Compression par profilage du code Java compilé pour les systèmes embarqués

Karim Ammous

▶ To cite this version:

Karim Ammous. Compression par profilage du code Java compilé pour les systèmes embarqués. Informatique [cs]. Université de Valenciennes et du Hainaut-Cambrésis, 2007. Français. NNT : 2007VALE0023 . tel-03000967

HAL Id: tel-03000967
https://uphf.hal.science/tel-03000967v1

Submitted on 12 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

2007 VALE 00 23

Université de Valenciennes et du Hainaut-Cambrésis

Numéro d'ordre: 07/12

Compression par Profilage du Code Java Compilé pour les Systèmes Embarqués

THÈSE

présentée et soutenue publiquement le 12 juillet 2007

pour l'obtention du

Doctorat de l'université de Valenciennes et du Hainaut-Cambrésis

Spécialité Automatique et Informatique des Systèmes Industriels et Humains
Discipline : Informatique

par

Karim AMMOUS

Président :	Pièrre BOULET, Professeur à l'Université LIFL de Lille
Rapporteurs :	Mo ADDA, Professeur à l'Université de Portsmouth
	Béchir AYEB, Professeur à l'Université de Monastir
Directeurs :	Mourad ABED, Professeur à l'Université de Valenciennes
	Nasser BENAMEUR, Maître de Conférences à l'Université de Valenciennes
Examinateurs :	Smail NIAR, Maître de Conférences Habilité à l'Université de Valenciennes
	Sid-Ahmed Ali TOUATI, Maître de Conférences à l'Université de Versailles
	Mohammed TEGGAR, Docteur Ingénieur du Groupe FININFO à Paris

00
94
ARCHIVES

...omatique et de Mécanique et d'Informatique Industrielles et Humaines — UMR 8530

Université de Valenciennes et du Hainaut-Cambrésis

Numéro d'ordre: 07/12

Compression par Profilage du Code Java Compilé pour les Systèmes Embarqués

THÈSE

présentée et soutenue publiquement le 12 juillet 2007

pour l'obtention du

Doctorat de l'université de Valenciennes et du Hainaut-Cambrésis

Spécialité Automatique et Informatique des Systèmes Industriels et Humains
Discipline : Informatique

par

Karim AMMOUS

Président :	Pièrre BOULET, Professeur à l'Université LIFL de Lille
Rapporteurs :	Mo ADDA, Professeur à l'Université de Portsmouth
	Béchir AYEB, Professeur à l'Université de Monastir
Directeurs :	Mourad ABED, Professeur à l'Université de Valenciennes
	Nasser BENAMEUR, Maître de Conférences à l'Université de Valenciennes
Examinateurs :	Smail NIAR, Maître de Conférences Habilité à l'Université de Valenciennes
	Sid-Ahmed Ali TOUATI, Maître de Conférences à l'Université de Versailles
	Mohammed TEGGAR, Docteur Ingénieur du Groupe FININFO à Paris

Laboratoire d'Automatique et de Mécanique et d'Informatique Industrielles et Humaines — UMR 8530

Remerciements

Le travail présenté dans ce mémoire a été réalisé au Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines (LAMIH) de l'université de Valenciennes et du Hainaut-Cambrésis, au sein de l'équipe Recherche Opérationnelle et Informatique (ROI). Je remercie leurs directeurs respectifs les professeurs Eric Markiewicz et Frédéric Semet de m'avoir accueilli.

Je tiens à adresser mes remerciements à mes directeurs de thèse le professeur Mourad Abed et Nasser Benameur, Maître de Conférences, pour encadrements, soutiens et encouragements tout au long de ce travail.

Je souhaite remercier le président de jury le professeur Pièrre Boulet ainsi que les rapporteurs de ce mémoire de thèse, les professeurs Mo Adda et Béchir Ayeb, qui ont accepté de juger ce travail. Mes remerciements vont également à Messieurs les examinateurs Smail Niar MdC Habilité, Sid-Ahmed Ali Touati MdC et Mohammed Teggar docteur ingénieur ; pour l'intérêt qu'ils ont manifesté.

Je tiens à dire un grand merci à tous mes professeurs, que j'ai connus à l'Université de Sfax, Sleheddine Jarboui, Adel Mahfoudi, Houcine Ezzedine, Mohamed Abid et Mounir Ben Ayed. Ils n'ont cessé de me conseiller et m'encourager.

Je remercie aussi tous mes collègues du groupe ROI, notamment Fabien, Nicolas, Éric, Mickael, Christophe et Frédéric, qui ont participé, par leur bonne humeur, à rendre ce travail agréable.

J'aimerais remercier mes amis frères que j'ai côtoyés à Valenciennes. Je pense à Abdelwaheb, Amin, Salah et les deux Mahdi, qui m'ont aidé de près ou de loin dans la réalisation de ce travail ; mais pas que ça !

Je remercie mes proches, qui le sont restés malgré mon éloignement. Un grand Merci à ma mère Ratiba à qui je suis redevable, ensuite à mon frère Soufien et mon beau frère Mohamed-Hédi d'avoir soutenu ma mère pendant mon séjour loin d'elle. Mes remerciements vont également à la famille Chevalier et la chère Marion, qui n'ont jamais hésité à me prêter main forte.

Je clôture ces remerciements par une pensée à mon père, paix à son âme !

Résumé

Les systèmes embarqués sont caractérisés par des ressources matérielles réduites. Bien que ces ressources ne cessent de s'étendre, elles restent tout de même insuffisantes. L'espace mémoire est l'une des ressources les plus critiques. La compression du code représente une des solutions la plus intéressante pour réduire l'encombrement mémoire.

Notre travail de recherche s'intéresse plus particulièrement à la compression du code Java compilé. Dans ce sens, nous proposons une approche qui couple la compression avec un profiler. L'intérêt de ce couplage est de prendre en compte les spécificités propres au code Java à compresser afin d'établir une stratégie de compaction efficace.

Notre contribution consiste en la conception et la mise en oeuvre d'un système basé sur un profiler pour guider la compression des fichiers class Java. Ce profiler permet d'établir une stratégie de compression efficace offrant le meilleur taux de compression tout en tenant compte des caractéristiques du code en entrée et des dépendances entre les techniques de compression employées. Notre démarche s'appuie sur quatre points :

- l'examen des fichiers en entrée en vue de l'extraction d'informations utiles pour le guidage du processus de compression.
- l'analyse des dépendances des opérations de compression en terme d'interaction mutuelle des unes avec les autres. Pour ce faire, nous avons mis au point deux méthodes, l'une numérique basée sur l'estimation des performances, l'autre analytique permettant de déterminer la nature des dépendances entre les opérations de compression.
- l'évaluation statique des performances permettant le choix de la stratégie de compression. Nous avons, à ce propos, établi les différents paramètres caractérisant chacune des méthodes traitées permettant cette évaluation.
- la construction du chemin de compression le plus efficace dans l'espace de recherche représenté par un graphe orienté et l'application d'heuristiques appropriées pour aboutir à des résultats intéressants.

Mots-clés: Compression, Fichiers Class Java, Profilage, Dépendance, Heuristiques.

Abstract

The embedded systems are characterized by reduced hardware resources. Although these resources are constantly increasing, they remain insufficient. The memory space is one of the most critical resources. The compression of the code designed for embedded systems constitutes an interesting solution to reduce memory footprint.

Our research work specially focuses on the compression of compiled Java code. In our approach, we suggest to couple the compression with a profiler. The advantage of this coupling is that it takes into account the specificities peculiar to the Java class files in order to draw up an efficient compactness strategy.

Our contribution consists in the design and the implementation of a system based on a profiler in order to guide the compression of the Java class files. Our profiler enables us to set up an efficient compression strategy presenting the best rate of compression while taking into consideration the features of the code given in input and the dependencies among the techniques of compression. Our approach is based on four points :

- the study of the input files with a view of extracting the necessary information to the guidance of the compression process.
- the analysis of the dependencies of compression techniques in terms of effects of a given technique on others. To do this, we developed two methods : the first, numerical ; based on the estimation of performance, the second, analytical ; allowing to determine whether or not there are common points among the different compression methods
- the static assessment of the performance permitting to choose a strategy of compression. On this subject, we established the different parameters which distinguish each of the methods we have dealt with allowing this assessment.
- the construction of the most efficient way of compression in the research space symbolised by an oriented graph. The use of appropriate heuristics ends up in interesting results.

Keywords: Compression, Java Class Files, Profiling, Dependency, Heuristics.

Table des matières

Table des figures xi

Liste des tableaux xiii

Introduction générale xv

1 Les systèmes embarqués et le langage Java 1

1.1 Introduction 2

1.2 Les systèmes embarqués 3

1.2.1 Définition 3

1.2.2 Historique et évolution 4

1.2.3 Contraintes et spécificités 5

1.3 Java dans l'embarqué 7

1.3.1 Généralités sur le développement embarqué 7

1.3.2 Choix du language Java 8

1.3.3 Spécifications Java 10

1.3.4 Machine Virtuelle Java 14

1.4 Adéquation entre Java et les systèmes embarqués 18

1.4.1 Java vs systèmes embarqués 19

1.4.2 Besoin en optimisation 19

1.5 Conclusion 21

2 Techniques de compression appliquées au code Java **23**

2.1 Introduction 24

2.2 Compression de code : principe, historique et orientation 25

2.2.1 Principe 25

2.2.2 Historique 26

2.2.3 Orientation 27

2.3 Les différentes classifications 28

2.3.1 Compression générique ou spécifique à l'architecture 28

2.3.2 Exécution avec ou sans décompression préalable 29

2.3.3 Purificateur, compresseur syntaxique et compresseur sémantique . . 30

2.3.4 Proposition d'un nouvelle classification 31

2.4 Techniques élémentaires de compression 32

2.4.1 Techniques matérielles 33

2.4.2 Techniques globales 34

2.4.3 Techniques relatives au bytecode 38

2.4.4 Techniques relatives à la table des constantes 40

2.4.5 Représentations alternatives au format de fichier *class* Java 44

2.5 Conclusion 47

3 Compression par profilage des fichiers *class* Java **49**

3.1 Introduction 50

3.2 Le profilage 51

3.2.1 Rôle et fonctionnement d'un profiler 51

3.2.2 Usage du profiler dans l'optimisation 53

3.3 Profiler dédié à la compression 55

3.3.1 Cadre général de la démarche proposée 55

3.3.2 Architecture du profiler 56

3.4 Principe de fonctionnement du profiler 59
3.4.1 Analyse de dépendances des techniques de compression 59
3.4.2 Scrutation des fichiers *class* Java 63
3.4.3 Module d'évaluation de performance 64
3.4.4 Génération de stratégies de compression 67
3.5 Conclusion 74

4 Mise en œuvre du profiler **77**
4.1 Introduction 78
4.2 Techniques de compression à implémenter dans le profiler 79
4.3 Analyse de dépendances des techniques de compression 80
4.4 Évaluation de performances 85
4.4.1 Rétrécissement des noms (RN) 86
4.4.2 Restructuration des descripteurs de méthodes (RDM) 87
4.4.3 Factorisation du bytecode (FB) 90
4.4.4 Représentation arborescente des noms (RAN) 90
4.4.5 Partitionnement de la table de constantes (PTC) 93
4.4.6 Suppression des informations de débogage (SID) 93
4.4.7 Indices implicites (II) 95
4.5 Implémentation du profiler 95
4.5.1 La librairie d'ingénierie BCEL 96
4.5.2 Conception du profiler 97
4.6 Interface du prototype 98
4.7 Conclusion 100

5 Expérimentations et perspectives **103**
5.1 Introduction 104
5.2 Expérimentations 105

5.2.1 Évaluation des heuristiques 105

5.2.2 Évaluation des stratégies 107

5.3 Perspectives 109

5.3.1 Enrichir l'ensemble des techniques de compression 109

5.3.2 Profilage pour l'optimisation de l'énergie 110

5.3.3 Autres perspectives 111

5.4 Conclusion 112

Conclusion générale **113**

Annexe **117**

Bibliographie **119**

Table des figures

1.1 Exemples de systèmes embarqués large public . . . 3

1.2 Architecture Java [10] . . . 10

1.3 La structure ClassFile . . . 11

1.4 Modules de la Machine Virtuelle Java . . . 15

1.5 Organisation mémoire de la JVM . . . 16

2.1 Évolution du nombre de publications sur la compression de code [58] . . . 26

2.2 Rapport des composants d'un fichier *class* Java . . . 32

2.3 Arbre de Huffman . . . 34

2.4 Application des optimisations à lucarne . . . 37

2.5 Partitionnement de table de constantes . . . 42

2.6 Rétrécissement des noms . . . 43

2.7 Table de constantes partagées du format JAZZ . . . 44

3.1 Profiler standard . . . 52

3.2 Architecture du profiler dédié à la compression . . . 58

3.3 Graphe de dépendances des techniques de l'ensemble T . . . 62

3.4 Le module analyseur de dépendances . . . 62

3.5 Le module scutateur . . . 63

3.6 Le module évaluateur de performance . . . 64

3.7 Exploitation des fiches pour l'évaluation de performance 66

3.8 Le module générateur de stratégies 67

3.9 Graphe de recherche de stratégies 68

3.10 Heuristique gloutonne basée sur le gain normal 71

3.11 Heuristique gloutonne basée sur le gain biaisé 73

4.1 Diagramme de classes 97

4.2 Interface graphique : premier onglet 98

4.3 Interface graphique : deuxième onglet 99

Liste des tableaux

1.1 Correspondance entre étiquette et type d'entrée 12

1.2 Java vs systèmes embarqués 19

3.1 Répartition des types de traitements sur les dépendances 61

3.2 Dépendances des techniques de compression 61

4.1 Répartition en classes des techniques de compression 80

4.2 Dépendances avec la technique de rétrécissement des noms 81

4.3 Dépendances avec la technique de restructuration des descripteurs de méthodes 82

4.4 Dépendances avec la technique de factorisation du bytecode 82

4.5 Dépendances avec la technique de représentation arborescente des noms . . 83

4.6 Dépendances avec la technique de partitionnement de la table de constantes 83

4.7 Dépendances avec la technique de suppression des informations de débogage 84

4.8 Dépendances avec la technique d'indices implicites 84

4.9 Table de dépendances des techniques de compression 85

5.1 Évaluation du gain des heuristiques 105

5.2 Évaluation temporelle des heuristiques 106

5.3 Librairies Java 107

5.4 Évaluation du gain des techniques isolées 107

5.5 Évaluation dépendante du gain 108

Introduction générale

Le domaine de la compression de code a émergé depuis quelques décennies avec les techniques de codage généraux telles que Huffman [30] et Lempel-ziv [64]. Bien que ces techniques permettent de réduire considérablement la taille du code, elles présentent la caractéristique de nécessiter une phase de décodage préalable à l'exploitation. Dans le cas du code machine compressé, l'exécution est pénalisée par la décompression ; ce qui implique une surcharge du processeur et un besoin en espace mémoire supplémentaire pour stocker le code décompressé. Cette caractéristique est désavantageuse lorsque le code est dédié pour les systèmes embarqués étant données les ressources limitées (espace mémoire, énergie, bande passante, fréquence du processeur) dont ils disposent. Malgré les avancées technologiques et l'évolution des systèmes embarqués avec de plus en plus de ressources, le problème de compression de code reste toujours d'actualité notamment avec la migration de certaines applications impliquant beaucoup de ressources (éditeurs de texte ou lecteur de vidéo) des ordinateurs vers les systèmes embarqués. Outre la réduction de l'encombrement mémoire, la compression permet de limiter le temps de transmission de code.

