pointeur de la zone réservée pour la pile qu'il faut faire avancer et l'identificateur qu'il faut enlever de la listes des identificateurs libres et initialiser. Les accès aux objets sont assez coûteux à cause des diverses barrières. Même si la barrière en écriture est nécessaire pour assurer la coordination entre l'application et le marquage fait par le GC, les deux autres barrières, toutefois, sont les conséquences du compactage et du temps réel. Si la technique est utilisée dans un système à mémoire virtuelle, l'éparpillement des identificateurs dans leur région peut diminuer la localité de référence. Finalement, un GC compactant déplace les objets et doit parcourir l'ensemble des objets *alloués*, contrairement à un GC copiant comme celui de Cheney ([Che70]) qui ne travaille qu'avec les objets *vivants*, ou même, comparativement à un GC comme le *treadmill* ([Bak92]) qui ne travaille qu'avec les objets *vivants* sans les déplacer.

La technique a un potentiel pour d'autres applications. Nous les mentionnons brièvement. La technique de base peut être très facilement adaptée pour fonctionner dans un milieu non coopératif (à la C, [Bar88, BW88, Del90]). Ceci vient du fait que la position de l'identificateur associé à un objet ne change pas et qu'il est simple de vérifier qu'une adresse référence un objet véritable. La technique se prête bien à l'ajout de générations. En effet, le compactage se fait en glissant les objets un à la suite de l'autre. Les objets sont donc toujours ordonnés selon leur âge. La technique permet d'installer des frontières mobiles entre les générations.

Bibliographie

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21 (4): 280-294, avril 1978.

[Bak92] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27 (3): 66-70, mars 1992.

[Bar88] Joel Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, Février 1988.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, 108-113, août 1984. Austin, Texas.

[BW88] Hans-Juergen Boehm et Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18 (9) : 807-820, Septembre 1988.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13 (11): 677-678, novembre 1970.

[CN83] Jacques Cohen et Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5 (4): 532-553, octobre 1983.

[Del90] Delacour, Vincent. Un gestionnaire mémoire indépendant pour K2. *Rapport de recherche de l'ICSLA*, LIX, Ecole polytechnique, Laboratoire d'informatique, 23-32, octobre 1990.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, et E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21 (11) : 966-975, Novembre 1978.

[Mor78] Morris, F. L. A time- and space-efficient garbage compaction algorithm. *Comm. ACM* 8 (1978), 662-665.

[Nil88] Nilsen, Kelvin. Garbage collection of strings and linked data structures in real time. *Software, Practice and Experience*, 18 (7): 613-640, juillet 1988.

[NS92] K. D. Nilsen et W. J. Schmidt, A High-Performance Hardware-Assisted Real-Time Garbage Collection System. En soumission.

[Ste75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18 (9): 495-508, septembre 1975.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. Dans Yves Bekkers et Jacques Cohen, éditeurs, *International Workshop on Memory Management*, numéro 637 dans Lecture Notes in Computer Science, pages 1-42, St Malo, France, Septembre 1992. Springer-Verlag.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11: 181-198, 1990.