Synthèse de gestionnaires mémoire pour applications Java temps-réel embarquées

Guillaume Salagnac

Laboratoire Verimag / Université Joseph Fourier - Grenoble I

10 avril 2008

Les systèmes embarqués/temps-réel

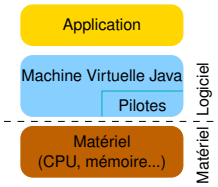
- de plus en plus répandus

- matériel hétérogène
- ressources limitées
- contraintes fortes
  - fiabilité / sécurité
  - temps-réel
- coûts de production élevés

▶ besoin de technologies standard

Java pour l'embarqué/temps-réel

Langage attrayant...
- facilité de programmation
  - riche bibliothèque standard
  - gestion *automatique* de la mémoire
- portabilité
- compacité du code exécutable



... mais peu utilisé dans l'embarqué/temps-réel
- plusieurs «variantes embarquées» incompatibles
- problèmes d'implémentation
  - comportement temporel *imprévisible* du ramasse-miettes (*Garbage Collector*, GC)

Notre approche

Problème :
Temps de pause non prévisibles du ramasse-miettes Java

Proposition :
- garder le langage Java Standard
  - pas de gestion *manuelle* de la mémoire
- changer la machine virtuelle
  - remplacer le ramasse-miettes par un allocateur prévisible
    - *gestionnaire mémoire en régions*
  - calculer les durées de vie des objets à l'avance
    - *analyse statique du programme*

Plan

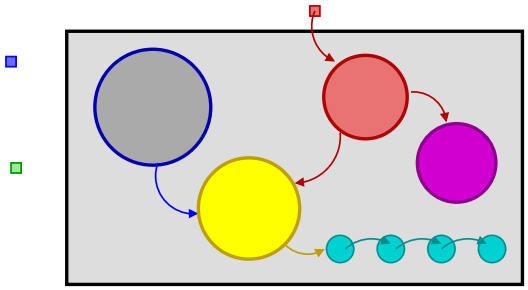
La gestion mémoire

Objectif :
satisfaire les *besoins en espace mémoire* de l'application

Gestion manuelle :
- interface : `allouer()` et `désallouer()`
- ▶ risques d'erreur !

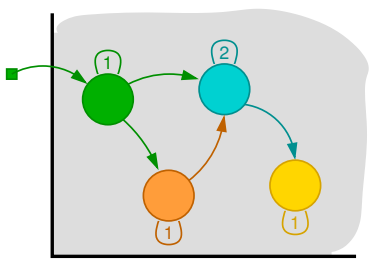
Gestion *automatique* :
- interface : seulement `allouer()`
- ▶ recyclage *transparent* grâce au ramasse-miettes

Ramasse-miettes à marquage-balayage [McCarthy, 1960]



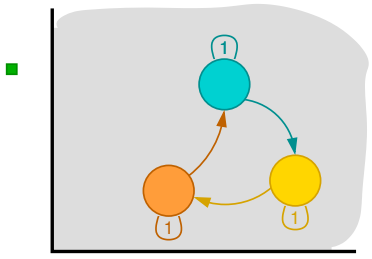
Marquage de proche en proche des objets accessibles

Ramasse-miettes à comptage de références [Collins, 1960]



Le GC compte les *références directes* vers chaque objet
compteur = 0 $\Longrightarrow$ *objet mort*

Ramasse-miettes à comptage de références (2)



... mais *objet mort* $\not\Rightarrow$ compteur = 0 !

À retenir : ramasses-miettes

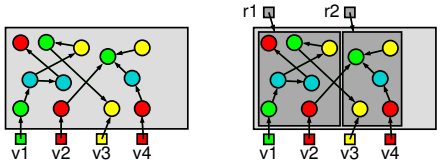
Techniques de *recyclage automatique* de l'espace inutilisé

Marquage-balayage :
- parcours exhaustif de la mémoire
- *temps de pause* intempestifs

Comptage de références :
- naturellement incrémental
- incapable de détecter les *cycles morts* !