Les algorithmes généraux basés sur la recherche de redondances ne sont pas adaptés à notre problématique, à savoir, la compression du code Java compilé. En effet, les fichiers *class* Java sont caractérisés par une taille réduite (quelques centaines d'octets) et une nature fragmentée (divisés en sections et chaque section contient des données de format différent). Par ailleurs, les nouveaux formats du code tels que CLAZZ [16], JAZZ [11] ou de Pugh [51] représentent des solutions "Java pures" qui tiennent compte de la structure des fichiers *class* Java. Néanmoins, le contenu de ces fichiers est ignoré par ces techniques.

Nous proposons dans ce travail de recherche une solution adaptée à la compression du code Java. L'originalité de notre approche réside dans l'utilisation du profilage qui permet d'obtenir des informations sur le code cible. Ces informations sont utilisées pour guider le choix de la stratégie de compression adéquate.

Afin d'expliciter les recherches effectuées sur ce sujet, ce mémoire est organisé en cinq parties articulées comme suit :

Le chapitre 1 présente deux notions qui forment le support de ce travail de recherche. D'une part, nous décrivons les propriétés intrinsèques des systèmes embarqués de part leur évolution et la gestion des ressources matérielles. D'autre part, nous donnons sur Java une vue globale qui couvre ses caractéristiques, sa Machine Virtuelle Java (JVM), le jeu d'instructions de la JVM et la structure des fichiers Java compilés. Ceci nous permet de montrer le réel besoin en terme d'optimisation que suscite les applications Java dans un contexte embarqué.

Dans le chapitre 2, nous passons en revue les techniques de compression des applications Java dédiées aux systèmes embarqués. Dans un premier temps, nous situons le cadre dans lequel s'inscrit notre travail ainsi que l'état de l'art exposé dans ce chapitre. Dans un deuxième temps, nous définissons une classification des techniques de compression adaptée au code Java compilé afin de mieux présenter l'état de l'art. Selon cette classification, nous décrivons les différentes techniques élémentaires de compression.

Dans le chapitre 3, nous proposons un système qui couple le profilage avec la compression. Après avoir introduit la notion de profiler et son intérêt dans l'optimisation, nous définissons l'architecture de notre profiler en quatre modules : analyseur de dépendance, scrutateur, évaluateur de performance, générateur de stratégies. Pour chaque module, nous décrivons son mode de fonctionnement et ses interactions avec les entrées/sorties ainsi qu'avec les autres modules du profiler.

Le chapitre 4 décrit la mise en œuvre du profiler dédié à la compression. Ce profiler intègre un ensemble de techniques de compression applicables sur les fichiers *class* Java. À chaque étape de la mise en œuvre, nous donnons les détails de conception et d'implémentation relatives au noyau du profiler.

Dans le chapitre 5, nous exposons les résultats expérimentaux obtenus suite à l'application du profiler sur une suite de benchmarks Java. Pour conclure, et en perspective de ce travail, nous présentons les différents apports qui pourraient être intégrés dans le profiler afin d'améliorer sa performance et étendre ses possibilités d'exploitation.

Chapitre 1

Les systèmes embarqués et le langage Java

1.1 Introduction

Ce premier chapitre présente les principales caractéristiques, de notre point de vue, des système embarqué et du langage Java qui constituent le contexte de ce travail de recherche. D'une part, les systèmes embarqués représentent le support sur lequel s'exécutent les applications Java. D'autre part, le code Java sert de champ d'application des techniques de compression. De plus, le domaine de recherche que nous traitons relève d'une technicité accrue. En effet, la compression dans notre cas s'applique sur un code qui est défini selon les spécifications de la Machine Virtuelle Java [28] [45]. Le code compressé doit également respecter ces spécifications afin d'assurer son exécution et de l'obtention de résultats identiques que ceux du code d'origine. Il est donc difficile de compresser le code sans avoir préalablement pris connaissance d'un certain acquis sur sa nature.

Dans la première section de ce chapitre, afin de préciser le champ de notre étude, nous donnons une définition des systèmes embarqués et nous dressons un bref historique traçant leur évolution. À la fin de cette section, nous soulignons les aspects importants pour notre travail, à savoir, certaines contraintes inhérentes aux systèmes embarqués et leurs spécificités.

La deuxième section est consacrée au langage Java. Après un bref rappel de certains langages de programmation parmi les plus utilisés pour les systèmes embarqués, nous exposons les raisons ayant motivé notre choix du langage Java. Ensuite, nous donnons quelques spécifications propres à ce langage. La dernière partie de cette deuxième section est consacrée à la présentation de la JVM avec les modules qui la composent, son principe de fonctionnement et son jeu d'instructions.

Dans la troisième section, nous étudions le langage Java dans un contexte embarqué. Outres les propriétés recensées du langage Java qui favorisent son embarquement, nous avons noté également que ce langage peut rencontrer quelques difficultés à gérer le matériel d'un système enfui. Nous donnons alors les raisons qui sont derrière ces difficultés. Nous terminons cette section par justifier la nécessité d'optimiser les applications Java pour améliorer leur adéquation avec les systèmes embarqués.

1.2 Les systèmes embarqués

La notion de système embarqué prête souvent à confusion. En effet, même en sachant sa définition, il n'est pas garantie de pouvoir l'identifier dans un environnement donné (par exemple : maison ou bureau) puisqu'un SE peut être confondu avec d'autres. Il est donc nécessaire de clarifier autant que possible cette notion en donnant une définition et également identifier un spectre d'exemples et de domaines d'application.

1.2.1 Définition

Établir une définition précise d'un système embarqué (on parle parfois de système enfoui) n'est pas toujours évident et controversée. En effet, cette définition évolue avec le progrès technologique. Ceci fait que la famille des systèmes embarqués englobe un large spectre d'exemples (cf. Figure 1.1) qui peut aller d'une "simple" puce électronique jusqu'au PDA ou le boîtier de télévision interactive (Set-Top-Box).

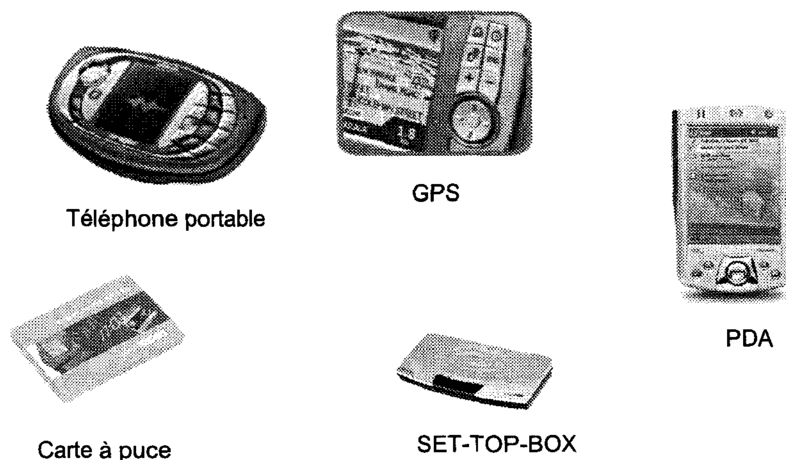

FIG. 1.1 – Exemples de systèmes embarqués large public

Une définition [34] précise identifie un système embarqué comme étant "un système électronique et informatique autonome, qui est dédié à une tâche bien précise. Il ne possède généralement pas des entrées/sorties standards et classiques comme un clavier ou un écran d'ordinateur. Le système matériel et l'application sont intimement liés, le logiciel embarqué étant enfoui, noyé dans le matériel. Le matériel et le logiciel ne sont pas aussi facilement discernables comme dans un environnement de travail classique de type

ordinateur PC".

En simplifiant cette définition, nous pouvons considérer un système embarqué minimal comme une combinaison d'éléments matériels et logiciels sur un même composant réduit.

Au niveau matériel, la configuration de base d'un système embarqué comporte une unité de calcul, une mémoire et un contrôleur des entrées/sorties. Le niveau logiciel est représenté par un programme (un système d'exploitation léger) assurant le contrôle et la gestion des ressources matérielles. Généralement, il est chargé d'effectuer peu de tâches voire qu'une seule tâche. Il réside dans la mémoire ROM (Read Only Memory).

Les deux niveaux (logiciel et matériel) peuvent être étendus respectivement par des périphériques et des applications. Ceci dépend du rôle que le système embarqué est censé accomplir. Ainsi, nous pouvons retrouver un afficheur, un mini clavier, un écran tactile, etc. Pour les mêmes raisons, la partie logicielle peut être enrichie par des applications facilitant le contrôle et l'usage du système.

Un système embarqué n'est pas une simple extension des ordinateurs de bureau. Il peut représenter un système à part entière pouvant fonctionner indépendamment de tout autre ordinateur de bureau ou portable. En effet, un système embarqué a été traditionnellement différencié des ordinateurs de bureau sur la base de la fonctionnalité [56] : les ordinateurs fournissent un large spectre de technologies pour servir les besoins d'une gamme étendue d'applications. Cependant, les systèmes embarqués sont équipés de juste assez de logiciel pour manipuler une application spécifique. Néanmoins, l'émergence d'internet a changé l'idée de *système embarqué à fonction fixe* (fonctionnalité unique) par des composants plus ouverts offrant quelques formes de connectivités.

1.2.2 Historique et évolution

Il existe une dépendance entre l'évolution des processeurs et celle des systèmes embarqués. En effet, il s'agit d'un composant primordial de tout système embarqué. Les premiers systèmes embarqués sont apparus en 1971 avec l'apparition de l'Intel 4004. Il s'agit du premier circuit intégré incorporant tous les éléments d'un ordinateur dans un seul boîtier : unité de calcul, mémoire, contrôle des entrées/sorties. Ce microprocesseur a eu un succès remarquable et son utilisation a augmenté régulièrement durant la décennie qui suit. Les systèmes embarqués ont réellement décollé en 1992 quand Ampro, RTD, et d'autres industriels ont créé le consortium PC/104. Ce groupe a établi un format pour les

microprocesseurs intel basé sur une carte mère d'environ $10cm^2$. Au départ, "le PC/104 a été destiné pour les marchés militaire et médical ; où il devient largement utilisé et respecté"[1]. Le premier système embarqué reconnu fut le système de guidage de la mission lunaire Apollo[2].

Depuis, les systèmes embarqués n'ont pas cessé d'évoluer aussi bien au niveau performance qu'usage. Pour montrer le succès enregistré par les SE, nous rapportons un élément qui concerne le marché du microprocesseur : en 1998, selon les derniers chiffres connus[3], 99% des microprocesseurs ont été destinés au marché des systèmes embarqués. Le succès des systèmes embarqués est tel, qu'il a monopolisé le marché des microprocesseurs par rapport aux ordinateurs de bureau. Cette évolution n'est pas prête de s'arrêter ni même de s'atténuer. En effet, les systèmes embarqués sont omniprésents dans la vie de tous les jours avec de nouvelles fonctionnalités et des coûts de plus en plus abordables permettant ainsi leur démocratisation.

1.2.3 Contraintes et spécificités

Les systèmes embarqués sont soumis à des contraintes matérielles et logicielles. En effet, leurs ressources sont limitées et très restreintes par rapport à celles disponibles sur un ordinateur de bureau. Ce fait constitue une des caractéristiques principales d'un système embarqué (cf. Section 1.2.1). Les ressources les plus critiques d'un système embarqué sont l'espace mémoire, la fréquence horloge du CPU, l'énergie et la bande passante.

1.2.3.1 Espace mémoire

Un système embarqué dispose d'un espace mémoire limité de l'ordre de quelques méga octets au maximum. Les dimensions réduites du système et le coût des espaces de stockage sont les deux paramètres qui font que les constructeurs se contentent du strict minimum en mémoire nécessaire pour le fonctionnement du système.

[1] Selon le site `http://www.ad-mkt-review.com/public_html/tech/esam.html`.

[2] développé par Charles Stark Draper du Massachusetts Institute of Technology (MIT).

[3] selon "embedded systems", 1er juin 2000, page 37.

1.2.3.2 Fréquence horloge du CPU

Les systèmes embarqués sont équipés de processeurs à fréquence d'horloge limitée. L'occupation du CPU par une application ne doit pas être longue ou indéfinie. Autrement dit, les temps d'exécution des tâches doivent être courts. Dans le cas où les délais d'exécution sont limités ou bornés, les systèmes embarqués ont des propriétés temps réel : Les composants doivent répondre aux entrées utilisateur/envionnement dans un délai de temps défini par les spécifications du système.

1.2.3.3 Énergie

La source d'énergie est fournie soit par des batteries ou par des panneaux solaires. Il est donc primordial de bien gérer cette source étant donné l'alimentation énergétique limitée.

1.2.3.4 Bande passante

Les systèmes embarqués disposent souvent d'une source de communication avec l'extérieur comme le wifi pour les PDA ou le réseau GSM pour les téléphones portables. En revanche, la bande passante est limitée au niveau du débit. Son coût est fonction du temps de connexion ou de la taille des fichiers téléchargés.

Outre les quatre contraintes matérielles que nous venons de citer, les systèmes embarqués sont caractérisés par une autre propriété, et non des moindres, à savoir, la sécurité et la sûreté de fonctionnement. En effet, il arrive (voire souvent) que certains de ces systèmes soient embarqués dans des appareils qui peuvent, s'ils subissent une défaillance, mettre des vies humaines en danger. Ils sont alors dits «critiques» et ne doivent jamais faillir.

La gestion de ces ressources est assurée principalement par des composants logiciels du système embarqué. En effet, le système d'exploitation et principalement les applications embarquées utilisent les ressources matérielles pour accomplir une tâche donnée. Une gestion des ressources efficace dépend en grand partie du langage de développement de l'application, dans notre cas, le langage Java que nous présentons dans la section suivante.

