Un Algorithme de Génération de Patterns Bytecode Java

Mourad KMIMECH* Mohamed Tahar BHIRI** et Nasser BENAMEUR*

(*) Université de Valenciennes et du Hainaut Cambrésis, France. mkmimech@yahoo.fr, nasser.benameur@univ-valenciennes.fr

(**) Département d'Informatique, Faculté des Sciences de Sfax, TUNISIE. tahar_bhiri@yahoo.fr

RÉSUMÉ. Le cadre de ce travail est la compression des programmes interprétés par des machines virtuelles (JVM et JCVM). Nous proposons un Algorithme de Génération de Patterns Bytecode Java (AGPBJ). L'algorithme AGPBJ admet deux paramètres principaux permettant de contrôler la génération des patterns : la fréquence minimum des patterns à garder (f_min) et la longueur maximum des patterns (l_max). Le réglage de ces deux paramètres conditionne la performance de notre algorithme. De plus, pour des raisons d'efficacité, l'algorithme AGPBJ propose et justifie un calcul approximatif (non exact) pour traiter les chevauchements des occurrences des patterns. L'algorithme AGPBJ a été implémenté en Eiffel sous forme d'une bibliothèque de classes : AGPBJOO. Un soin particulier a été accordé à l'exactitude des classes formant la bibliothèque AGPBJOO en utilisant les assertions exécutables préconisées par Eiffel. Enfin, la bibliothèque AGPBJOO a été testée avec succès sur plusieurs Benchmarks.

ABSTRACT. The contexte of this work is the compression of the programs interpreted by virtual machines (JVM and JCVM). We propose an Algorithm of Generation of Patterns Bytecode Java (AGPBJ). The AGPBJ algorithm admits two main parameters permitting to control the generation of the patterns: the minimum frequency of the patterns to keep (f_min) and the maximum length of the patterns (l_max). The regulating of these two parameters conditions the performance of our algorithm. Besides, for reasons of efficiency, the AGPBJ algorithm proposes and justify an approximate calculation (non exact) to treat the overlaps of the occurrences of the patterns. The AGPBJ algorithm was implemented in Eiffel as a library of classes: AGPBJOO. A particular care was granted to the correctness of the classes forming the AGPBJOO library while using the executable assertions recommended by Eiffel. Finally, the AGPOO library has been tested with success on several Benchmarks.

MOTS-CLÉS : Compression, Factorisation, Bytecode, Pattern, Macro-instruction, Machine virtuelle

KEYWORDS: Compression, Factorization, Bytecode, Pattern, Macro instruction, Virtual Machine.

1. Introduction

La taille des programmes (en octets) est devenue une préoccupation avec l'avènement des systèmes embraqués à mémoire réduite et des programmes transférés sur réseau longue distance.

Plusieurs travaux ont été effectués sur la compression de programmes [4] [6] [7]. Ces travaux concernent aussi bien les programmes exécutables (ou machines) que les programmes interprétés.

La compression de programmes exécutables s'adresse à des processeurs RISC, PowerPc, et à l'architecture Thumb (processeurs pour système embraqué) [7].

La compression des programmes interprétés s'adresse à des machines virtuelles telles que SDL, LCC, JVM, Machina et OmmiVM [4].

Notre travail s'inscrit dans le cadre de la compression des programmes interprétés par la machine virtuelle de Java (JVM) : programmes Bytecode Java [3]. Dans cet article, nous proposons un Algorithme de Génération des Patterns Bytecode Java : AGPBJ. Le paragraphe 2 présente un exemple du programme Bytecode Java comportant des patterns. Le paragraphe 3 constitue la partie centrale de cet article. Il propose d'une façon assez détaillée un Algorithme de Génération des Patterns Bytecode Java . Enfin, le paragraphe 4 donne des indications sur l'implémentation orientée objet en Eiffel [8] de l'algorithme AGPBJ.

