Analyse de la démographie des objets dans les systèmes Java temps-réel

Nicolas BERTHIER
Laboratoire VERIMAG
Responsables du stage : Christophe RIPPERT et Guillaume SALAGNAC

le 29 septembre 2006

1 Introduction et problématique

1.1 Contexte

Bien qu'il ait été initialement destiné au développement sur systèmes embarqués, 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. Des versions allégées de l'environnement d'exécution Java standard ont été proposées [Che00, Sun02] pour permettre l'exécution de programmes Java sur ces systèmes, mais elles offrent cependant beaucoup moins de services et compliquent le travail du développeur d'applications (par exemple, l'environnement Java Card ne permet pas l'allocation dynamique de mémoire ou le chargement dynamique de classes).

1.2 JITS et la gestion de la mémoire dynamique

L'architecture JITS (pour *Java In The Small*) [Biz02] a été développée [1] pour permettre le déploiement sur les systèmes très contraints d'applications réalisées en Java avec l'API standard (J2SE). Elle doit donc pouvoir fournir un environnement d'exécution Java complet incluant des outils permettant d'assurer la gestion de la mémoire, et ainsi de prendre en charge l'allocation dynamique des objets et la libération automatique de la mémoire occupée. Cette gestion de la mémoire est actuellement assurée par différents types de ramasse-miettes (par comptage de références, marquage et nettoyage, etc.), mais ces mécanismes ne sont pas adaptés aux contraintes imposées par les systèmes temps-réel. En effet, ils ont un coût non négligeable et très variable, provocant ainsi des temps de pause non déterministes. Ceci représente un inconvénient majeur dans le cas des applications temps-réel, où il est nécessaire de pouvoir prédire le temps d'exécution des instructions.

1.3 La gestion de la mémoire en régions

L'équipe dans laquelle j'ai effectué mon stage propose un autre type de mécanisme [NRSY06] basé sur un regroupement des objets alloués, après analyse statique du programme. Chaque groupe d'objets ainsi créé constitue une région qui, lors de sa destruction, provoque la libération en bloc de la mémoire occupée par son contenu. Cela permettrait de gérer la mémoire sans utiliser de ramasse-miettes, donc sans occasionner de surcoût imprévisible. C'est dans le but de prédire l'efficacité des régions résultantes qu'une étude de la démographie des objets alloués par les applications Java a été réalisée.

[1] JITS est une plate-forme développée au sein du projet POPS à Lille (`http ://www.lifl.fr/POPS`)

1.4 Travail réalisé

Cette étude a été faite empiriquement en instrumentant la machine virtuelle de JITS afin d'obtenir les données disponibles sur les objets alloués, puis en analysant celles-ci afin de mettre en avant des informations pertinentes. Cette méthode a l'avantage d'être relativement simple à mettre en oeuvre puisque la machine virtuelle utilisée permet d'ajouter des méta-données aux instances de chaque objet de l'application exécutée. Des résultats intéressants ont ainsi pu être obtenus, permettant de mettre en avant certaines propriétés des applications analysées.

Après une présentation détaillée de la méthode employée dans cette étude seront résumés et commentés quelques résultats obtenus.

2 Analyse de la démographie des objets

L'analyse de la vie des objets alloués par les applications Java s'est faite en instrumentant la JJVM (pour *Java Java Virtual Machine*), une machine virtuelle implémentée en Java permettant notamment d'expérimenter relativement aisément les outils de la machine virtuelle native. Cette étude s'est décomposée en deux parties distinctes : la première a consisté à recueillir les informations disponibles sur chacun de ces objets ; la seconde étape était destinée à résumer ces données afin de les rendre plus explicites.