1.3 Java dans l'embarqué

À l'origine, le langage assembleur fut adopté comme langage universel de programmation embarquée. Néanmoins, face au développement des systèmes embarqués et leur évolution, tant au niveau matériel que logiciel, il a fallu orienter certains langages de programmation afin de répondre au besoin de créer des applications dédiées. Ce qui a conduit à l'utilisation de langages de programmation de haut niveau. Dans le paragraphe suivant, nous passons en revue les principaux langages de programmation utilisés pour créer des applications embarquées.

1.3.1 Généralités sur le développement embarqué

Les développeurs d'applications embarquées ont souvent recours à des langages de programmation d'applications classiques (pour les ordinateurs de bureau par exemple). Par conséquent, seuls les environnements et certaines "normes" de programmations liées aux ressources matérielles changent. À titre d'exemple, certains systèmes embarqués disposent d'un écran nettement plus petit que celui des ordinateurs de bureau. Il est donc nécessaire d'adapter les entrées/sorties du programme à celles du système embarqué, d'autant plus que celles-ci sont parfois inexistantes. Parmi ces langages de programmation, autre que Java, nous citons :

le langage assembleur : Encore maintenant, de nombreux systèmes embarqués sont écrits en assembleur, pas forcément à cause d'un manque d'expérience ou de savoir-faire, mais le plus souvent pour des raisons économiques. En effet, dans beaucoup d'applications où la complexité logicielle est très faible, l'assembleur permet de réaliser des programmes de taille minimale, donc d'utiliser des composants bon marché ;

le langage C : La plupart des systèmes automobiles par exemple sont écrits en C. Ce langage est devenu incontournable dans les années 1990 quand les systèmes électroniques ont gagné en complexité ;

le langage C++ : Jugé parfois trop complexe pour être totalement maîtrisé par une large communauté de programmeurs et posant des problèmes difficiles de vérification par des outils automatiques, il n'a pas réellement percé pour les logiciels embarqués complexes ;

le langage C# : Le langage C# est une version simplifiée du langage C++. Il accompagne la nouvelle plate-forme .NET (proposée par Microsoft) de développement

d'application. Il est intégré dans des environnements de développement tels que Visual Studio .NET et Borland C# Builder ;

le langage ADA : Il est utilisé dans beaucoup de systèmes embarqués complexes comme les applications militaires, avioniques ou spatiales. Le logiciel du lanceur Ariane par exemple est écrit en Ada.

Un dernier langage constitue le champ de notre travail, il s'agit du langage Java. Le paragraphe suivant lui est consacré.

1.3.2 Choix du language Java

Outre les langages précédemment cités, le langage Java est l'un des plus utilisé dans ce domaine. Plusieurs raisons ont motivé notre choix. Nous les répartissons en deux catégories : générales et particulières. D'un côté, de façon générale, l'orientation objet facilite la conception de composants logiciels avec des interfaces strictement définies, jusqu'au code source. En plus, la richesse de l'API Java offre aux développeurs d'applications destinées aux systèmes embarqués un large choix de composants (par exemple : les boites à outils spécifiques pour la construction d'interfaces). D'un autre côté, d'autres raisons, notamment celles avancées par Cornu [4], sont propres à Java. Ce sont les exigences suivantes qui ont servi de base à la définition de ce langage :

- **simple** : le nombre de constructions du langage est réduit au minimum. La syntaxe Java est proche de celle du C/C++ pour faciliter la migration ;
- **distribué** : vérifié par la pénétration historique de Java dans le World Wide Web, le langage apporte par exemple le RMI (Remote Method Invocation) pour l'invocation de méthodes distantes ;
- **interprété** : le compilateur génère du bytecode, et non pas du code natif. Un interpréteur est nécessaire pour exécuter ce bytecode. Cette architecture facilite le transfert d'un programme sur des plates-formes d'exécution différentes. Une plate-forme d'exécution, que l'on appelle JVM ou Java Virtual Machine (cf. Section 1.3.4), est constituée d'un interpréteur et des bibliothèques de base ;
- **robuste** : c'est un langage fortement typé qui permet de nombreuses vérifications dès la compilation (problèmes de transtypage par exemple, source courante de bugs). La déclaration explicite des méthodes est obligatoire. L'absence de pointeurs évite

[4]Dans son article "Java dans les systèmes embarqués et temps-réel" publié dans "Techniques de l'Ingénieur" en juin 2004, page 4.

les effacements ou corruption de mémoire. Le mécanisme de ramasse-miettes évite les fuites mémoires et les bugs de malveillance dus aux cycles d'allocation et de libération de mémoire. À tout ceci s'ajoutent de nombreuses vérifications effectuées automatiquement à l'exécution, comme la validité d'un index de tableau ;

- **protégé** : l'absence de pointeur empêche le programmeur de contourner les protections mises en place dans le langage. De plus, le compilateur ne décide pas de l'agencement des objets, effectué par la JVM, donc le programmeur ne peut pas deviner prévoir comment les données Java sont stockées en regardant simplement le code source, ce qui rend difficile toute stratégie de contournement par des moyens extérieurs au langage ;
- **neutre du point de vue de l'architecture** : une application Java doit pouvoir être exécutée sur n'importe quel système muni d'une JVM ;
- **portable** : en plus de la caractéristique précédente, la portabilité est facilitée par le retrait du langage de tout aspect lié au type de matériel. Par exemple, chaque type de base est défini explicitement : un entier de type "int" est codé sur 32 bits, quel que soit le microprocesseur. Cela évite les contorsions auxquelles étaient habitués les programmeurs de systèmes embarqués en C jonglant avec des contrôleurs 8 bits, 16 bits ou 32 bits (la taille d'un "int" y est généralement celle du bus) ;
- **performant** : certes, les applications Java sont généralement moins performantes que leur équivalent C car elles sont interprétées. Cependant, les nouvelles architectures d'exécution comprenant un compilateur JIT (Just In Time : une partie du code est compilée sur la cible avant d'être exécutée, pour être réutilisée efficacement) ou AOT (Ahead Of Time : une partie du code est compilée au chargement) le rendent comparable au C ;
- **multithread** (multiples fils d'exécution) : Java fournit un support pour l'exécution de plusieurs threads (classe Thread), ainsi qu'un mot-clef dédié à la synchronisation de ces threads (synchronized) ;
- **dynamique** : les classes sont chargées par la JVM en fonction du besoin, sans nécessiter l'interruption du flot d'exécution.

1.3.3 Spécifications Java

1.3.3.1 Déploiement

Une application Java suit un schéma de transformation pour être exécutée sur le système hôte. En premier temps, il s'agit de créer un fichier source selon la syntaxe et les spécifications du langage de programmation Java [28]. Les développeurs peuvent se servir d'un simple éditeur de textes ou encore d'environments dédiés. Ces outils facilitent le travail de développement grâce à leur convivialité (exemple : affichage en couleur des mots clés) et leurs outils d'assistance (débogage, développement d'interfaces graphiques, etc.). Le fichier source a comme extension .java.

Ensuite, lors de la compilation, le code source Java est traduit en pseudo code ; une sorte de code assembleur portable d'assez haut niveau. Le compilateur *javac* prend en entrée le programme source et génère en sortie un ou plusieurs fichiers *class*. Les tâches effectuées jusqu'à présent se déroulent en local (Machine 1). Elles constituent la première étape de préparation du code à exécuter (cf. Figure 1.2).

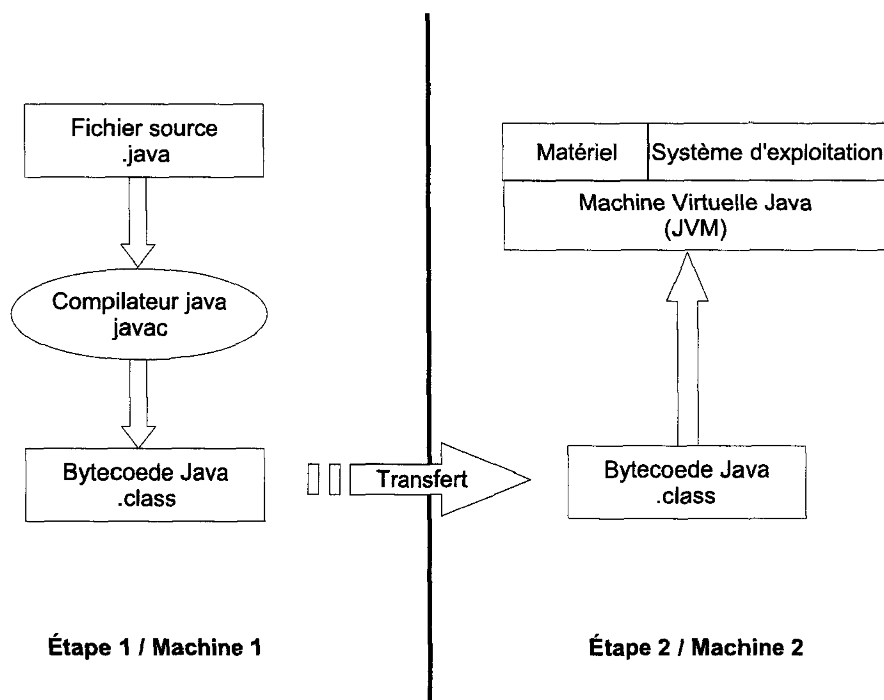

FIG. 1.2 – Architecture Java [10]

La deuxième étape est équivalente à la phase d'exécution. Elle peut se dérouler sur une machine hôte désignée par Machine 2. Néanmoins, les machines 1 et 2 peuvent être

confondues en une seule machine. Dans ce cas, l'exécution se déroule en local. Lors de la dernière phase, l'exécution d'un programme Java est réalisée par la machine virtuelle Java (JVM). Elle se charge d'exécuter le bytecode contenu dans le fichier *class*, par interprétation ou par compilation au vol (Just In Time). Nous développons dans le paragraphe qui suit le format de ce fichier. *class*.

1.3.3.2 Le format du fichier. *class*

Un fichier *class* Java (identifié par l'extension .class) respecte un format défini par les spécifications Java [45]. Chaque fichier contient la définition d'une seule classe ou interface. Les données y sont disposées sous forme de flot d'octets (8-bits). Les données de plus grand taille : 16, 32, et 64 bits sont construites en lisant respectivement deux, quatre et huit octets. Les données représentées sur plusieurs octets sont stockées dans l'ordre big-indian[5]. Le fichier *class* Java est constitué d'une seule structure *ClassFile* illustrée par la Figure 1.3. Elle est définie dans une syntaxe apparentée au langage Java.

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
        }
```

FIG. 1.3 – La structure ClassFile

La Figure 1.3 représente les éléments (ou entrées) de la structure *ClassFile*. Dans l'ordre, nous trouvons :

- magic : un numéro unique pour identifier le format de fichier *class*. Sa valeur est 0xCAFEBABE.

[5]Désigne une manière dont un ordinateur numérote les octets en commençant par les octets de poids fort.

- minor_version et major_version : représentent respectivement la valeur mineure et majeure de la version du format de fichier *class*. La version de ce dernier doit être comprise entre *minor_version* et *major_version*.
- constant_pool_count : correspond au nombre d'entrées de la table des symbôles *constant_pool*.
- constant_pool[] : table de structures représentant des symbôles de type chaînes de caractères, des noms de classes ou d'interfaces et d'autres symbôles qui sont des références vers la table des symbôles. Les structures ont le format suivant :

```
cp_info {  u1 tag;
           u1 info[];
        }
```

  Tag est une étiquette de taille *u1* (un octet) qui donne le type de l'entrée *cp_info*. Le contenu de la table *info[]* varie selon le type de l'étiquette. La liste de toutes les étiquettes possibles est donnée par le Tableau 1.1.

Type d'entrée	Valeur
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

TAB. 1.1 – Correspondance entre étiquette et type d'entrée

- access_flags : sa valeur représente les droits d'accès et les propriétés de la classe ou l'interface en question.
- this_class : sa valeur est l'index vers une entrée de la table des symbôles représentant la classe ou l'interface définie par le fichier *class*.
- super_class : indique sa classe mère. Sa valeur est l'index d'une entrée de la table des symbôles représentant sa classe mère directe, si cette dernière existe. Le cas échéant, elle prend la valeur zéro.

- interfaces_count : donne le nombre d'interfaces mères de la classe en question.
- interfaces[] : tableau d'indexes vers des entrées de la table des constantes. Chaque entrée est une structure de type *CONSTANT_CLASS_info* représentant une interface.