2. Les patterns dans un programme Bytecode Java

Un programme Bytecode Java comporte potentiellement plusieurs séquences d'instructions qui se répètent plusieurs fois. Ces séquences d'instructions sont appelées patterns. Ces patterns traduisent des opérations élémentaires dans le langage source telles que : accès à un attribut d'un objet, accès à un élément d'un tableau avec le même indice, invocation des méthodes ayant les mêmes paramètres effectifs, invocation des constructeurs venant des classes ascendantes. Pour illustrer notre propos, nous allons prendre un exemple : la classe Java donnée ci-dessous a pour objectif d'implémenter la structure de données Pile en adoptant une représentation physique contiguë (voir figure 1).

La compilation du fichier source Pile.java engendre un fichier binaire Pile.class. Une représentation textuelle du fichier binaire Pile.class est fournie par un désassembleur (javap).

La figure 2 donne le contenu du fichier Pile.bytecode produit par le désassembleur (javap –c). Le tableau TAB 1 récapitule des exemples des patterns identifiés dans ce fichier.

```
public class Pile{private final int taille=100;
  private int[]tab;private int sommet;
  public void creer(){tab=new int[taille];sommet=-1;}
  public boolean vide(){return sommet==-1;}
  public boolean plein(){return sommet+1==taille;}
  public int dernier(){return tab[sommet];}
  public void empiler(int x){sommet++;tab[sommet]=x;}
  public void depiler(){sommet--;}
  public void autotest(){creer();empiler(7);depiler();creer();
     for(int i=1;i<=10;i++)empiler(i);
     for(int i=1;i<=9;i++)depiler();}
  public static void main(String[]args){Pile p=new Pile();
    p.autotest();System.out.println("OK...");}}
```

Figure 1. *Fichier source Pile.java*

```
Method boolean vide()
0 aload_0
1 getfield #2 <Field int sommet>
...
Method boolean plein()
0 aload_0
1 getfield #2 <Field int sommet>
...
Method int dernier()
0 aload_0
1 getfield #3 <Field int tab[]>
4 aload_0
5 getfield #2 <Field int sommet>
...
Method void empiler(int)
0 aload_0
1 dup
2 getfield #2 <Field int sommet>
5 iconst_1
...
10 aload_0
11 getfield #3 <Field int tab[]>
14 aload_0
15 getfield #2 <Field int sommet>
...
```

```
Method void depiler()
0 aload_0
1 dup
2 getfield #2 <Field int sommet>
5 iconst_1
...
Method void autotest()
0 aload_0
1 invokevirtual #5 <Method void creer()>
...
9 new #1 <Class int[]>
10 dup_x1
13 aload_0
14 invokevirtual #5 <Method void creer()>
...
41 aload_0
42 invokevirtual #6 <Method void depiler()>
...
```

Figure 2. *Extrait du fichier Pile.bytecode*

Une méthode connue sous le nom de factorisation du code, basée sur les patterns, permet de compresser des programmes Bytecode Java [2] [7]. Une telle méthode comporte trois modules. Le module <<Génération des patterns>> a pour objectif d'identifier tous les patterns appartenant au fichier class ou CAP. Ensuite, le module <<sélection des patterns optimaux>> permet de sélectionner un jeu de patterns, parmi une base des patterns préalablement construite. Un tel jeu est censé maximiser le taux de compression du fichier class ou CAP.

Patterns	Nombre d'occurrences	Signification : fichier source Pile.java
aload_0 invokevirtual #5 <Method void creer()>	2	creer();
aload_0 invokevirtual #7 <Method void depiler()>	2	depiler();
aload_0 getfield #4 <Field int sommet>	4	accès à l'attribut sommet
aload_0 getfield #3 <Field int tab[]> aload_0 getfield #4 <Field int sommet>	2	accès à tab[sommet]
aload_0 dup getfield #4 <Field int sommet> iconst_1	2	Partie commune de la séquence d'instructions Bytecode permettant de traduire les deux instructions en Java: sommet++ et sommet--

TAB 1. *Exemples des patterns identifiés dans le fichier Pile.bytecode*

Le problème de <<sélection des patterns optimaux>> est un problème NP-complet [5]. Enfin, le module <<compression>> utilise la technique de macros-instructions. Une telle technique consiste à assimiler les patterns sélectionnés lors de l'étape précédente à des nouvelles instructions. Celles-ci font enrichir la machine virtuelle (JVM ou JCVM). Chaque macro-instruction est codée sur un octet. Les macros-instructions sont identifiées par des codes vacants (non utilisés) de la machine virtuelle (53 codes pour la JVM et 69 pour JCVM). Ainsi, le nombre des patterns sélectionnés transformés en macro-instructions est borné par le nombre des codes non utilisés de JVM ou JCVM.

