International Journal of Engineering Research and Development
e-ISSN: 2278-067X, **p**-ISSN: 2278-800X, www.ijerd.com
Volume 7, Issue 5 (June 2013), PP.99-103

Solution A La Gestion Des Objets Java Pour Des Systèmes Embarqués

Laila Moussaid[1], Mostafa Hanoune[2]
[1,2]Laboratoire TIM, Faculté des sciences Casablanca.

Résumé:- Le langage Java offre un avantage considérable en terme de facilité de l'utilisation et la portabilité , ainsi la gestion transparente de la mémoire dans Java se fait de plusieurs façon mais on observe que deux grandes familles peuvent être présentées : la première famille est basé sur les analyses statiques et la gestion de mémoire en régions et la deuxième consiste en l'implantation d'un mécanisme de gestion de la mémoire basé sur l'utilisation des Ramasse Miettes. Notre Travail se focalise sur la deuxième famille .en effet, les algorithmes du RM sont nombreux mais deux algorithmes qui sont implantés dans plusieurs machines virtuelles java dans le système embarqué ou une combinaison basé sur ces deux algorithmes : RM par comptage de références et RM marquage-balayage. Cependant, ces deux mécanismes du RM ne sont pas efficaces sur certains types de programmes, pour lesquels elles engendrent des fuites de mémoire pouvant conduire à la saturation de l'espace mémoire et ne traitent pas le problème des structures cycliques. Après une étude des techniques de gestion de la mémoire dynamique dans les systèmes embarqués et un ensemble d'études de cas, nous présentons notre solution à ce problème, en utilisant notamment un nouveau algorithme du ramasse-miettes

Mots-clés:- système embarqué, ramasse-miettes, gestion de la mémoire, J2ME, KVM

I. INTRODUCTION

L'expertise des développeurs pour les environnements embarqués s'oriente, dans ces dernières années vers des nouvelles technologies, qui nécessitent des environnements d'exécution plus complexes. À cet égard, Java constitue un excellent exemple oriente objet Conçu pour pouvoir être utilisé partout, l'environnement Java s'est alourdi de nombreuses fonctionnalités et n'est pas utilisable dans les équipements embarqués,pour ceci,Sun a proposé des versions dégradées et incompatibles comme J2ME [1] ou Java Card [2].
Mais dans ce dernier cas, la règle d'or de Java 'Compile once, run everywhere ' n'est plus respectée ,mais dans J2me la règle est valable, et la gestion des objets en mémoire est piloté par la technique 'ramasse miettes : RM '(en anglais : garbage collector).

En effet, RM désigne l'ensemble des techniques permettant d'automatiser la détection des zones mémoires inutilisées (garbage) en vue de leur réutilisation [3] [4].
Le concept de RM provient des premières implantations du langage Lisp, dans les années 60 [5].

L'objectif d'un RM est de déterminer, parmi l'ensemble des objets présents en mémoire, lesquels sont encore vivants et lesquels sont devenus inaccessibles, de façon à recycler l'espace occupé par les objets morts. Ainsi, l'environnement d'exécution se réapproprie automatiquement la mémoire lorsque le programme ne l'utilise plus. Cette section présente le principe général des deux techniques principales utilisées à cet effet :
le marquage-balayage, et le comptage de références. Il en existe bien sûr d'innombrables variantes et améliorations, , mais tous les RM utilisent une combinaison de ces deux techniques [6]. La première ramasse les objets morts ou inutiles périodiquement dans le tas, tout en prenant soin de conserver les objets vivants. La seconde cherche plutôt à détecter des objets morts pour les recycler immédiatement.

II. ETAT DE L'ART

Dans Java, la gestion dynamique de la mémoire se fait automatiquement .en effet,la récupération des objets java inaccessibles ou morts dans la mémoire peut être assurée à l'exécution par des mécanismes de RM . Après une rapide présentation des principaux algorithmes de RM, deux principales adaptations aux systèmes embarqués seront présentées : le marquage balayage et le comptage de références.

A. l'algorithme comptage de références.

Cette technique 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 .une fois le compteur devient nul, l'objet n'est plus référencé et la mémoire qu'il occupe peut être récupérée.
Cet algorithme présente l'avantage d'être très simple à mettre en oeuvre : 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.

En revanche, cette technique présente une limitation importante pour les structures cycliques. La fig 1 illustre cet inconvénient majeur de l'algorithme comptage de références.

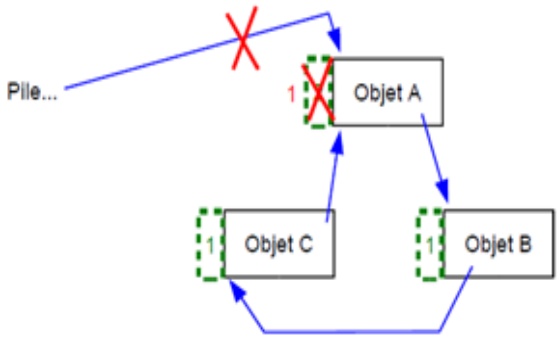


fig 1 : structure cyclique

B. l'algorithme marquage balayage.

L'algorithme 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. 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.