```
CONSTANT_Class_info { u1 tag ;
                      u2 name_index ;
                    }
```

- fields_count : donne le nombre de structures *field_info* (cf. annexe) dans la table des méthodes *fields*.
- fields[] : table de structures *field_info* qui donnent une description d'un champs de la classe ou de l'interface en question.
- methods_count : donne le nombre de structures *method_info* (cf. annexe) dans la table des méthodes *methods*
- methods[] : table de structures *method_info* qui représentent la description complète d'une méthode de la classe ou de l'interface en question.
- attributes_count : donne le nombre d'éléments de la table des attributs *attributes*.
- attributes[] : table de structures de type *attribute* (cf. annexe).

Nous nous contentons de ce niveau d'approfondissement dans l'exploration de la structure du fichier *class*. D'autres détails figurent en annexe de ce mémoire. Pour une idée complète et exhaustive sur cette structure, nous renvoyons le lecteur vers le livre référence dans ce domaine [45].

Nous notons que la table des symbôles occupe environ 60% de la taille totale du fichier *class*. Ceci s'explique par le fait que les autres composants de la structure *ClassFile* font référence à des entrées de la table des symbôles. Le mécanisme d'indexage et d'indirection y contribuent également. En effet, il faut souvent passer par des indexes vers des structures intermédiaires avant de trouver la représentation finale.

Un élément primordial dans la définition du concept Java est la Machine Virtuelle Java. Cette dernière est un facteur aussi important derrière le succès du langage. Nous la présentons dans la section suivante.

1.3.4 Machine Virtuelle Java

1.3.4.1 Définition

La Machine Virtuelle Java (ou JVM selon l'abréviation anglophone) est un programme spécifique à chaque plate-forme ou couple (machine/système d'exploitation). Elle permet aux applications Java compilées en bytecode de produire les mêmes résultats quelle que soit la plate-forme, tant que celle-ci est pourvue de la Machine Virtuelle Java adéquate. La JVM est composée des trois principaux modules suivants :

1. **Chargeur de classes :** les programmes Java sont structurés en classes. Une classe Java engendre un fichier compilé. Ces fichiers sont chargés dynamiquement. À la première création d'une instance d'une classe, le chargeur de classes recherche le fichier localement ou depuis le réseau, charge les classes héritées, réalise l'édition des liens et vérifie le contenu sémantique et syntaxique des fichiers chargés. L'exécution des blocs statiques des classes chargées est effectuée.
2. **Interpréteur des instructions :** l'interpréteur exécute l'instruction référencée par le registre *PC* (pointe sur l'instruction suivante à exécuter par la CPU.) Une instruction de la machine est codée sur un octet, spécifiant le code opération, suivie de zéro ou plusieurs octets de données.
3. **Ramasse-miettes** : le ramasse-miettes libère les zones qui ne sont plus référencées par les variables du programme. Ces zones compactées deviennent libres pour une nouvelle allocation. Le déclenchement du ramasse-miettes intervient lorsque la mémoire des objets est pleine ou de manière incrémentale, selon l'algorithme du récupérateur. Un récupérateur séquentiel interrompt le programme utilisateur de la durée nécessaire au compactage. Un récupérateur concurrent agit en même temps que l'exécution du programme utilisateur. Le déclenchement intervient en temps partagé ou en parallèle sur un coprocesseur chargé de la mémoire des objets. Il existe un nombre important d'algorithmes de récupération mémoire. Pour en savoir plus, le lecteur se reportera au livre de Jones et Lins [33].

Les trois modules cités ci-dessus sont en interaction avec la mémoire (cf. Figure 1.4). Dans le cas des systèmes embarqués, il s'agit de la ROM (Read Only Memory). Les données sont organisées par la JVM selon un schéma particulier. La Figure 1.5 donne un aperçu de cette organisation :

Ainsi, nous trouvons :

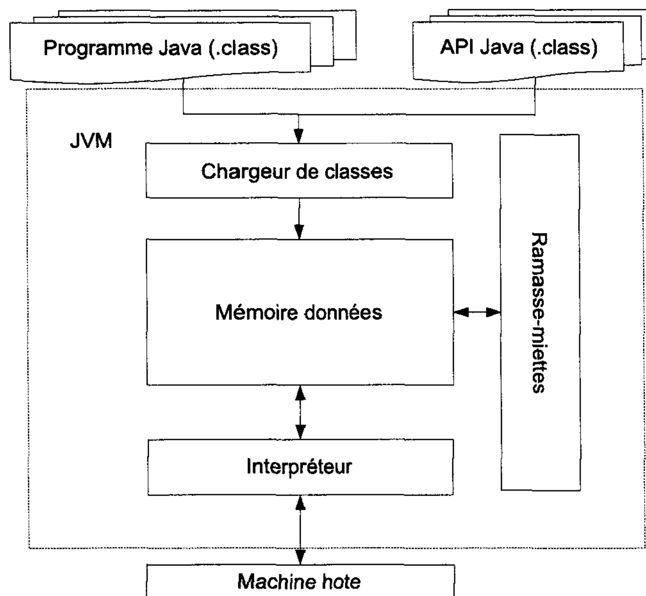

FIG. 1.4 – Modules de la Machine Virtuelle Java

- **L'espace des méthodes** où se trouve le bytecode proprement dit. Il contient aussi les tables de symbôles. La zone des constantes, les tables associées aux méthodes, les définitions des classes sont rangées dans un tas non géré par le ramasse-miettes ;
- **La pile** où sont rangées les frames d'invocation des méthodes. Elle est gérée par les trois pointeurs *frame*, *vars* et *optop* qui pointent respectivement sur le segment d'activation de la méthode en cours, la première variable locale de la méthode en cours et le sommet de la pile d'évaluation. Il existe une pile par thread ;
- **Les registres** sont le *PC*, le *SP* (Stack Pointer : il point sur le sommet de pile) et trois pointeurs pour gérer la pile. Il existe un jeu de registres par thread ;
- **Le tas** (heap) où sont rangés les objets (instances des classes et les tableaux). Il est parcouru régulièrement par le ramasse-miettes. Le *tas* est unique.

1.3.4.2 Fonctionnement de la JVM

Après que le fichier *class* soit transféré sur la machine hôte (ou en local), la JVM le prend en charge. Ainsi, elle lui applique trois processus pour qu'il soit prêt pour l'exécution. Son cycle de vie, qui précède l'exécution, se compose de trois étapes : chargement, édition des liens et initialisation.

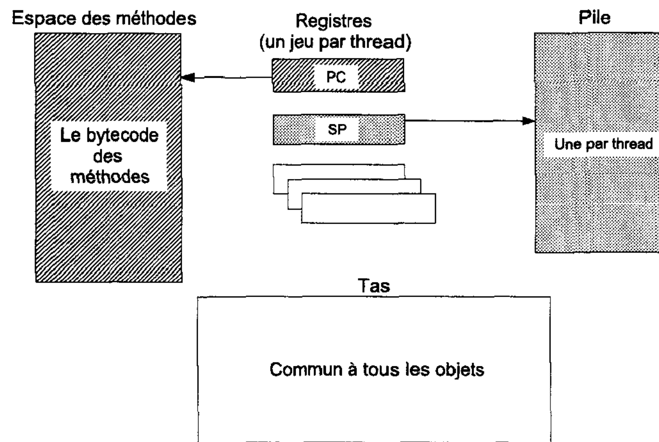

FIG. 1.5 – Organisation mémoire de la JVM

1. Chargement des classes : Le processus de chargement consiste à rechercher le fichier compilé et à créer une représentation interne de cette classe. Les classes héritées sont également chargées, les autres classes mentionnées dans ce fichier sont chargées au moment de leur utilisation, lors de la première création d'une instance par exemple. La recherche du fichier est soit locale soit distante. Le fichier peut être extrait d'une archive (zip, jar, cab ou autres). Le chargeur peut demander la compilation d'un code source Java afin de produire le fichier *class*. Le chargement d'une librairie en code natif est à la charge du programmeur.
2. Édition des liens : L'édition de liens permet d'intégrer la classe chargée dans le contexte courant de la machine. Cette étape est constituée de trois phases :

   Vérification : pour s'assurer que le fichier est bien formé ;

   Préparation : consiste en l'allocation des variables de classes ;

   Résolution : convertit les références symboliques en références internes de la machine.
3. Initialisation : Elle consiste en l'exécution de la méthode d'initialisation *<clinit>* d'une classe ou interface. Cette méthode regroupe les instructions du bloc statique de la classe. Elle n'est exécutée qu'une seule fois au chargement de la classe ; elle intègre les initialisations des variables de classe.

1.3.4.3 Jeu d'instructions de la JVM

Une instruction de la Machine Virtuelle Java consiste en un opcode spécifiant l'opération à exécuter, suivie de zéro ou plusieurs opérandes renfermant des valeurs sur lesquelles elle opère. Une instruction est codé sur un octet ; ce qui implique l'existence de 256 instructions. Cependant, la JVM n'utilise actuellement que 200 instructions, plus 25 instructions appelées "quick" que la JVM peut utiliser en interne. Le reste des instructions sont non réservées. De nouvelles instructions peuvent alors être ajoutées ultérieurement. Pour avoir la liste complète des instructions et leurs définitions, le lecteur peut se référer au livre de Lindholm et Yellin [45]. Nous trouvons ainsi différentes catégories d'instructions :

- arithmétiques et logiques : *iadd, imul*
- manipulation de la pile : *dup, pop* ;
- gestion des variables locales et des constantes : *aload, iload, istore, iconst* ;
- gestion des conversions entre types primitifs : *i2l* ;
- contrôle, branchement : *if_icmplt, goto, jsr, return* ;
- accès aux variables de classe : *getstatic, putstatic* ;
- appels de méthodes de classe : *invokestatic* ;
- gestion des exceptions : *athrow* ;
- allocation de l'espace mémoire : *new, newarray* ;
- accès aux champs d'instance : *getfield, putfield* ;
- accès aux éléments de tableau : *aaload, aastore* ;
- appels de méthodes d'instance : *invoke_virtual, invoke_super, invokeinterface* ;
- gestion des conversions entre classes : *check_cast, instance_of* ;
- gestion de la concurrence : *monitor_enter, monitor_exit.*

Pour obtenir le bytecode relatif à un fichier *class*, il est possible d'utiliser, entre autres, l'outil *javap* fourni par Sun. Nous donnons un aperçu du bytecode d'un programme moyennant l'exemple typique *HelloWorld.* Soit le code source Java suivant contenu dans le fichier HelloWorld.java :

```
public class HelloWorld
{
    public static void main(String[] args)
    {   System.out.println("Hello World!");   }
}
```

Le lancement de l'outil *javap* avec l'option de désassemblage donne le code suivant :

```
Compiled from "HelloWorld.java"
public class HelloWorld extends java.lang.Object{