Contrairement aux méthodes classiques de compression de code, la factorisation du code n'exige pas une décompression préalable avant l'exécution.

Dans la suite de cet article, nous allons proposer un algorithme de génération des patterns Bytecode Java.

3. Un Algorithme de Génération des Patterns Bytecode Java

En se basant en partie sur des idées émises dans [2], nous avons établi un Algorithme de Génération des Patterns Bytecode Java (AGPBJ). À notre connaissance, il n'existe pas des algorithmes de génération de patterns Bytecode Java publiés, suffisamment détaillés pour que nous puissions les comparer avec notre algorithme. Pour présenter notre algorithme, nous allons suivre une démarche **descendante**.

3.1. Les variables principales de l'algorithme AGPBJ

On distingue :

- `entree` : mémorise le texte source à factoriser
- `m` : ensemble de groupes de patterns à calculer

Un groupe de patterns comporte plusieurs patterns de même longueur. La longueur d'un pattern correspond à la longueur de sa séquence d'instructions. De plus, un pattern localise les différentes occurrences de sa séquence d'instructions.

- `f_min` : fréquence minimum des patterns à garder
- `l_max` : longueur maximum des patterns

Les trois paramètres `entree`, `f_min` et `l_max` sont des paramètres in (paramètre d'entrée). Le paramètre m est un paramètre out (paramètre résultat). En plus de ces quatre paramètres, l'algorithme AGPBJ a besoin d'une variable provisoire groupe_actuel qui mémorise l'ensemble de patterns actuels (ou courants) de même longueur.

3.2. La procédure principale générer

Elle a forme suivante :

```
début
    démarrer
    itérer
            sortir si fin
            suivant
    finitérer
    patterns_positifs
    couvrir
fin
```

La procédure *générer* permet d'engendrer tous les groupes de patterns dans la variable `m` relatif au texte donné dans la variable *entree*. Cette procédure est traduite par un traitement itératif qui démarre par la génération du groupe de patterns de longueur 1 (voir la procédure *démarrer*) et à chaque itération, elle génère le groupe de patterns de longueur i+1 (voir la procédure *suivant*). La convergence de ce traitement itératif est assurée par la fonction *fin* (voir la fonction *fin*). Après avoir calculé les groupes de patterns dans la structure *m*, la procédure *générer* lance la procédure *patterns_ positifs*. Celle-ci agit sur *m* en supprimant tous les patterns qui n'apportent pas de bénéfice (voir la procédure *patterns_positifs*).

Enfin, la procédure `générer` active la procédure `couvrir` permettant de supprimer de la structure `m` tous les patterns qui sont totalement couverts par d'autres patterns (voir la procédure `couvrir`).

3.3. La procédure démarrer

Elle a forme suivante :

```
début
    itérer
        sortir si entree est parcourue
        occurrence :=position de l'élément courant de la variable entree
        modele :=élément courant de la variable entree
        (j)p:=voir s'il existe dans groupe_actuel un pattern dont sa séquence
        coïncide avec modele
        si p existe
        alors
            ajouter occurrence à p
        sinon
            p1 :=(modele,occurrence)
            ajouter p1 à groupe_actuel
        finsi
        passer à l'élément suivant de la variable entree
    finitérer
    patterns_maintenus
    ajouter groupe_actuel à m
fin
```

La variable `entree` est perçue comme une succession d'éléments. Elle est parcourue élément par élément. Dans le cas de la JVM de Java, un élément est une instruction Bytecode (mnémonique et opérandes) comme iload, aload_0, iadd, getfield, etc. Les résultats calculés par la procédure `démarrer` sont stockés dans `m`. En effet, cette procédure permet de calculer le groupe de patterns de longueur 1.