▶ incompatibles avec l'embarqué/temps-réel

Gestion mémoire en régions [Tofte, Talpin, 94]



Une région = un ensemble d'objets désalloué *d'un seul bloc*

- objets alloués *côte à côte*
- région détruite *d'un bloc*
- ▶ temps de pause prévisible

mais...
- *difficulté* de programmation
  - quand créer et détruire les régions ?
  - dans quelle région placer les objets ?
- *explosion* des régions

Gestion mémoire en régions : à retenir

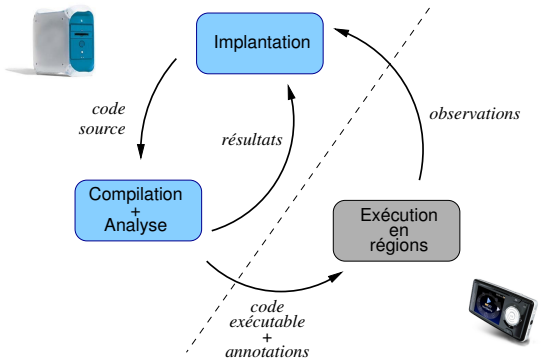
Variantes :
- régions *extensibles* ou de taille fixée ?
- règles d'*organisation* entre régions ?

Utilisation manuelle :
- sous forme de *bibliothèque* : *memory pools* Apache
- au niveau *langage* : `ScopedMemory` RTSJ
  - *Real-Time Specification for Java*

Synthèse automatique de régions :
- Standard ML [Tofte, Talpin, 94], Prolog [Makholm, 2000]
- Java [Cherem, Rugina, 04], [Qin *et al*, 04]

Plan

Intégration dans le cycle de développement



Heuristique

Hypothèse générationnelle : [Hirzel, 2002]
Des objets situés dans la même structure de données (connectés entre eux) ont souvent une durée de vie similaire

Idée :

- une région pour chaque structure de données
  - pas de pointeur entre deux régions

Analyse statique :

- trouver les variables référençant des objets qui *peuvent* être *connectés* (sur-approximation)

Politique d'allocation :

- placer les objets de façon à *grouper* chaque structure *dans une région*

Analyse d'interférence de pointeurs

```
main()
{// list~o1   o2
 ArrayList list =
        new ArrayList ;
 list.<init>(3) ;

 Object o1=new Object ;
 Object o2=new Object ;

 list.add(o1) ;
 o2 = null ;
 ...
}
```

```
class ArrayList
{
 Object[] data ;
 int index ;

 <init>(int capacity)
 {// this~tmp
  this.index = 0 ;
  tmp = new Object[capacity] ;
  this.data = tmp ;
 }

 void add(Object o)
 {// this~o
  this.data[this.index] = o ;
  this.index ++ ;
 }
}
```

résultats :
injectés dans le code

main()
list ∼ *o1* *o2*

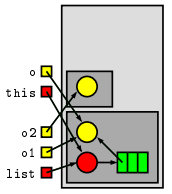
<init>()
this ∼ *tmp*

add()
this ∼ *o*

Politique d'allocation en régions

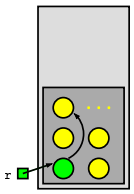
```
main()
{// list~o1   o2
 ArrayList list =
        new ArrayList ;
 list.<init>(3) ;

 Object o1=new Object ;
 Object o2=new Object ;

 list.add(o1) ;
 o2 = null ;
 ...
}
```

```
class ArrayList
{
 Object[] data ;
 int index ;

 <init>(int capacity)
 {// this~tmp
  this.index = 0 ;
  tmp = new Object[capacity] ;
  this.data = tmp ;
 }

 void add(Object o)
 {// this~o
  this.data[this.index] = o ;
  this.index ++ ;
 }
}
```



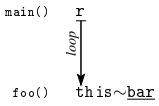
Syndrome d'explosion de région

```
class RefObject
{
  Object f ;

  foo()
  {// this~bar
    Object bar=new Object ;
    this.f=bar;
  }
}

main()
{
  RefObject r=new RefObject() ;
  while(true)
  {
    r.foo() ;
  }
}
```



Analyse de comportement des régions

Principe :
chercher des *motifs à risque* dans le graphe d'appel+interférence :