public HelloWorld();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/
                            Object."<init>":()V
   4:   return
public static void main(java.lang.String[]);
  Code:
   0:   getstatic       #2; //Field java/lang/System.out:
                            Ljava/io/PrintStream;
   3:   ldc             #3; //String Hello World!
   5:   invokevirtual   #4; //Method java/io/PrintStream.
                            println:(Ljava/lang/String;)V
   8:   return
}
```

Le bytecode généré correspond aux méthodes *HelloWorld* (équivalente à un constructeur par défaut ajouté par le compilateur) et *main* (méthode statique de lancement d'un programme). Remarquons que lorsqu'une instruction (opcode) possède un paramètre (opérande), *javap* le cherche dans la table des symbôles à l'entrée indiquée et le place sur la même ligne que l'instruction en question.

1.4 Adéquation entre Java et les systèmes embarqués

Dans cette dernière section de ce chapitre, nous comparons les caractéristiques relatives aux systèmes embarqués et au langage Java. L'intérêt de cette comparaison est de révéler le degré d'adéquation entre les applications Java et les supports embarqués.

1.4.1 Java vs systèmes embarqués

Nous avons présenté précédemment (cf. Section 1.3.2) les différentes caractéristiques qui ont motivés le choix du language Java pour des applications destinées aux systèmes embarqués. Néanmoins, ce langage a d'autres propriétés moins avantageuses qui sont encore accentuées lorsqu'elles se rajoutent (cf. Tableau 1.2) aux contraintes des systèmes embarqués évoquées dans la section 1.2.3.

Programme Java	**Système embarqué**
Encombrant	Espace mémoire limité
Lenteur d'exécution	Vitesse lente du processeur

TAB. 1.2 – Java vs systèmes embarqués

La Machine Virtuelle Java est en grande partie responsable de ce surcoût en termes de ressources matérielles nécessaire à l'exécution. En effet, un programme Java n'est pas exécuté directement par le matériel du système. Il existe plutôt un programme intermédiaire, à savoir, la Machine Virtuelle Java qui se charge de l'interprétation de son code compilé. Bien qu'elle apporte à Java une architecture flexible, sécurisée et portable, elle consomme des ressources matérielles en termes de temps processeur et d'espace mémoire lorsqu'elle gère l'exécution d'un programme Java. Cette influence [4] diffère selon le mode opératoire de la JVM, notamment celui de son ramasseur de miettes.

1.4.2 Besoin en optimisation

Malgré l'évolution permanente de la technologie embarquée, avec toujours plus de ressources, le besoin en terme de performance n'est pas toujours satisfait. Ce besoin persistant s'explique par l'évolution parallèle des applications qui s'exécutent sur les systèmes embarqués. Les applications sont de plus en plus gourmandes en terme de ressources matérielles. En plus, on assiste à la migration d'applications complexes, destinées au départ pour les ordinateurs de bureau, vers les systèmes embarqués.

La solution triviale pour disposer d'avantage de ressources et de mieux équiper les systèmes embarqués consiste à ajouter des extensions mémoire, des processeurs plus performants, des batteries plus économiques, etc. Néanmoins, le coût parfois exorbitant de ces ressources est souvent dissuasif. D'un autre côté, il n'est parfois pas possible d'ajouter des extensions au matériel comme dans le cas des cartes Java. Enfin, c'est le propre d'un

SE de disposer de resources matérielles réduites.

Une solution "acceptable" doit donc faire avec les ressources disponibles sans envisager un surcoût pénalisant. Une solution logicielle est toutefois envisageable et constitue une bonne alternative pour bien gérer ces ressources.

1.5 Conclusion

Dans ce chapitre nous avons présenté les deux principales bases représentant le contexte de notre travail de recherche : les systèmes embarqués et le langage Java.

En effet, nous avons consacré la première section à faire le tour des systèmes embarqués. Ainsi, nous avons retenu une définition généraliste d'un système embarqué minimal qui l'identifie comme étant une combinaison d'éléments matériels et logiciels, le tout sur un composant réduit. Historiquement, l'évolution des systèmes embarqués, depuis leur apparition, a montré que cette définition reste toujours valable. D'ailleurs, le caractère réduit est toujours présent et se traduit par un manque de ressources matérielles : un espace mémoire restreint, un processeur lent, une source d'énergie faible, etc.

Dans la deuxième section de ce chapitre, nous avons présenté les spécifications du langage Java. Parmi tous les langages de programmation, c'est le langage Java qui est le plus intéressant pour plusieurs raisons : simplicité, performance, robustesse et mobilité. Nous avons également présenté la Machine Virtuelle Java avec ses modules, son principe de fonctionnement et son jeu d'instructions.

Malgré tous les avantages qu'offre le langage Java, son exécution dans un contexte embarqué suscite quelques interrogations quant à son adéquation à cet environnement particulier. Ceci s'explique par le fait que Java nécessite des ressources matérielles relativement conséquentes, qui font défaut aux systèmes embarqués dans de grandes proportions. Dans ce sens, dans la deuxième section de ce chapitre, nous avons exprimé et justifié le besoin en terme d'optimisation du code.

Beaucoup de travaux ont fait de la réduction de l'encombrement mémoire du code Java leur champ d'étude. Des travaux aboutis ont vu le jour et ont contribué à apporter des solutions afin d'arriver à un code moins gourmand en terme de mémoire occupée. Le deuxième chapitre présente les techniques de compression pour les applications Java.

Chapitre 2

Techniques de compression appliquées au code Java

2.1 Introduction

Dans ce deuxième chapitre nous passons en revue les techniques de compression des applications Java dédiées aux systèmes embarqués. Trois sections composent ce chapitre.

La première section a pour objectif de situer, d'une part, le cadre général de notre travail de recherche et, d'autre part, ses frontières. Ainsi, nous présentons la compression du code comme un axe de recherche de l'optimisation du code. Ce même axe est à son tour divisé en plusieurs directions.

Un grand intérêt est manifesté par les chercheurs pour la compression du code. Il se traduit par un nombre important de travaux dans ce domaine. Pour mieux présenter les techniques de compression des programmes Java, nous avons jugé plus judicieux de les classer. Il existe différentes classifications des techniques de compression. Elles sont fondées sur des critères tels que le format résultant de la compression, sa nature ou encore son principe de fonctionnement. Après l'analyse de ces classifications, nous avons montré qu'elles manquent soit de structuration soit d'adéquation aux techniques de compression des fichiers *class* Java, nous avons opté pour une nouvelle classification. Ceci fait l'object de la deuxième section.

Dans la troisième section de ce chapitre nous décrivons, sans être exhaustif, les techniques de compression des applications Java. Dans un souci de clarté, la description des techniques se présente d'une façon granulaire. En effet, nous avons isolé les techniques de leur contexte général. Chaque technique élémentaire est donc présentée et est illustrée, dans certains cas, d'exemple. Vers la fin de cette section, nous donnons quelques représentations alternatives au format de fichiers *class* Java. Ces nouveaux formats résultent de combinaisons de quelques techniques élémentaires.

2.2 Compression de code : principe, historique et orientation

La compression s'inscrit dans le cadre global de l'optimisation. Nous consacrons cette section à cerner ce cadre et à définir l'orientation de notre travail de recherche.

2.2.1 Principe

L'optimisation consiste à modifier le code visant à rendre maximale sa vitesse d'exécution, ou minimale la taille mémoire nécessaire à cette exécution, ou bien rendre leur rapport optimal. Dans le cas des systèmes embarqués, trois principales ressources (énergie, CPU et mémoire) sont au centre d'intérêt de la recherche dans le domaine de l'optimisation des applications embarquées. Ainsi, on distingue un axe de recherche en optimisation pour chacune de ces ressources. Ces axes se résument en la réduction de l'espace mémoire (occupé par le code), du temps d'exécution et de la consommation d'énergie.

Avec l'avènement des nouveaux systèmes embarqués et leur ouverture à l'extérieur (connectivité à Internet), un quatrième axe a fait apparition ; il s'agit de l'optimisation de l'utilisation de la bande passante. Cependant, cet axe est étroitement lié au premier (réduction de l'espace mémoire) avec une relation de cause à effet. En effet, un code plus compact nécessite, par conséquent, moins de bande passante pour être transféré via le réseau. L'optimisation de code est perçu comme un corollaire de la compression de code.

Donald Ervin Knuth met en garde contre les risques que peut engendrer l'optimisation. Selon lui, "l'optimisation prématurée est la source de tous les maux." [38]. En effet, l'optimisation peut avoir un effet néfaste sur la clarté du code et le bon fonctionnement du programme. Il faut donc y recourir seulement quand elle s'avère nécessaire. Dans le cas des applications Java dédiées aux systèmes embarqués, le besoin en optimisation a été montré au chapitre 1 (cf. section 1.4.2).

Le but de l'optimisation étant de décharger le système embarqué afin qu'il dispose de plus de ressources possibles, il est donc naturel que les techniques d'optimisation se déroulent généralement en dehors du système embarqué (hors-ligne) ; alors que son résultat est perceptible sur le système hôte, une fois le code optimisé y est embarqué. Notons que des machines performantes sont utilisées pour effectuer l'optimisation. Ceci provient du fait que l'optimisation nécessite parfois un calcul lourd.

2.2.2 Historique

La compression du code est un domaine qui date depuis des décennies. Les travaux de Huffman [30] parus en 1952 ont marqué le début des recherches dans ce domaine. Depuis, plusieurs autres chercheurs s'y sont intéressés ; ce qui explique le nombre considérable de travaux disponibles. Van de Wiel a recensé les publications les plus intéressantes qui concernent la compression du code [58] depuis les années cinquante jusqu'à l'année 2002. Le diagramme illustré par la figure 2.1 montre l'évolution du nombre de références (publications) qu'il a recensées en fonction de l'année d'apparition. Bien que ce recensement ne soit pas exhaustif, il nous permet néanmoins de suivre l'évolution de la recherche dans ce domaine.

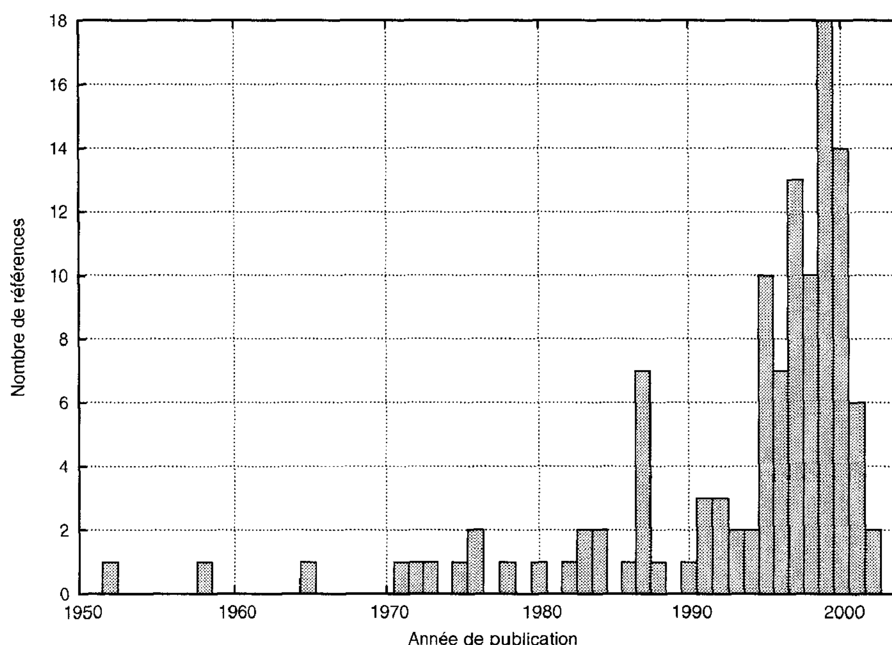

FIG. 2.1 – Évolution du nombre de publications sur la compression de code [58]

Nous pouvons distinguer trois périodes sur l'histogramme illustré par la figure 2.1. De 1950 jusqu'à 1970, seules trois études ont été publiées. Un nombre relativement faible qui traduit un besoin loin d'être conséquent. La seconde période, qui s'étale de 1970 à 1990, est marquée par un accroissement du nombre de publications (en moyenne une par an). Ceci traduit l'apparition des premiers systèmes embarqués programmables à cet époque (cf. chapitre 1 section 1.2.2). Ainsi est né le besoin d'y embarquer du code, de préférence sous un format compact vu le faible espace de mémoire disponible. De 1990 jusqu'à 2002, le nombre de travaux sur la compression a considérablement augmenté. Les applications

Java embarquées sont en grande partie derrière cet accroissement.

2.2.3 Orientation

Parmi les trois axes d'optimisation cités précédemment (cf. section 2.2.1), nous nous intéressons plus particulièrement à celui de la compression du code. L'importance de la réduction de la taille du code sur les systèmes embarqués est majeure. D'ailleurs, il en témoigne le nombre phénoménal de travaux qui s'y sont intéressés [58].

Quant aux deux autres axes de recherche (optimisation du temps d'exécution et de l'énergie consommée), même s'ils ne constituent pas pour le moment notre principal centre d'intérêt, ils représentent néanmoins des extensions complémentaires à explorer en perspectives. Toutefois, il existe des travaux de recherche, portant sur la réduction de l'encombrement mémoire, qui abordent l'énergie consommée [14][29][5]. Il ne s'agit pas dans ces études d'un but en tant que tel, mais plutôt d'une estimation de l'effet que peut avoir le code compressé sur la consommation de l'énergie (augmentation, stabilité ou réduction).

En effet, la consommation de l'énergie est étroitement liée à la compression du code sur plusieurs niveaux. À titre d'exemple, l'exécution de moins d'instructions et l'accès moins fréquent à la mémoire externe peuvent réduire le besoin en énergie d'un système [9]. Dans le passage en revue des techniques de compression (cf. section 2.4), nous mentionnons leur effet sur la consommation de l'énergie lorsque ce facteur est étudié.

Pour présenter un état de l'art sur une thématique aussi riche, il est judicieux de procéder par classe de techniques. Les critères de classification peuvent porter sur le niveau d'application (source, compilateur, exécutif ...), la nature des méthodes utilisées, les spécificités du code compressé, etc. Dans la section suivante, nous présentons ces différentes classes de techniques de compression.

2.3 Les différentes classifications

En observant les techniques de compression, nous constatons que plusieurs classification ont été proposées. Dans un premier temps, nous décrivons les différentes classes et leurs critères. Ensuite, nous dressons un constat basé sur l'analyse et la critique de chaque classe. Ce constat nous permettra d'évaluer le niveau d'adéquation des classifications par rapport aux techniques de compression du code Java.

2.3.1 Compression générique ou spécifique à l'architecture

Les optimisations spatiales de haut niveau peuvent être appliquées par un compilateur qui produit le fichier *class*. Il en résulte un code optimisé indépendant de l'architecture du système sur lequel il va être exécuté. Cependant, il existe un ensemble d'optimisations qui dépendent de la machine et qui doivent être adaptées à une architecture spécifique. Elles ne sont donc applicables que lorsque le code machine est généré à partir du bytecode (par exemple au niveau du compilateur JIT). Budimlić et Kennedy [13] opte pour ce critère de classification et distingue trois classes.

Les techniques de la première classe produisent un pseudo code pour la JVM indépendamment de la plateforme sur laquelle il sera exécuté. De ce fait, aucun changement, au niveau de l'environnement d'exécution Java, n'est nécessaire puisque les techniques préservent les spécificités de Java comme la portabilité et la sécurité. À titre d'exemple, citons les techniques d'élimination de code mort [12] et la propagation de constantes [60] que nous verrons ultérieurement (cf. section 2.4.2.4).