2.1 Extraction des informations sur la vie de chaque objet

Les informations concernant le cycle de vie des objets alloués par une application ont été obtenues en instrumentant la JJVM de façon à extraire les données concernant la création et la mort de ces objets. Les instructions allouant de la mémoire ont donc été identifiées, ainsi que celles étant susceptible de provoquer la mort d'un objet (*i.e.* de supprimer ou modifier la dernière référence vers cet objet ; qu'il ne soit plus accessible). Pour chacune de ces dernières, un ramasse-miettes de type marquage et nettoyage instrumenté est exécuté.
Il est alors possible d'identifier, pour chaque objet, le moment de sa création et celui de sa mort ; on peut donc obtenir pour ces deux événements le nombre de bytecodes qui ont été exécutés depuis le lancement de l'application (indication temporelle), ainsi que l'état de la pile d'exécution à ces deux moments. Cet état de la pile d'exécution a été nommé un contexte, et correspond à une description de cette pile, composée de *frames* (chacune correspondant à un appel de méthode).

2.2 Analyse des informations

L'ensemble des informations concernant la naissance et la mort des objets concernés constituant une grande quantité de données, une étape d'analyse statistique a été ajoutée afin de les synthétiser.

Le but de ce travail étant d'identifier des caractéristiques des sites d'allocation d'une application, des regroupements par site de création ont été effectués sur ces données. Ce traitement génère alors des statistiques, pour chaque site de création, sur le cycle de vie des objets qu'il a alloués (*e.g.* durée de vie moyenne, plage de temps pendant laquelle ces objets ont été créés, plage de temps pendant laquelle ces objets sont morts, nombre d'objets créés, etc.).

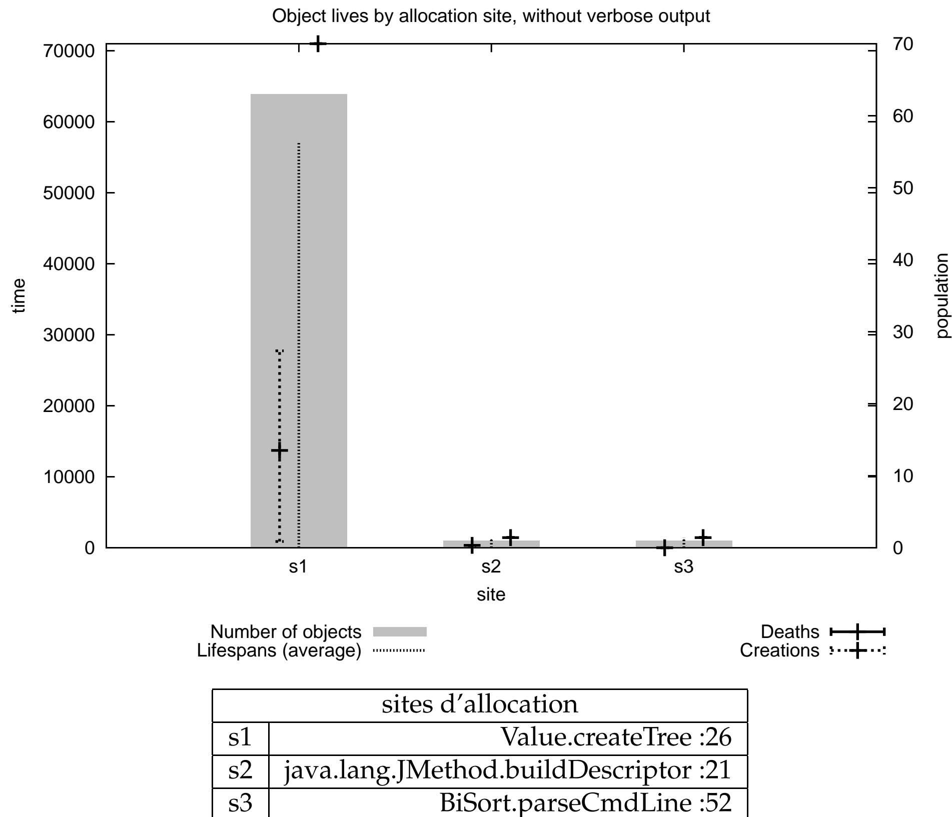
Il est en outre possible d'effectuer les mêmes regroupements par site de mort des objets. Ces informations sont très utile pour comparer la date de mort d'une région créée avec celle des objets qu'elle contient. Cela pourrait aussi permettre de déterminer les objets qui meurent dans la méthode, ou même dans la boucle où ils ont été créés.