L'opération (j) qui consiste à voir si un pattern ayant une séquence donnée appartient ou non au `groupe_actuel` de patterns est une opération fondamentale. L'implémentation de cette opération conditionne en grande partie l'efficacité de notre

algorithme. De plus, cette opération peut être paramétrée sur l'équivalence de deux séquences d'instructions : introduire ou exclure des paramètres dans les macro-instructions. À la sortie de la boucle, la procédure `démarrer` lance la procédure `patterns_maintenus`. Celle-ci agit sur `groupe_actuel` en supprimant tous les patterns ayant un nombre occurrences < `f_min` (voir la procédure `patterns_maintenus`).

3.4. La fonction fin

La fonction `fin` a pour rôle d'arrêter le processus de génération de groupes de patterns. Elle est évaluée à vrai si la longueur (commune) des patterns présents dans `groupe_actuel` >= `l_max` ou (ou progressif) tous les patterns appartenant au `groupe_actuel` (groupe de patterns qui vient d'être généré) ont une **seule** occurrence.

3.5. La procédure patterns_maintenus

Elle balaye la structure `groupe_actuel` pattern par pattern. Tout pattern ayant un nombre d'occurrences < `f_min` est supprimé de la structure `groupe_actuel`. Le paramètre `f_min` est initialisé par défaut à 2. Une « bonne » initialisation de ce paramètre pourrait être bénéfique pour l'efficacité de notre algorithme AGIPBJ. En effet, le paramètre `f_min` exerce une influence sur la cardinalité de la structure `groupe_actuel` et par conséquent sur le nombre futur des patterns à générer.

Des renseignements empiriques peuvent être collectés pour bien régler le paramètre `f_min`, en fonction de l'application à compresser.

3.6. La procédure patterns_positifs

La procédure `patterns_positifs` agit sur la structure `m` en gardant uniquement les patterns ayant un bénéfice positif. Les patterns ayant un bénéfice nul ou négatif (perte) sont exclus. Pour y parvenir, la procédure `patterns_positifs` doit parcourir la base des patterns générée `m` pattern par pattern, en appliquant à chaque pattern une formule –donnant une valeur (négative, nulle, ou positive)- permettant de calculer l'apport éventuel de ce pattern.

3.7. La procédure suivant

Elle a forme suivante :

```
début
    groupe_precedent :=groupe_actuel
    itérer
        sortir si tous les patterns de groupe_precedent sont traités
        p:=élément courant de groupe_precedent
        --balayer toutes les occurrences de p
        itérer
            sortir si toutes les occurrences de p sont traitées
            occurrence :=occurrence courante de p
            fin :=la position indiquant la fin de la variable occurrence
            occurrence :=occurrence augmentée de la position (fin+1) si elle
existe
            modele :=modèle de p
            modele :=modele augmenté de l'élément en position (fin +1) s'il existe
            (j)pr :=voir s'il existe dans groupe_actuel un pattern dont sa séquence
            coincide avec modele
            si pr existe
            alors
              ajouter occurrence à pr
            sinon
              pr1 :=(modele,occurrence)
              ajouter pr1 à groupe_actuel
            finsi
            passer à l'élément suivant de p

        finitérer
      passer à l'élément suivant de groupe_precedent
    finitérer
    patterns_maintenus
     chevaucher
    ajouter groupe_actuel à m
fin
```

La procédure `suivant` construit des séquences de longueur i+1 à partir de séquences de longueur i. Elle comporte deux boucles imbriquées. La boucle externe balaye les patterns de longueur i (variable `groupe_precedent`) pattern par pattern. La boucle interne balaye les occurrences du pattern actuel (variable `p`) occurrence par

occurrence. Pour chaque occurrence, on essaye de l'étendre et de l'affecter à un ancien ou un nouveau pattern en fonction du prédicat (j). Avant d'ajouter le contenu de la variable `groupe_actuel` à `m`, la procédure `suivant` lance la procédure `patterns_maintenus` permettant de garder uniquement dans `groupe_actuel` les patterns ayant un nombre d'occurrences >= `f_min`. Ensuite, elle traite les chevauchements éventuels de patterns appartenant à `groupe_actuel` (voir la procédure `chevaucher`).