La deuxième classe va à l'encontre de la première en désavantageant la portabilité et la sécurité de la Machine Virtuelle Java au bénéfice d'un compilateur optimisateur de code natif hautement sophistiqué. Les techniques de compression incluses dans cette classe sont inspirées des spécificités architecturales de la machine hôte. Budimlić [13] transforme le code Java en ILOC (Intermediate Language for Optimizing Compilers) pour appliquer des techniques de compression spécifiques aux machines RISC. Benitez [8] montre, par l'exemple, que certaines techniques d'optimisation indépendantes (supposées l'être) de l'architecture ne sont efficaces que lorsqu'elles sont appliquées sur une représentation bas niveau une fois l'information sur la machine cible connue. Les techniques de cette classe tirent donc bénéfice de cette connaissance préalable de l'architecture offrant ainsi des taux de compression très compétitifs par rapport aux techniques de la première approche. La

perte de la portabilité du code généré représente leur inconvénient majeur.

La troisième et dernière classe combine les critères des deux premières en compromettant la portabilité en faveur d'une meilleure performance. Les techniques de cette classe relaxent les contraintes relatives au mode d'exécution du code Java pour améliorer significativement la performance. Ces techniques sont adaptées aux applications qui requièrent une exécution via Internet et une sécurité élevée (assurée par les instructions de la JVM), tout en se limitant aux plateformes dont le mode de fonctionnement est supporté.

L'un des atouts du langage Java est sa portabilité. Cet atout provient de la présence de la forme intermédiaire du code, contenu dans le fichier *class*, et l'intervention de la machine virtuelle Java comme interpréteur entre le code et le matériel. Les techniques de compression spécifiques à l'architecture ne sont pas adéquates au code Java puisqu'elles le privent de son avantage majeur à savoir la portabilité. En se basant sur cette propriété, la classe des techniques spécifiques à l'architecture n'a pas lieu d'être. Ce qui remet en cause la convenance de cette classification à notre contexte.

2.3.2 Exécution avec ou sans décompression préalable

Debray [18] et Franz [25] adhèrent à une optique qui différencie les techniques selon la nécessité d'une phase de décompression préalable à l'exécution ou non. Ainsi, la forme compressée d'un programme peut être soit décompressée (ou peut être compilée) avant l'exécution [21] [25] [24], soit exécutée (ou interprétée [26] [25]) sans décompression [15] [22].

Les techniques de la première classe donne une représentation compressée plus compacte que celles de la seconde, mais entraîne un surcoût de décompression nécessaire avant l'exécution. Le temps de la décompression peut être négligeable. D'ailleurs, il peut même être compensé par l'économie de temps faite lors de la transmission et de l'extraction [24]. Par contre, un problème peut surgir, il s'agit de l'espace nécessaire pour placer le code décompressé. Ce problème est devenu peu important avec l'avènement des techniques de décompression partielle ou de décompression à la volée [7] [21]. Néanmoins, ces techniques requièrent la modification du processus d'exécution ou le matériel du composant.

Cette classification, n'étant composée que de deux classes, présente la particularité de regrouper plusieurs techniques dans une même classe. Par conséquence, nous estimons qu'elle n'est pas assez discriminante. Elle ne permet donc pas de structurer les différentes

classes de techniques de compression vis à vis de notre problématique.

2.3.3 Purificateur, compresseur syntaxique et compresseur sémantique

Les techniques de compression, selon Antonioli [5], agissent essentiellement sur la syntaxe et la sémantique du code, en plus de la purification. Ainsi nous distinguons trois classes de techniques :

1. **Les compresseurs purificateurs :** ils réduisent le contenu des classes Java à travers différentes analyses. Les méthodes utilisées localisent les portions de code qui sont réellement nécessaires pour l'exécution de l'application et suppriment le reste des portions. Bien que le premier outil, JDistill [48], est restreint aux membres des classes, champs et méthodes, des outils plus récents, comme jax [55], transforment aussi la hiérarchie des classes, collapsent ou suppriment les classes qui ne sont pas utilisées directement. Ces outils offrent une réduction du code entre 17% et 32% et sont d'usage juste avant le stockage de l'application.
2. **Les compresseurs sêmantiques du fichier objet :** ils essaient de déterminer un ensemble alternatif d'instructions qui encodent le même programme avec moins d'octets. L'idée générale est de remplacer les séquences d'instructions récurrentes avec des super opérations. Proebsting et al. [50] [26] présentent une alternative d'algorithme qui sélectionne l'ensemble des super opérations pour un programme donné. Ainsi, chaque application est écrite avec son propre ensemble d'instructions et par conséquent nécessite un interpréteur personnalisé. Franz [23] et Kistler [36] proposent d'identifier les patterns d'instructions et de les encoder avec un dictionnaire dynamique.

   La compression sémantique est typiquement limitée pour les séquences d'instructions et ne touche pas au reste du fichier objet, comme les constantes.
3. **Les compresseurs syntaxiques :** ils prennent en considération la structure du programme plutôt que son contenu et tentent d'y localiser la redondance. La transformation des noms de classe [5] en est un bon exemple : elle est basée entièrement sur la définition grammaticale de l'identificateur Java et non sur une analyse statistique. Corless [16] [29] a établi le fondement de la compression d'un fichier *class*. Bradley et al. [11] l'ont étendu vers des groupes de fichiers *class* en introduisant

l'idée du partage de la table de constantes entre les classes. Pugh [51] est allé encore plus loin dans ce sens, en établissant, avec *pack*, un format complètement nouveau pour représenter un ensemble de fichiers *class*.

Parmi les trois classifications présentées, la dernière semble être la mieux adaptée à notre contexte, à savoir la compression des applications Java. En effet, elle couvre avec ses trois classes l'ensemble des techniques existantes. En plus, elle a le mérite d'avoir des classes à frontières bien définies. Autrement dit, il n'existe pas de techniques qui sont à cheval sur deux ou plusieurs classes de techniques.

Cependant, nous ne relevons pas d'adéquation aux particularités du code Java. Cette classification se veut générique. En effet, elle peut être facilement adoptée pour classer les techniques relatives à un format de code autre que Java. En revanche, le langage Java se distingue des autres langages avec des caractéristiques structurelles qui lui sont propres (cf. chapitre 1 section 1.3.3).

2.3.4 Proposition d'un nouvelle classification

Nous avons cité jusqu'à présent trois types de classification. Les critères adoptés reflètent des visions différentes des techniques de compression. Elles ont toutes le mérite de pouvoir identifier deux ou trois classes qui couvrent l'ensemble des techniques existantes. Toutefois, nous avons révélé certains points négatifs (énoncés vers la fin de la précédente section) qui montrent leurs limites quant à leur adoption pour notre étude. Sur la base des remarques, énumérées précédemment, concernant les limites de ces classifications, une nouvelle classification est proposée.

La nouvelle classification se doit d'être contextuellement adéquate : propre au code Java. Autrement dit, elle doit tenir compte, en priorité, de la structure même du code Java. Par ailleurs, l'examen de la composition du format de fichier *class* montre l'existence de deux composants primordiaux : le bytecode et la table des constantes. Ces deux composants tirent leur importance, entre autres, de par l'espace qu'ils occupent dans le fichier *class* Java (cf. figure 2.2).

Le composant table des constantes est considéré comme un champ d'investigation prometteur. D'ailleurs, il représente 67% de la taille d'une classe. Nous lui consacrons une classe de techniques de compression. Quant au composant bytecode, il occupe 26% du fichier *class* Java. Bien que la taille du bytecode soit moins importante que celle de la

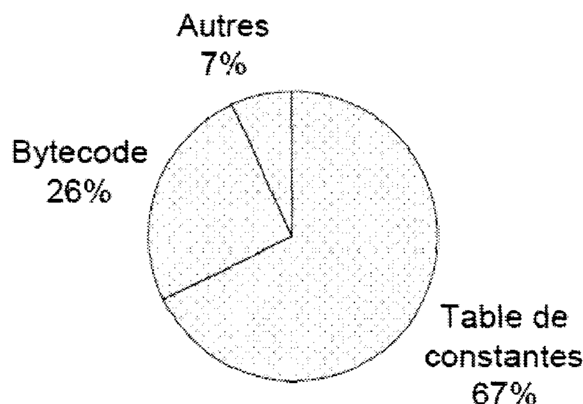

FIG. 2.2 – Rapport des composants d'un fichier *class* Java

table des constantes, elle reste tout de même un composant non négligeable du fichier *class* Java. Nous lui consacrons une deuxième classe. Étant donnée que le bytecode se compose d'opcodes et d'opérandes, nous consacrons une sous-classe pour chacun d'entre eux. Néanmoins, il existe des techniques qui n'agissent pas particulièrement sur un composant ou l'autre exclusivement. Certaines techniques considèrent le fichier *class* Java comme un tout indissociable. D'autres entraînent des changements au niveau de plusieurs composants à la fois. Nous regroupons ces techniques dans une même classe dite globale.

Notre classification a le mérite en plus d'être affine. Il ne s'agit pas d'identifier uniquement deux ou trois classes sur lesquelles est réparti l'ensemble des techniques de compression, mais plutôt d'une structure semblable au format de fichier *class* : dans une même classe, nous pouvons retrouver d'autres classes (sous classes). Force est de constater que cette classification tient compte de la nature des composants du fichier *class* Java. En effet, le bytecode est composé d'instructions (opcodes) et de paramètres (opérandes). Quant à la table des constantes, elle est formée de données qui représentent, entre autres, des valeurs et des descriptions.

2.4 Techniques élémentaires de compression

Nous présentons dans cette section les techniques de compression de code. Nous les qualifions des techniques élémentaires vu leur aspect granulaire. Avant d'aborder les tech-

niques logicielles, nous faisons une brève description des techniques matérielles de compression. Bien que ces techniques se situent sur le plan matériel et non logiciel, leur exploration nous semble bénéfique par les enseignements que l'on peut tirer. Un tour d'horizon des techniques de compression matérielles fait l'objet de la section qui suit.

2.4.1 Techniques matérielles

Les techniques de niveau matériel, explorées dans cette section, s'apparentent aux méthodes utilisées au niveau logiciel. De plus, les résultats obtenus (taux de compression) sont des indicateurs des possibilités. On se limite simplement à donner, sans être exhaustif, quelques exemples et principes de leurs fonctionnements.

Wolfe, Kozuch et Chainin [62] [39] [40] ont analysé l'utilisation du code de Huffman pour la compression de programmes pour des systèmes embarqués. Les programmes sont compressés en mémoire centrale, par ligne d'antémémoire ; c'est-à-dire qu'une séquence d'octets du programme non compressé, remplissant une ligne de l'antémémoire, est compressée par un code de Huffman, préétabli une fois pour toute (modèle statique). Chaque séquence compressée est alignée sur une frontière d'octet. Lors du transfert de la mémoire centrale à l'antémémoire, la séquence est décompressée. Ainsi, le processeur ne traite pas le code compressé mais plutôt les octets décompressés. Leur facteur de compression est de 70% pour le Mips R2000. D'autres travaux [7] [6] ont incorporé un décodeur de Huffman (à modèle statique) au matériel. Le taux de compression est à peu près le même (environ 73%).

Ces techniques présentent l'inconvénient d'être spécifiques à une architecture donnée. Néanmoins, leur usage est bénéfique lorsqu'une attention particulière est apportée au temps de décodage. En effet, elles permettent l'exécution d'une tâche (en l'occurrence la décompression) en un temps réduit car l'algorithme sous-jacent est câblé. Notons que les techniques employées au niveau matériel peuvent être retrouvées au niveau logiciel et vice versa comme le codage de Huffman (cf. section 2.4.2.1). De plus, les résultats obtenus (taux de compression) donnent une idée sur les performances que l'on peut atteindre en implémentant de façon logicielle les techniques matérielles. En revanche, le temps de décodage est plus lent.

Les techniques présentées dans la suite du chapitre se situent au niveau logiciel. Nous les présentons selon la classification définie dans la section 2.3.4.

2.4.2 Techniques globales

Parmi les techniques globales, nous retrouvons les techniques classiques de compression des données. Elles se déclinent selon deux catégories générales : statistique et dictionnaire.

2.4.2.1 La compression statistique

La compression statistique utilise les fréquences d'apparition d'un caractère pour déterminer la taille du code qui le remplace. Les caractères fréquents sont codés avec le plus petit code, minimisant ainsi la taille totale du texte compressé. L'encodage de Huffman [30] en est l'exemple le plus connu.

Le principe du codage de Huffman repose sur la création d'un arbre composé de nœuds. Il recherche, tout d'abord, le nombre d'occurrences de chaque caractère de la chaîne à compresser. Chaque caractère constitue une des feuilles de l'arbre à laquelle on associe un poids valant son nombre d'occurrences. Puis, l'arbre est créé suivant un principe simple : on associe à chaque fois les deux nœuds de plus faibles poids pour donner un nœud dont le poids équivaut à la somme des poids de ses fils jusqu'à n'en avoir plus qu'un : la racine. On associe ensuite à la branche la plus faible d'un nœud le code 0 et la plus forte le code 1. Nous appliquons l'algorithme de Huffman sur une chaîne formée par les caractères A, B, C, D et E dont les probabilités respectives sont 0.4, 0.2, 0.2, 0.1 et 0.1. L'arbre de Huffman correspondant à cette chaîne est illustré par la figure 2.3 :

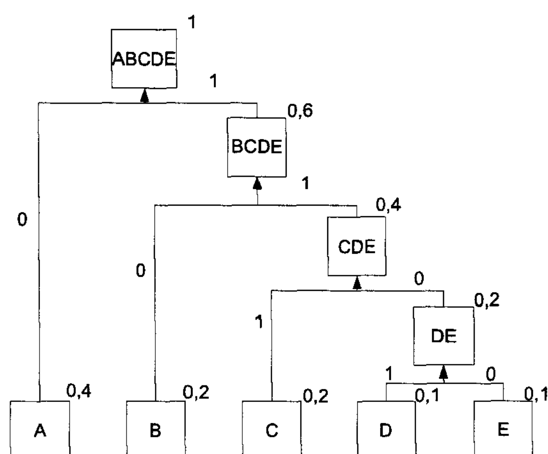

FIG. 2.3 – Arbre de Huffman

L'encodage de Huffman affecte donc aux caractères A, B, C, D et E les codes 0, 10, 111, 1101 et 1100 respectivement. La taille moyenne de ce code est de 2.2bits/symbole. Son calcul est donnée par l'équation 2.1.

$$0.4 \times 1 + 0.2 \times 2 + 0.2 \times 3 + 0.1 \times 4 + 0.1 \times 4 = 2.2 bits/symbole \tag{2.1}$$

Notons que le code Huffman relatif à une chaîne n'est pas unique. C'est pour cette raison que l'on doit disposer de l'arbre pour pouvoir procéder à la décompression et retrouver la chaîne originale.

2.4.2.2 La compression par dictionnaire

La compression par dictionnaire sélectionne des phrases entières de caractères communs et les remplace par du code. Ce code est utilisé comme un index à l'entrée du dictionnaire qui contient la séquence de caractères originale. La compression résulte du fait que le code utilise moins de bits que les caractères qu'il remplace.

Lempel et Zip [64] ont établi les algorithmes d'encodage par dictionnaire les plus connus, LZ77 et LZ78. Ces derniers algorithmes ont servi par la suite de base pour d'autres variantes d'encodage (LZW [61]) et de formats (zip, gif, etc). La compression selon l'algorithme LZ77 considère le fait que les chaînes de caractères (mots, phrases, ...) se reproduisent plusieurs fois dans le message à compresser. Le principe de compression utilise une fenêtre coulissante divisée en deux parties : une fenêtre de parcours (contient le code en cours d'encodage) et un tampon d'anticipation (contient le texte non encore codé). L'algorithme LZ77 se présente comme suit :