3 Résultats expérimentaux

3.1 Analyse du benchmark `BiSort`

Ce processus d'analyse a été effectué sur plusieurs applications, ayant chacune des comportements différents. L'une d'elles va être présentée ici : il s'agit d'un programme faisant partie de la suite de

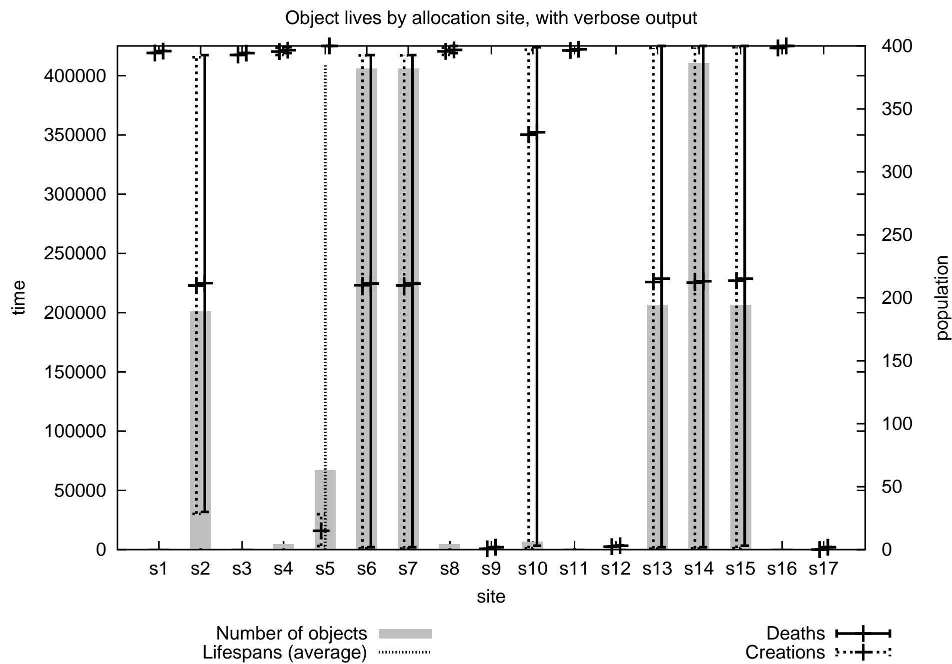
benchmarks JOlden[2], qui ont l'avantage d'inclure des méthodes récursives ou encore d'allouer dynamiquement beaucoup de mémoire, aspects qui constituent des particularités très intéressantes puisque nécessairement supportées par un environnement Java complet. Cette application alloue un arbre composé d'un nombre arbitraire d'objets, puis trie cet arbre. Il est de plus possible de faire afficher diverses informations comme le temps d'exécution des opérations ou encore l'arbre manipulé à différents moments du processus. Les graphes des figures 1 de la présente page et 2 page suivante montrent les résultats obtenus dans les deux cas (La valeur "time" est le nombre de *bytecodes* exécutés depuis la première allocation par l'application).



sites d'allocation	
s1	Value.createTree :26
s2	java.lang.JMethod.buildDescriptor :21
s3	BiSort.parseCmdLine :52

FIG. 1 – Représentation des caractéristiques des différents sites d'allocation du benchmark BiSort, sans affichage

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



sites d'allocation	
s1	BiSort.main :229
s2	Value.inOrder :28
s3	BiSort.main :196
s4	java.lang.Double.toString :12
s5	Value.createTree :26
s6	java.lang.Integer.toString :104
s7	java.lang.Integer.toString :26
s8	java.lang.Double.toString :35
s9	BiSort.main :24
s10	java.lang.StringBuffer.ensureCapacity_unsynchronized :40
s11	BiSort.main :263
s12	java.lang.JMethod.buildDescriptor :21
s13	java.lang.StringBuffer.<init> :31
s14	java.lang.String.<init> :127
s15	java.lang.StringBuffer.toString :11
s16	BiSort.main :297
s17	BiSort.parseCmdLine :52