...et les signaler au programmeur :

```
example.java:7: Possible memory leak
calling context: example.java:17:     r.foo() ;
    Object bar=new Object ;
               ^
```

```
class RefObject
{
  Object f ;

  foo()
  {// this~bar
    Object bar=new Object ;
    this.f=bar ;
  }
}

main()
{
  RefObject r=new RefObject() ;
  while(true)
  {
    r.foo() ;
  }
}
```


À retenir : analyse de pointeurs

Deux nouveaux algorithmes d'analyse de pointeurs

Analyse d'*interférence de pointeurs* :

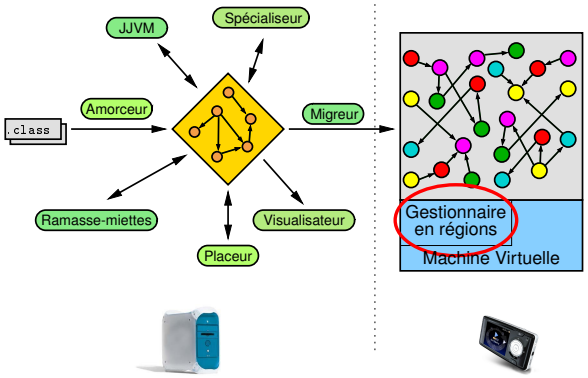
- détecter les connexions potentielles entre objets

Analyse de *comportement des régions* :

- anticiper les risques d'explosion de régions

Plan

JITS : Java In The Small [Grimaud *et al*, Lille]



Protocole expérimental

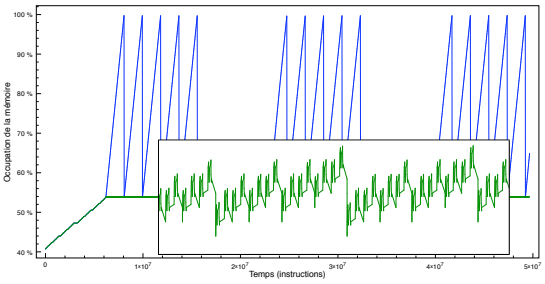
Idée : comparer l'*occupation mémoire* à l'exécution

- avec le ramasse-miettes par défaut (marquage-balayage)
- avec le gestionnaire mémoire en régions

Deux types de programmes :

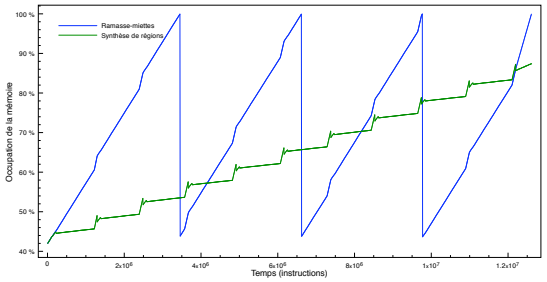
- suite de programmes-témoins *JOlden*
  - algorithmes typiques
  - beaucoup d'allocation
  - différents comportements mémoire
- étude de cas : le décodeur mp3 *JLayer*
  - application réaliste pour un système embarqué/temps-réel
  - code de grande taille

Résultats expérimentaux : JOlden (1)



programme : Bisort

Résultats expérimentaux : JOlden (2)



programme : BH

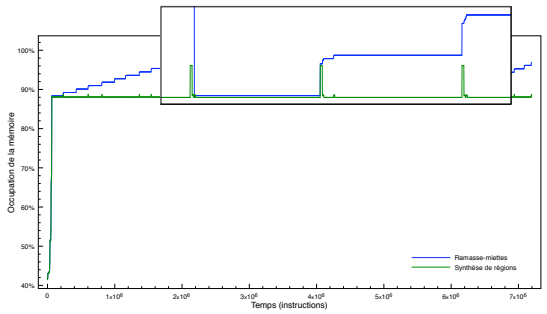
Commentaire : JOlden

Des résultats variés :

- 1/3 des programmes : *mieux* que le GC
- 1/3 des programmes : moins bien que le GC (*explosion*)
- 1/3 des programmes : comportement *similaire*

▶ mais en tous cas, *conformes à l'analyse* de comportement

Résultats expérimentaux : décodeur MP3



programme : JLC

Plan

Bilan

Deux nouveaux algorithmes d'*analyse de pointeurs*

- *synthèse de régions*
  - comportement mémoire OK pour la plupart des programmes
- anticipation des *explosions de région*
  - résultats d'analyse intelligibles pour le programmeur