```
Pointer la position courante sur le début de la chaîne.
TANT QUE (le buffer de lecture n'est pas vide) FAIRE
DEBUT
Trouver une équivalence entre une chaîne de caractères
dans la fenêtre de parcours et le buffer d'anticipation.
SI (la longueur de la chaîne équivalente > 0) ALORS
   On retourne le triplet (p, t, c);
      où p : la position de la chaîne équivalente
             dans la fenêtre de lecture ;
         t : sa taille;
```

```
        c : caractère suivant dans le buffer.
    On décale la fenêtre de la taille t+1.
SINON
    Renvoyer le triplet (0, 0, c);
        où c : le premier caractère du buffer
    On décale la fenêtre d'un caractère.
FIN SI
FIN TANT QUE
```

La décompression d'un encodage LZ77 présente la particularité d'être rapide. Ceci donne un avantage de taille puisqu'il n'engendre pas un surcoût significatif à l'exécution.

2.4.2.3 Suppression des informations de débogage (purificateur)

Les informations de débogage, présentes dans le code, servent à assister le développeur à la recherche d'éventuelles erreurs dans le programme. Les systèmes embarqués ne sont pas spécialement conçus pour se charger de tâches optionnelles telles que le débogage qui peut être effectué sur une station de travail par exemple. Yourst [63] propose de réduire la taille des fichiers *class* en supprimant ces informations dispensables.

Les informations de débogage sont constituées par les entrées de la table de constantes qui ne représentent pas des informations sur d'autres constantes ou sur les types. Ces informations n'incluent pas également les attributs *Code*, *ConstantValue* et *Exceptions*. Dans un fichier *class* Java, les attributs *LineNumberTable*, *LocalVariableTable* et *SourceFile* représentent les informations de débogage.

- LocalVariableTable : un attribut optionnel de taille variable de l'attribut *code* d'une méthode. Il peut être utilisé par les débogueurs pour déterminer la valeur d'une variable locale donnée durant l'exécution d'une méthode.
- LineNumberTable : un attribut optionnel de taille variable dans la table des attributs de l'attribut *code* d'une méthode. Il peut être utilisé pour déterminer quelle partie de la table de code correspond à un numéro de ligne dans le fichier source original.
- SourceFile : un attribut optionnel de taille fixe dans la table des attributs de la structure *ClassFile*. Il représente le nom du fichier source .java.

2.4.2.4 Les optimisations à lucarne pour la compression

Lors de la génération de code, la séquence d'instructions produite n'est pas toujours très efficace. Les techniques d'optimisation à lucarne (appelées aussi *peephole*) consistent à examiner des petites séquences d'instructions et à essayer de les remplacer par une séquence plus efficace.

Ce type d'optimisation s'applique généralement sur un code sous forme d'instructions avec des variables et des registres. En revanche, lorsqu'il s'agit d'un code pour machine à pile tel que le bytecode, la tâche est compliquée [57]. Ceci s'explique, entre autres, par le fait que les expressions du bytecode sont arbitrairement longues et non assez explicites. Il est donc nécessaire d'appliquer des transformations sur le bytecode qui passe par des représentations intermédiaires (cf. figure 2.4). Ces transformations sont guidées par les techniques d'analyse de flots de données [35]. Dans ce sens, la représentation SSA (pour Static Single Assignement ou AUS pour Assignation Unique Statique) est utilisée pour rendre possible l'application des techniques d'optimisation à lucarne.

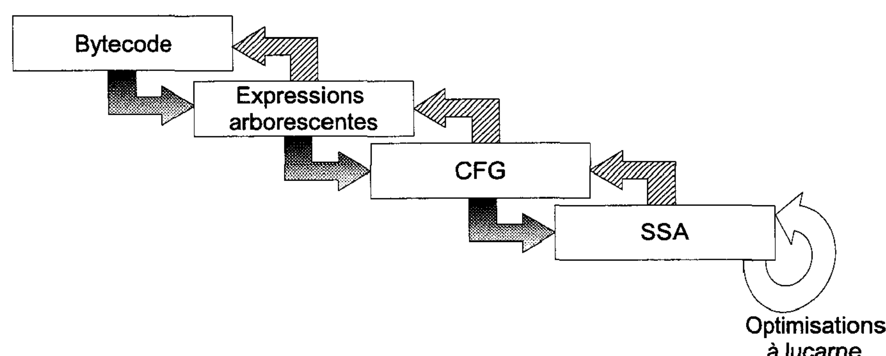

FIG. 2.4 – Application des optimisations à lucarne

D'abord, la représentation du bytecode (code à pile) est convertie vers des expressions arborescentes puis vers un graphe de flot de contrôle CFG (Control Flow Graph). Par la suite, ce graphe est transformé pour aboutir à la forme SSA. Les optimisations à lucarne sont donc appliquées sur la SSA. Pour finir, le bytecode (compressé) est reconstruit à partir de la forme SSA en appliquant la fonction inverse des transformations précédentes.

Les techniques de suppression du code mort et de propagation de constantes sont des exemples de techniques d'optimisation à lucarne.

Suppression du code mort : Cette technique est une optimisation assez connue dans les langages traditionnels [12]. Elle consiste à supprimer les portions de code ne pouvant pas être atteintes au cours de l'exécution, ou bien atteintes, mais ne produisant pas d'effet sur les résultats du programme. Son application à Java nécessite un soin particulier. En effet, plusieurs instructions ont le potentiel de déclencher une exception, ce qui pourrait limiter les opportunités d'élimination de code mort. Budimlić [13] traite ce mécanisme des exceptions en utilisant les transformations illustrées par la figure 2.4.

Propagation des constantes : Il s'agit d'un problème d'analyse du flot de données global. Elle consiste à "découvrir les valeurs qui sont constantes lors de toutes les exécutions possibles d'un programme et de propager ces valeurs aussi loin que possible à travers le programme" [60]. L'application de cette technique se passe également au niveau de la représentation SSA. Le schéma de transformations de la figure 2.4 est de nouveau utilisé.

2.4.3 Techniques relatives au bytecode

Le bytecode Java peut être séparé en deux groupes [52] : un premier groupe pour les opcodes et un deuxième pour les opérandes. Des techniques appropriées sont appliquées à chaque groupe.

2.4.3.1 Opcodes

Des observations faites sur l'ensemble des opcodes permettent de guider le choix des techniques de compression à appliquer. D'une part, la fréquence d'apparition des opcodes dans le fichier *class* est très irrégulière d'un opcode à un autre. L'algorithme de Huffman (cf. section 2.4.2.1 et 2.4.2.2) est bien adapté à un contexte de ce genre. Il permet de réduire le nombre de bits pour représenter les opcodes les plus fréquents. Une table de correspondance entre les opcodes et les symboles basée sur leur fréquence de distribution est ainsi construite.

D'autre part, les opcodes contiennent des redondances de quelques paires d'entre eux. L'encodage de Huffman peut encore être utilisé pour une meilleure compression. Ainsi, il associe un préfixe pour chaque paire d'opcode redondante.

La dernière observation s'étend aussi sur un nombre d'opcodes parfois supérieur à deux. En effet, nous retrouvons dans le bytecode des séquences d'opcodes redondantes

(appelées aussi patterns). Dans la pratique, deux approches sont évoquées pour réduire les redondances dans le bytecode Java.

Une première approche [52] consiste à utiliser les chaînes de Markov [17]. Chaque opcode peut être représenté comme un état dans le modèle d'état de Markov. Une séquence d'opcodes peut être ainsi représentée par le premier état et une séquence de transition d'états.

Une deuxième approche [52] [14] préconise l'usage des instructions non définies par la spécification de la Machine Virtuelle Java [45]. Clausen et al. [14] créent des macros pour compresser le bytecode des programmes Java. Cette méthode consiste à construire de nouveaux codes opérationnels pour des séquences fréquentes de bytecode ; ces séquences sont appelées des macros. Les macros sont insérées dans les fichiers objets *class* sous la forme de séquences de codes opérationnels. Une macro peut contenir des références à d'autres macros, sans introduire de récursion. Cependant, aucun paramètre n'est utilisé et leur qualité est limitée au nombre de codes opérationnels disponibles. Ce qui donne environ 50 macros si un seul octet est utilisé, car il y a au départ 203 codes opérationnels pour la JVM (sans compter les 23 instructions "quick" et l'instruction breakpoint de débogage). Dans le cas de la machine virtuelle JavaCard [54], 152 codes opérationnels sont disponibles. Toutefois, pour les tests rapportés, il y a toujours 60 marcos, et dans certains cas, plus de 160 ; dans ces cas les codes opérationnels peuvent avoir deux octets. Des facteurs de compression de 80% à 70% sont atteints pour plusieurs benchamrks. L'implantation Harissa [49] est utilisée pour mesurer la perte de vitesse d'exécution. Celle-ci varie selon les benchmarks utilisés : entre 2% et 27%.

2.4.3.2 Opérandes

Chaque opcode est suivi d'un nombre prédéfini d'opérandes (excepté les opcodes *tableswitch* et *lookupswitch*). La majorité de ces opérandes sont des références vers la table des constantes ou vers le début d'une méthode. Rayside [52] compresse ces deux groupes d'opérandes en appliquant la même technique de façons différentes.

Dans le premier groupe, les références vers la table de constantes sont représentées, par définition, par deux octets. Il propose de les coder sur uniquement un octet lorsque le nombre total d'entrées de la table de constantes ne dépasse pas 256. Encore mieux, en utilisant seulement le nombre de bits nécessaires pour référencer la table des constantes. En ce qui concerne le deuxième groupe, la même technique est également appliquée avec

un traitement préalable. Ce traitement consiste à modifier le principe de référencement de début de méthode pour augmenter la probabilité d'avoir un nombre de références inférieur à 256. Par définition [45], le début d'une méthode est référencé par un entier signé qui représente le nombre d'octets depuis l'opcode courant jusqu'à l'opcode cible. Le calcul du nombre d'octets tient compte des opcodes ainsi que des opérandes. Le nouveau principe de référencement se contente des opcodes car la taille des opérandes qui suivent un opcode est connue.

2.4.4 Techniques relatives à la table des constantes

La table des constantes représente un champ d'investigation intéressant pour la réduction de l'encombrement mémoire des fichiers *class* Java. Ceci s'explique par le fait qu'elle représente en moyenne les deux tiers de la taille d'un fichier *class* Java. En plus, la présence de chaînes de caractères favorise la redondance. D'ailleurs, il existe plusieurs techniques de compression relatives à la table des constantes [1] ; six parmi elles sont décrites dans les paragraphes suivants.

2.4.4.1 Restructuration des descripteurs de méthodes

Cette technique consiste à transformer les constantes Utf8 représentant les descripteurs de méthodes (*MethodDescriptor*) par des indices vers des entrées de la table de constantes qui représentent les paramètres et le résultat de la méthode en question. [52] utilise la structure intermédiaire suivante pour mettre en œuvre cette technique :

```
MethodDescriptor
{
    parameter_count;
    parameter_indices[parameter_count];
    return_type_index;
}
```

Avec :

- *parameter_count* : le nombre de paramètres de la méthode ;
- *parameter_indices* : le tableau d'indices vers les paramètres ;
- *return_type_index* : l'indice du résultat de la méthode.

L'intérêt de cette restructuration est de créer une redondance entre les nouvelles entrées et celles déjà existantes dans la table de constantes.

2.4.4.2 Représentation arborescente des noms de classes

Cette technique repose sur le principe d'élimination des noms de classes ou de packages récurrents en adoptant une représentation arborescente [52]. Selon la spécification de la Machine Virtuelle Java [45], le nom d'une classe est représenté par le nom entier de son chemin en séparant les super classes par des '/'. À titre d'exemple, la représentation de la classe *Object* est la suivante :

```
java/lang/Object;
```

La technique permet l'ajout de la structure suivante à la table des constantes :