FIG. 2 – Représentation des caractéristiques des différents sites d'allocation du benchmark BiSort, avec affichage

On remarque qu'apparaissent différents types de sites d'allocation. Par exemple, la méthode `Value.createTree` est appelée récursivement pour construire les noeuds de l'arbre initial. Les objets qu'elle alloue (les noeuds de l'arbre) ont une durée de vie conséquente et sont en nombre élevé par rapport à ceux construits par les autres sites d'allocation. De plus, on peut constater que tous ces objets meurent en même temps, à la fin de l'exécution.
En revanche, l'activation de l'affichage d'informations introduit d'autres types de sites. En effet, on peut constater que les méthodes utilisée pour l'affichage créent généralement beaucoup d'objets à durée de vie très courte (*e.g.* `java.lang.Integer.toString` ou `java.lang.String.<init>` (constructeur)).

3.2 Commentaires

D'après les résultats présentés précédemment, on peut remarquer que le grand nombre d'objets composant l'arbre aurait provoqué un allongement du temps d'une éventuelle exécution d'un ramasse-miettes de type marquage et nettoyage. En revanche la création d'une région (à taille variable, puisque le nombre de nœuds de l'arbre n'est à priori pas connu à l'avance) regroupant l'arbre entier serait certainement très bénéfique, puisque ne comportant que des objets devenant inaccessibles en même temps.

En outre, pour ce qui est des sites de création d'objets à durée de vie très courte, une région pourrait être créée (éventuellement dans la pile d'exécution) de manière à encapsuler un maximum d'objets ayant la même durée de vie. Pour cela, une analyse du site de mort des objets concernés pourrait permettre de déterminer une région optimale.

Enfin, une méthode pour prédire l'efficacité des régions créées après l'analyse statique serait de comparer le cycle de vie des objets (obtenu empiriquement) créés dans une même région, afin de déterminer si la durée de vie de celle-ci diffère de celle de son contenu (*i.e.* si beaucoup d'objets morts vont rester longtemps dans une région encore en vie).

4 Conclusion et perspectives

Le travail qui a été réalisé a permis d'aboutir à une technique d'analyse empirique de la démographie des objets d'une application Java. Elle permet d'obtenir une synthèse du cycle de vie de ces objets, et de déterminer des caractéristiques de chaque site d'allocation d'une application ; ces propriétés peuvent alors être utilisées dans le but de prédire l'efficacité des régions créées après l'analyse statique du programme.

En revanche, l'aspect empirique de cette étude limite son champ d'application. En effet, l'exécution d'un programme temps-réel dépend souvent d'évènements extérieurs, et il devient alors difficile d'obtenir des résultats pertinents par l'intermédiaire d'une simulation sans créer de scénarios typiques d'exécution. Cette analyse peut cependant être une bonne approche pour prédire l'efficacité des régions obtenues après une analyse statique.

Enfin, une étude supplémentaire pourrait permettre de déterminer la possibilité de combiner une gestion de la mémoire en régions, avec un ramasse-miettes plus efficace là où ces dernières poseraient problème (*e.g.* utilisation d'un ramasse-miettes par comptage de références associé au mécanisme des régions).

Références

[Biz02] Gabriel Bizzotto. JITS : Java In The Small. Master's thesis, Université de Lille 1, 2002.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards : Architecture and Programmer's Guide*. Addison Wesley, 2000.

[NRSY06] Chaker Nakhli, Christophe Rippert, Guillaume Salagnac, and Sergio Yovine. Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems. In *Proceedings of "ICOOOLPS 2006"*, 2006.

[Sun02] Sun Microsystems. *The CLDC Hotspot Implementation Virtual Machine*, 2002. `http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf`.