III. CONTIBUTION ET EXPERIENCES

Pour résoudre le problème des structures cycliques et minimiser le nombre des compteurs des objets pour le RM par comptage de références, une nouvelle approche est de découper la mémoire en des "pages" de 8 à 16ko. Cet algorithme repose sur l'idée que les pages les plus utilisées dans un passé proche ont plus de chance d'être à nouveau utilisées dans le futur proche et par conséquent le RM par marquage-balayage et compactage de la mémoire sera appliqué dans les pages les plus utilisées dans un passé proche, et le RM par marquage-balayage sans compactage de la mémoire dans les pages les moins utilisées dans le passé proche [8]. On associe donc un compteur d'utilisation à toutes les pages afin de distinguer les pages les plus utilisées des pages moins utilisées. Dans [8] nous avons conçu cette approche sous forme ci-dessous (fig 2).

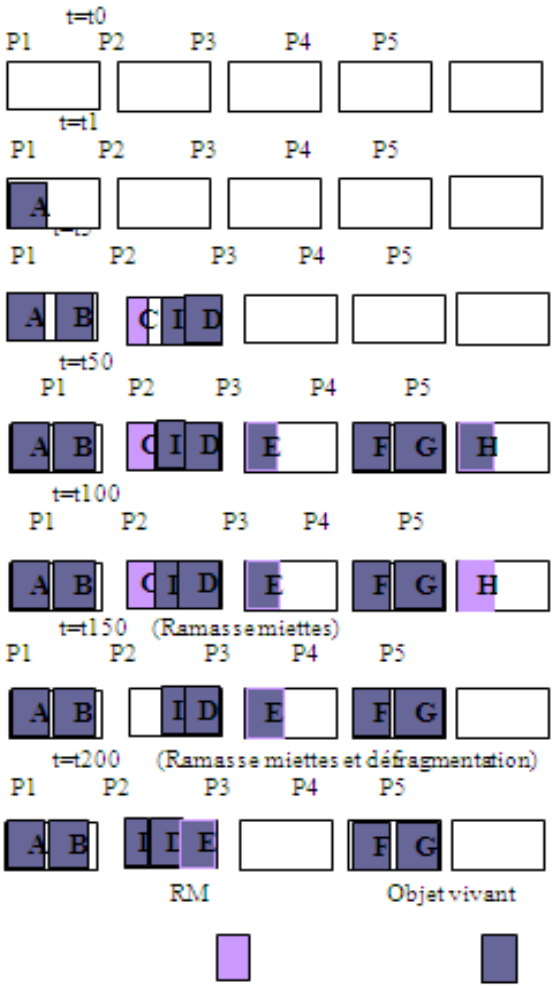


Fig2: Opération de RM avec marquage balayage et défragmentation

Cette seconde phase du travail a consisté à analyser différents comportements des objets java et surtout optimisation de la mémoire avec l'exploitation de notre approche dans la kilobyte machine virtuelle.

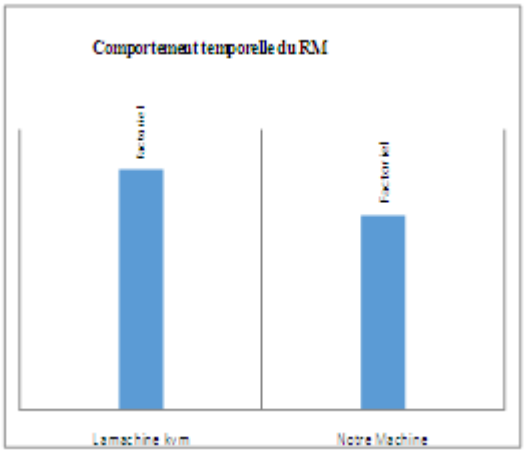
Les Programmes qui ont été choisis pour cela sont, deux applications très connues dans le monde du mathématiques :-Fibonacci, -Factorielle.

ces applications qui ont l'avantage d'utiliser les modèles de programmation courants (structure cyclique, récursivité, forte utilisation de la mémoire dynamique...) :c'est pourquoi ils représentent bien le genre de programmes Java que la solution à proposer devrait pouvoir gérer.

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 utilisant kvm de sun . Dans un second temps, les variations de son occupation mémoire lorsque celle-ci est gérée par la nouvelle approche proposée [9] et avec un RM classique de KVM de sun ont été comparées. Ceci a permis de confirmer et préciser l'efficacité de notre proposition .