3.8. La procédure chevaucher

Actuellement, notre algorithme AGPBJ traite les chevauchements concernant les patterns de même longueur. La stratégie du traitement des chevauchements adoptée consiste à balayer **séquentiellement** la liste des occurrences du `groupe_actuel`. À chaque fois où on trouve des occurrences consécutives du `groupe_actuel` (occ1, occ2) chevauchant alors on supprime l'occurrence occ2 de la structure `groupe_actuel`.

3.9. La procédure couvrir

La procédure `couvrir` agit sur la structure `m`. Celle-ci est censée mémoriser des patterns de différentes longueurs et ayant un bénéfice positif. Elle permet de supprimer de la structure `m` tous les patterns qui sont totalement couverts par d'autres patterns.

4. Une implémentation orientée objet de l'algorithme AGPBJ

Nous avons retenu le langage Eiffel [8] pour la réalisation de l'algorithme proposé dans le paragraphe 3. Le choix d'Eiffel peut être justifié, dans notre contexte, par deux raisons essentielles. La première touche aux possibilités fournies par Eiffel liées aux structures de données fondamentales (listes, arbres, piles, tableaux, ...). Ces structures de données sont offertes par Eiffel sous formes des **classes génériques**. Quant à la deuxième raison, elle est liée aux **assertions internes et externes**. En effet, les assertions sont un outil intéressant notamment lors de la mise au point des algorithmes.

L'algorithme AGPBJ a été implémenté en Eiffel sous forme d'une bibliothèque de classes : AGPBJOO. La bibliothèque AGPBJOO comporte plusieurs types de données abstraits (TDAs). Ces derniers sont **déduits** de l'analyse des notions manipulées par l'algorithme AGPBJ.

Nous avons testé avec succès la bibliothèque AGPBJOO sur plusieurs classes Java issues de plusieurs Benchmarks [1].

5. Conclusion

Nous avons proposé un algorithme de génération de patterns appliqué aux programmes interprétés par la JVM de Java et une implémentation orientée objet de cet algorithme. Quant aux perspectives de notre recherche, nous pourrions envisager les prolongements suivants :

– améliorer l'algorithme AGPBJ : deux améliorations sont à étudier : boucle interne de la procédure suivant et la procédure couvrir ;

– traiter les deux autres étapes formant le processus de compression;

– définir des macros-instructions paramétrées pour éviter une multiplication des macros-instructions dans le cas ou toutes les instructions d'une macro-instruction sont contenues dans une autre. Par exemple la macro-instruction ((aaload) (iload 4)) est contenue dans ((aaload) (iload 4) (iaload)) ;

– implémenter la factorisation en modifiant la machine virtuelle Java pour qu'elle puisse traiter d'une façon efficace aussi bien les macros-instructions que les macros-instructions paramétrées.

6. Références bibliographiques

[1] Chen G., Kandemir M., Vijaykrishnan N., Irwin M. J., « PennBench: A Benchmark Suite for Embedded Java », Microsystems Design Lab, The Pennsylvania State University, University Park, PA, 2002.

[2] Clausen L.R., Schultz U.P., Consel C., Muller G., « Java Bytecode compression for Low-End Embedded Systems», In ACM Transactions on Programming Languages and Systems (TOPLAS'00) Volume 22, No.3 (may 2000).

[3] Engel J., « Programming for the Java™ virtual machine », Addison-Wesley 1999.

[4] Ernst J., Fraser C.W., Evans W., Lucco S., Proebstring T.A., « Code compression. In Proc. Conf. on Programming Languages Design and Implementation », pages 358-365, June 1997.

[5] Garey M.R., Johnson D.S, « Computers and intractability », a guide to the theory of NP-Completeness. W.H.Freeman, 1979.

[6] Gupta R., « Code compression in Embedded Systems », March 22, 2002.
www.cs.ucsd.edu/~gupta/cse291-s03/PastProjects/CompressingSurvey.pdf

[7] Latendresse M., « Génération de machines virtuelles pour l'exécution de programmes compressés », Thèse, Faculté des études supérieures, Université de Montréal, décembre 1999.

[8] Meyer B., « Eiffel, le langage », InterEditions, 1994.