... assez rapides pour une utilisation interactive

Gestionnaire mémoire en régions

- comportement temporel prévisible (temps constant)
- utilisable dans un contexte embarqué/temps-réel

Combinaison avec un *ramasse-miettes*

- gérer certaines régions par comptage de références

Perspectives

Validation :

- d'autres *expérimentations*
- d'autres *études de cas*

Implantation :

- inclusion dans un *Environnement de Développement Intégré*
- automatiser la *combinaison avec le comptage de références*
- meilleure *intégration dans JITS*

Développements théoriques :

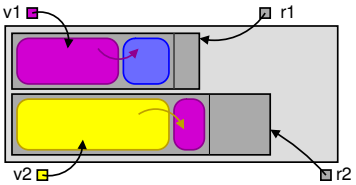
- évaluation *quantitative* des explosions
- analyse de forme globale (présence de *cycles*)
- gestion de la *concurrence*

Merci de votre attention

Des questions ?

Transparents supplémentaires

Gestion mémoire en régions



Modèle classique

- `allouer(`*taille*`)`
  - ▶ renvoie une adresse
- `désallouer(`*adresse*`)`

Modèle en régions

- `créer_région(···)`
- `allouer(`*taille*`,`*region*`)`
- `détruire_région(`*region*`)`

Real-Time Specification for Java (RTSJ)

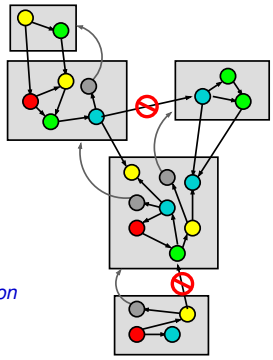
Une extension *temps-réel* de Java

- langage : de nouvelles API
- JVM : de nouvelles garanties

Mémoire gérée en régions

- de taille fixe
- organisées en *arbre*

▶ nombreuses *règles de programmation*



Difficulté de programmation : exemple

```
void runLoop()
{
  while ( true )
  {
    ... lire les entrées
    ... faire quelque chose
  }
}
```

```
class RunLoopIteration implements Runnable
{
  void run()
  {
    ... lire les entrées
    ... faire quelque chose
  }
}

void runLoop()
{
  ... fixer la taille de la région
  memory = new LTMemory( init_sz, max_sz) ;
  runLoop = new RunLoopIteration() ;
  while ( true ) memory.enter( runLoop ) ;
}
```

Patron de conception «*Scoped loop*» [Pizlo *et al*, 2004]

Exemple de fause alerte : le décodeur MP3

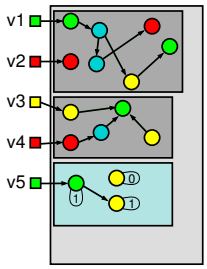
```
main()
{
  player = new Player();

  player.play("file.mp3");
}
```

```
class Player
{
  play(String filename)
  {
    OutputBuffer output = null;

    File file = new File(filename);
    Decoder decoder = new Decoder(file);
    Bitstream stream = new Bitstream(file);

    while( true ) {
       if( output == null ) {
         output = new OutputBuffer();
       }

       Frame frame=stream.readFrame();
       if( frame == null )
          break;

       decoder.decodeFrame(frame,output);
       ...
  }
}
```


Combinaison avec un comptage de références

Idée : gérer certaines régions avec un ramasse-miettes

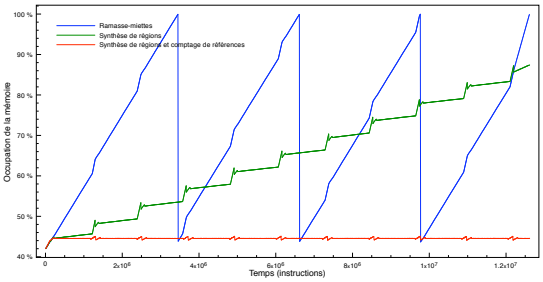
- comportement prévisible du GC
- recyclage de l'espace : pas d'explosion



Comment ?

- en plaçant certains objets dans une région spéciale

Résultats : régions et comptage de références



programme : BH