A. Comparaison de comportement temporelle du RM

Suite à des tests d'un programme qui calcul la suite de Fibonacci (les 10 premiers termes), nous avons relevé le temps d'exécution du RM de la machine virtuelle KVM et Notre machine virtuelle proposé (Fig4) de la même façon pour le programme de Factorielle cas12 (Fig3)



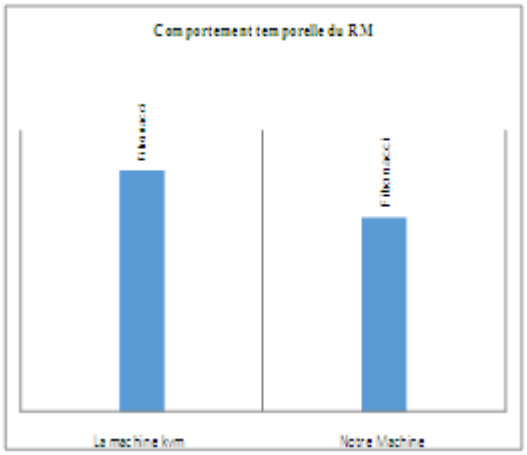
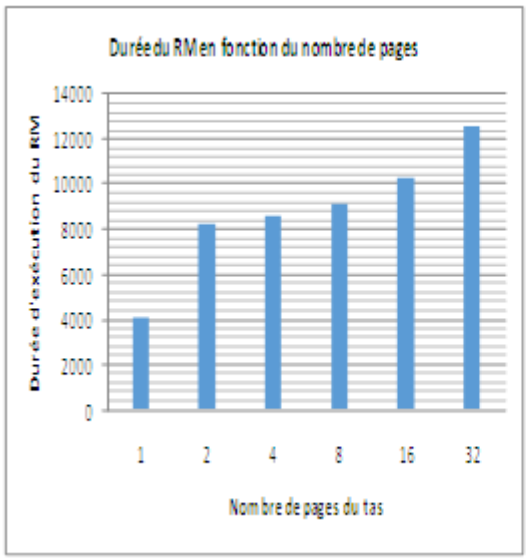


Fig 3: Durée d'exécution du RM pour Factorielle **Fig 4:** Durée d'exécution du RM pour Fibonacci

Nous remarquons que la durée d'exécution de notre RM égale à 8268µs pour Fibonacci et 8336 µs pour Factorielle sont inférieure à celle du RM de la KVM classique avec les valeurs respectivement 10281µs et 10299 µs .

B. Variations des nombre de pages

En fonction du nombre de pages à parcourir, le temps d'exécution du RM est différent.
Pour les mêmes programmes de la section précédente, nous avons relevé les données pour illustrer les graphes Fig5 et Fig 6 ci-dessous.



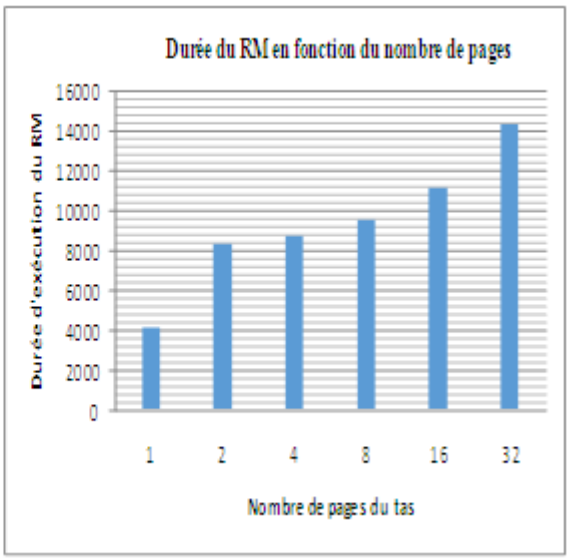


Fig 6 : temps d'éxécution du RM en fonction du nombre de pages pour Factorielle

Fig 6 : temps d'éxécution du RM en fonction du nombre de pages pour Factorielle

Effectivement,Nous observons que le nombre de pages adaptés pour exécuter les deux programmes participe dans l'optimisation temporelle du RM . Ceci confirme notre approche qui propose au lieu de parcourir toutes les pages du Tas ,il suffit de parcourir les pages les plus utilisées dans un passé proche [10].

IV. CONCLUSIONS

Le résultat obtenu concernant les deux programmes jugés compatibles avec notre approche [11],il est donc envisageable de traiter autres programmes plus complexes. Pour poursuivre, une perspective de ce travail serait l'utilisation de notre solution dans divers machines virtuelles Java et de les embarque dans divers équipements :
Jplayer, Robot éducatif….

REFERENCES

[1]. J2ME Building Blocks for Mobile Devices. Sun Microsystems, 2000.

[2]. Java Card Virtual Machine Speci_cation. 2003.

[3]. Richard E. Jones. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, 1996. [p. 35, 39, 40, 48]

[4]. Paul R. Wilson. Uniprocessor garbage collection techniques. In Proceedings of the 1992 International Workshop on Memory Management (IWMM'92), pages 1–42. Springer, 1992. [p. 39, 42]

[5]. John McCarthy. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 3(4):184–195, 1960. [p. 39]

[6]. David F. Bacon, Perry Cheng, and V.T. Rajan. A unified theory of garbage collection. In Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04). ACM Press, 2004. [p. 39]

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

[8]. [8]. Laila MOUSSAID, Mostafa HANOUNE (2011, Mai) "Gestion du TAS: Application sur J2ME". International Journal of Mathematical Archive, Page: 716-719

[9]. [9,10,11]. Laila Moussaid,Mostafa Hanoune(2012,september) "new approach to manage objects in environment java embedded ". International Journal of Mathematical Archive, Page: 3410-3412