```
QualifiedName
    {
        u2 name_index;
        ti parent_index;
    }
```

Avec :
- *name_index* : index d'une constante *Utf8* contenant le nom de la classe ou du package;
- *parent_index* : index d'une structure *QualifiedName* qui représente la classe mère de la classe courante.

Ainsi, la représentation des noms de classes se fait d'une façon arborescente de manière à ce que les nœuds correspondent aux noms de packages et les feuilles aux noms de classes.

2.4.4.3 Indexage implicite

Plusieurs structures de la table des constantes contiennent des indices vers d'autres structures. Il devient ainsi possible de réordonner les entrées de façon à ce que les indexes soient implicites. Ainsi, les indices peuvent être supprimés. À titre d'exemple, en remplaçant les champs déclarés par une classe (ou interface) ceci permet de supprimer les champs *class_or_interface_index*. De la même manière, le champ *name_index* peut être éliminé en ordonnant la partition des *Utf8* selon l'ordre d'apparition des champs *name_index*.

2.4.4.4 Partitionnement de la table des constantes

Comme nous l'avons décrit antérieurement (cf. chapitre 1 Section 1.3.3), le type d'une entrée de la table de constantes est identifié grâce à une étiquette de taille d'un octet. La réorganisation des entrées selon le type permet de se passer des étiquettes. Le groupe d'entrées de même type est précédé par un compteur qui représente le nombre d'entrées du groupe. La transformation du fichier *class* de *HelloWorld* selon cette technique donne un nouveau format illustré par la figure 2.5.

Avant partitionnement		Après partitionnement
		NameAndType_Count = 3
		1 [1, 2]
		2 [13, 14]
1 Methodref	[6, 15]	3 [16, 17]
2 Fieldref	[16,17]	Methodref_Count = 2
3 String	[18]	1 [2, 1]
4 Methodref	[19,20]	2 [4,3]
5 Class	[21]	Class_Count = 4
6 Class	[22]	1 [10]
7 Utf8	"<init>"	2 [11]
8 Utf8	"()V"	3 [12]
9 Utf8	"Code"	4 [15]
10 Utf8	"LineNumberTable"	String_Count = 1
11 Utf8	"main"	1 [9]
12 Utf8	"([Ljava/lang/String;)V"	Fieldref_Count = 1
13 Utf8	"SourceFile"	1 [3,2]
14 Utf8	"HelloWorld.java"	Utf8_Count = 17
15 NameAndType	[7, 8]	1 "<init>"
16 Class	[23]	2 "()V"
17 NameAndType	[24, 25]	3 "Code"
18 Utf8	"Hello World!"	4 "LineNumberTable"
19 Class	[26]	5 "main"
20 NameAndType	[27, 28]	6 "([Ljava/lang/String;)V"
21 Utf8	"HelloWorld"	7 "SourceFile"
22 Utf8	"java/lang/Object"	8 "HelloWorld.java"
23 Utf8	"java/lang/System"	9 "Hello World!"
24 Utf8	"out"	10 "HelloWorld"
25 Utf8	"Ljava/io/PrintStream;"	11 "java/lang/Object"
26 Utf8	"java/io/PrintStream"	12 "java/lang/System"
27 Utf8	"println"	13 "out"
28 Utf8	"(Ljava/lang/String;)V"	14 "Ljava/io/PrintStream;"
		15 "java/io/PrintStream"
		16 "println"
		17 "(Ljava/lang/String;)V"

FIG. 2.5 – Partitionnement de table de constantes

Deux octets suffisent pour coder le compteur. Dans le cas où une section ne contient aucun élément, les deux octets sont perdus. Latendresse [44] propose d'utiliser les codes Start-Step-Stop pour mieux perfectionner l'encodage et éviter la perte des deux octets.

2.4.4.5 Rétrécissement des noms

Notons que la technique de rétrécissement des noms est l'une des fonctionnalités accomplies par les obscurcisseurs dont le rôle est de protéger le code afin d'empêcher le rétro-engineering. La compression des noms se fait à travers l'opération de renommage des packages, classes, méthodes et variables, en remplaçant les noms originaux par des lettres comme a, b, c, etc (exemple de la figure 2.6).

``public class Voiture
{
    int prix;
    public int AvoirPrix() {return prix;}
}```
	``public class a
	{
	    int b;
	    public int c() {return b;}
	}```

Avant rétrécissement des noms	**Après rétrécissement des noms**

FIG. 2.6 – Rétrécissement des noms

Ainsi, les noms sont sans signification et le code généré par décompilation est incompréhensible. Néanmoins, certains noms ne peuvent pas être compressés. Ces noms font référence à une classe, un champ dans une librairie externe, une méthode qui redéfinie d'autres méthodes dans une librairie externe, un programme accédé avec une réflexion, le nom de la classe contenant la routine *main*, les constructeurs et les initialiseurs statiques.

2.4.4.6 Partage de la table des constantes

Chaque fichier *class* java doit stocker dans sa table de constantes le nom de toute classe référencée (appartenant au package de la classe en question ou faisant partie de l'API Java[6]). Il en va de même pour les noms et signatures des méthodes. Ceci entraîne un nombre exceptionnel de redondances entre les fichiers.

Bradley [11] propose dans son format JAZZ (cf. figure 2.7) de partager les constantes entre les zones des constantes de différentes classes. Ces constantes sont groupées dans une seule table qui contient les constantes relatives à toutes les classes. Le nombre de constantes dans la table unifiée est normalement inférieur à la somme des constantes dans les classes séparées. Par contre, ce nombre est considérablement supérieur à celui de n'importe quelle classe impliquée dans l'unification. Une représentation plus large des indexes doit donc être utilisée pour tenir compte de ce changement.

[6]Application Programming Interface : représente les librairies standard Java.

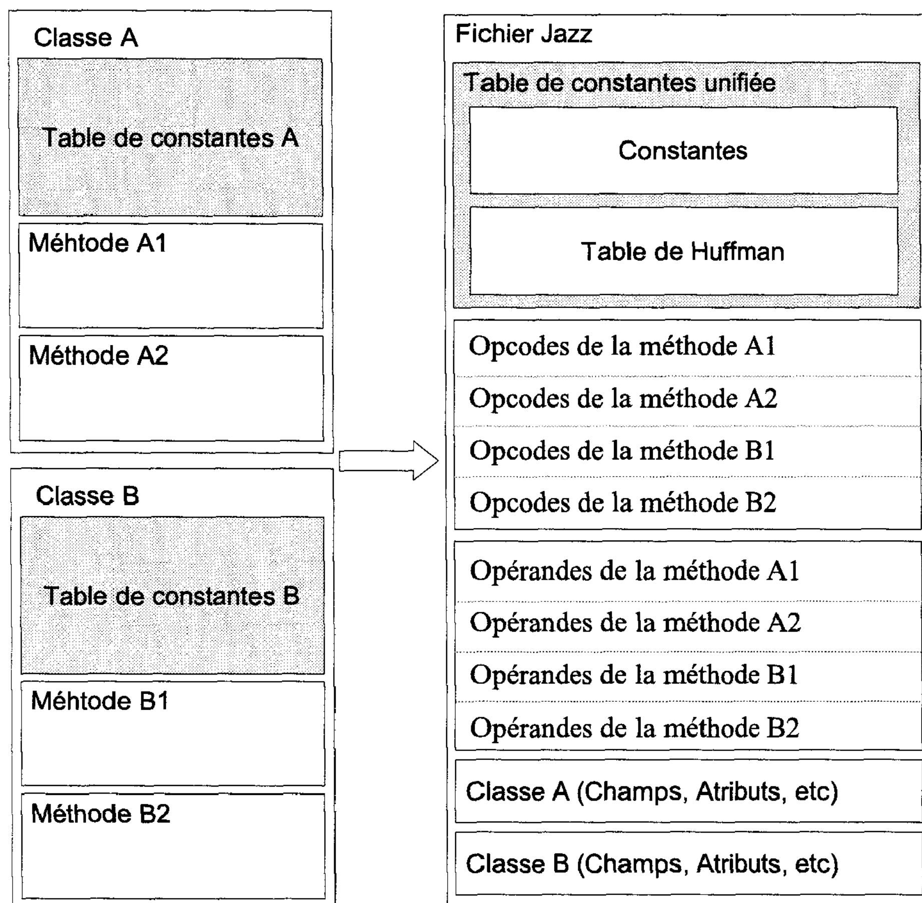

FIG. 2.7 – Table de constantes partagées du format JAZZ

2.4.5 Représentations alternatives au format de fichier *class* Java

Après avoir présenté les solutions générales pour la compression de fichiers objets Java dans la section précédente, celle-ci traite explicitement les fichiers *class* Java. Nous présentons quelques travaux qui exploitent certaines techniques de compression parmi celles précédemment présentées. Le choix des techniques et les éléments concernées par la compression donne une variété de représentations des fichiers objets Java.

Michael Franz [36][24] a proposé un format (Slim binaries) comme mécanisme de distribution de programmes compacts indépendants de la machine hôte. Il s'agit d'une représentation intermédiaire du programme Java bien plus compacte que la représentation sous forme de bytecode. Le format est obtenu grâce à l'encodage de l'abstraite arbre syntaxique et de la table de constantes du programme. Cependant, le programme compressé ne peut pas être interprété ; il nécessite une génération dynamique du code au moment du chargement. Le format de Franz n'est pas approprié aux systèmes embarqués

à cause des ressources supplémentaires (espace mémoire, temps processeur) nécessaires à son exécution.

Corless et Horspool [16] [29] ont introduit avec CLAZZ un nouveau format pour les fichiers *class*. Les auteurs font la différenciation entre la compression 'sans perte de données' et avec 'préservation de la sémantique' et optent pour le deuxième choix. Ils proposent des techniques adaptées aux fichiers *class* Java. La compression passe par la localisation des parties les plus encombrantes du fichier et leur compactage avec des algorithmes taillés sur leur mesure.

Dans la continuité du projet précédent, Bradley et al. [11] présente JAZZ, une meilleure alternative pour le format de fichier jar. La principale contribution de Jazz est la construction de groupes d'informations similaires recueillies de l'ensemble des classes. Le regroupement a deux objectifs : d'abord les classes Java ont des références communes à la librairie de classes ; une table de constantes partagée élimine les répliques. Ensuite, l'algorithme de compression zip est plus efficace lorsque les propriétés statistiques demeurent comparables sur de longues périodes. Pour mieux favoriser cette homogénéité, Jazz compresse les opcodes et les opérandes séparément [21].

Le travail de Pugh [51] a été appliqué principalement à la compression des fichiers objets Java avec décompression préliminaire avant exécution. Les fichiers objets Java contiennent une table de constantes occupant un espace substantiel comparativement au code-octet. Ainsi, l'auteur utilise plusieurs méthodes différentes, appliquées à un même fichier objet pour bien les compresser. Il vise à la fois le code octet et la table de constantes. Au niveau du code-octet, la technique générale est de séparer les codes opérationnels des opérandes en deux flots d'octets [27] pour rendre homogènes les données et par la suite mieux déceler la présence de répétitions. Ces flots sont compressés séparément par différentes méthodes dont celle utilisée par gzip. Au niveau de la table des symboles, un encodage performant est important pour minimiser l'impact du large nombre de symboles résultants de la combinaison de plusieurs classes. Des résultats impressionnants sont obtenus : l'auteur rapporte des facteurs de compression de 17% à 49% par rapport au format jar. Par ailleurs, les classes compressées ne peuvent être exécutées avant décompression.

Rayside [52] propose un nouveau format plus compact. Il s'agit d'un format interprétable comme alternative au format binaire (obtenu par compression de la structure existante en se focalisant sur la table de constantes et la table du code). Contrairement au travail de Pugh, Horspool et Corless, cette représentation, tout comme Jax [55], ne

nécessite pas une phase de décompression qui précède l'exécution. Cependant, le nouveau format interprétable requiert soit une JVM légèrement modifiée, soit un chargeur de classes personnalisé.

2.5 Conclusion

Nous avons consacré ce chapitre à l'état de l'art des techniques de compression du code Java. Pour commencer, nous avons défini, dans la première section, l'optimisation qui est le cadre général dans lequel se situe notre travail de recherche. Nous avons également précisé ses frontières : le champ d'investigation, et par conséquent l'état de l'art, qui concernent la compression des fichiers *class* Java. Nous avons exclu les techniques de compression qui agissent sur le code source ou encore sur le code natif.

Historiquement, le domaine de l'optimisation spatiale a été marqué par des faits comme par exemple l'apparition des systèmes embarqués ou la création du langage Java. Ceci explique l'évolution de l'intérêt accordé par les chercheurs à ce domaine. Le nombre important de travaux dans ce domaine nous a conduit à faire une étude préalable de leurs classifications. Ceci a fait l'objet de la deuxième section dans laquelle nous avons présenté trois classes de techniques. La première classification est basée sur l'aspect générique ou spécifique des techniques de compression par rapport à l'architecture. La deuxième classification est axée sur le code compressé généré ; nécessitant une décompression préalable à l'exécution ou exécuté/interprété directement par la machine virtuelle Java. La troisième différencie entre les compresseurs syntaxiques, sémantiques et purificateurs. Nous avons constaté qu'aucune classification n'est adéquate à notre contexte. Nous avons donc proposé une nouvelle classification qui tient compte des spécificités des fichiers *class* Java. Elle distingue trois classes de techniques qui s'appliquent respectivement sur la table de constantes, le bytecode et sur l'ensemble du fichier *class* Java.

Dans la troisième et dernière section, nous avons présenté les techniques de compression des applications Java. Nous les avons qualifiées d'élémentaires puisqu'elles se présentent parfois combinées avec d'autres techniques. Par conséquent, et pour des raisons de clarté, nous avons décrit ces techniques de façon granulaire. Vers la fin de cette section, nous avons présenté des travaux qui proposent des combinaisons de ces techniques élémentaires. Il en résulte des représentations compactes alternatives au format de fichiers *class* Java.

Chapitre 3

Compression par profilage des fichiers *class* Java

3.1 Introduction

Dans ce chapitre, nous présentons notre contribution concernant la compression des applications, sous format de fichiers *class* Java, dédiées aux systèmes embarqués. L'idée est d'utiliser un profiler afin de sonder le code Java à compresser et d'en extraire les informations pertinentes qui nous permettent d'établir la stratégie de compression appropriée. Ce chapitre est composé de trois sections.

Dans la première section, la notion de profilage est introduite en mettant l'accent sur sa relation avec le domaine de l'optimisation. Nous commençons par présenter le profiler, ses modes de fonctionnement ainsi que les différentes formes de descriptions qu'il fournit. Ces descriptions portent sur les ressources et les caractéristiques d'une application donnée. Ensuite, nous décrivons d'autres formes de relations plus étroites entre le profilage et l'optimisation. Enfin, quelques exemples de travaux sont présentés où le profiler identifie les données nécessaires pour guider le processus d'optimisation.

Dans la deuxième section, nous exposons quelques observations au niveau de la performance des techniques de compression et l'influence des unes sur les autres. Ensuite, nous enchaînons par la définition d'une solution basée sur un profiler qui tient compte de ces observations. Ainsi, nous proposons un nouveau concept de profiler qui permet de sonder le code Java compilé afin d'extraire les informations pertinentes et d'établir par la suite une stratégie de compression efficace en adéquation avec le code à compresser. Ce concept est détaillé par la présentation de son architecture, ses entrées/sorties et ses modules. En entrée, le profiler prend un ensemble de classes Java et un ensemble de techniques de compression. En sortie, il fournit une stratégie de compression et le gain escompté. Quatre modules composent le profiler : un analyseur de dépendances, un scrutateur, un évaluateur de performance et un générateur de stratégies.

La troisième section de ce chapitre est consacrée à la description de notre démarche de profilage. Le processus de profilage est composé de quatre parties consacrées chacune à un module du profiler. Nous décrivons les modules sous deux angles complémentaires : fonctionnels et opérationnels. Le premier angle donne une vue sur le mode de fonctionnement d'un module. La vue interactionnelle décrit la nature des données échangées avec les autres modules ou encore les entrées/sorties du profiler.

3.2 Le profilage

L'usage des outils de profilage est inhérent à l'optimisation. En effet, ces outils contribuent d'une façon ou d'une autre au processus d'optimisation. Ils fournissent les informations pertinentes nécessaires aux développeurs pour procéder à l'optimisation. Dans la section suivante, nous décrivons le profiler et présentons des exemples de données qu'il fournit. Nous expliquons également comment ces informations peuvent contribuer à l'optimisation.

3.2.1 Rôle et fonctionnement d'un profiler

Un profiler est un système qui sonde un programme (une application) en entrée dans le but d'en dresser le profil. Ce dernier est une description des caractéristiques de l'application. Certaines informations obtenues portent totalité de la mémoire consommée et le temps d'occupation du CPU. D'autres ressources matérielles peuvent être prises en compte par le profiler, comme la consommation d'énergie. Les données fournies par le profiler peuvent être encore plus précises ; c'est à dire au delà de celles relatives uniquement à l'ensemble du programme. Le profiler permet d'englober les fonctions ou routines qui forment le programme. Avec ces informations supplémentaires, il est plus facile de localiser l'origine d'une éventuelle perte de ressource. Des profilers plus sophistiqués, comme Optimizeit[7] et JProbe[8], fournissent d'avantage d'informations. Ils renseignent par exemple sur le nombre de fois qu'une fonction est appelée ou encore sur le graphe d'appels.

Pour extraire ces données, les profilers (cf. Figure 3.1) opèrent selon deux formes bien distinctes :

Profilage à l'aide de la méthode d'échantillonnage : l'échantillonnage est une méthode de profilage dans laquelle l'observation des données est périodiquement effectuées pour déterminer la fonction active. Les données qui en résultent fournissent le nombre de fois que la fonction elle-même était au-dessus de la pile des appels au moment de l'échantillonnage du processus. Cette solution a un impact limité sur le comportement de l'application mais étant donné qu'il s'agit d'échantillonnage, il est possible qu'une fonction coûteuse passe inaperçue.

[7] Un produit Borland formé d'une suite d'outils de profilage pour les applications J2EE.

[8] Un produit de la firme Quest Software qui fournie entre autres une description détaillée de la consommation mémoire et le graphe d'appels.

Profilage à l'aide de la méthode d'instrumentation : l'instrumentation est une méthode de profilage par laquelle les versions spécialement générées des binaires profilés contiennent des fonctions de sonde qui collectent des informations de durée au niveau de l'entrée et de la sortie sur les fonctions d'un module instrumenté. Étant donné que cette méthode de profilage est plus importante que l'échantillonnage, elle entraîne une plus grande quantité de charge mémoire. Les binaires instrumentés sont également plus volumineux que les versions de débogage ou définitives et ne sont pas conçus pour le déploiement.

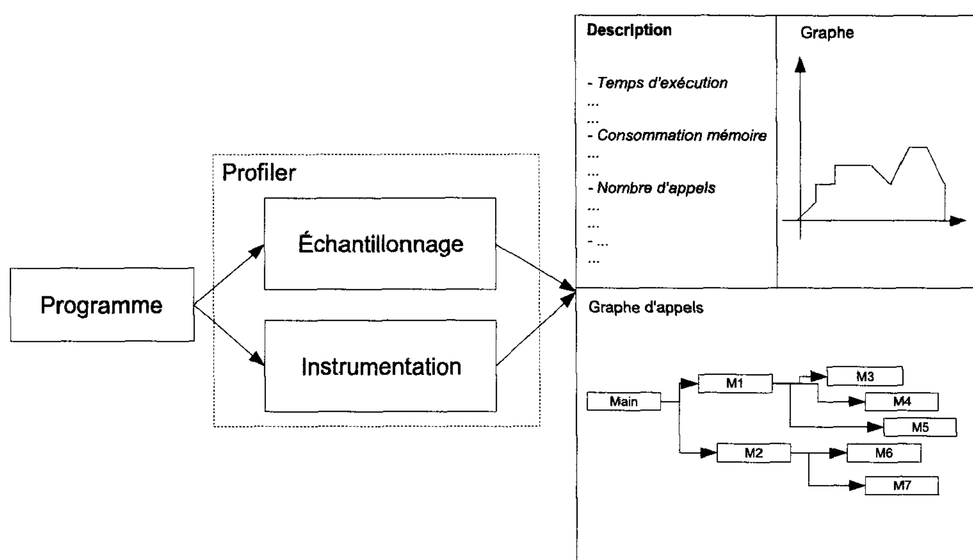

FIG. 3.1 – Profiler standard

Les données recueillies par le profiler ont pour intérêt d'attirer l'attention du développeur sur les portions de code qui contribuent le plus à l'accroissement de la consommation de ressources matérielles afin d'apporter les changements nécessaires pour optimiser le programme.

Avec une telle forme de description, la relation entre les profilers et l'optimisation reste indirecte. En effet, il est à la guise du programmeur de choisir la ou les techniques d'optimisation adéquates pour remédier aux problèmes signalés par le profiler. Cette relation est devenue de plus en plus directe grâce à des travaux de recherche qui ont réussi à impliquer d'avantage le profiler dans le processus d'optimisation. Nous consacrons la section suivante à l'étude de ces travaux.

3.2.2 Usage du profiler dans l'optimisation

Nous abordons dans cette section des cas de corrélation du profilage avec l'optimisation. Les profilers, présentés ici, sont dotés de facultés qui leurs permettent d'améliorer la performance de l'optimisation en tenant compte des caractéristiques propres au code à optimiser. Dans ce qui suit, quatre techniques d'optimisation sont présentées où chaque technique fait appel à un profiler dans l'objectif de mieux optimiser l'application.

3.2.2.1 Profiler pour la sélection des instructions ARM et Thumb

Thumb est une extension d'un chip en architecture ARM [53] qui fournit une densité améliorée du code. Cette extension permet d'enregistrer un sous-ensemble des instructions 32-bits sous forme d'instructions 16-bits compressées et les remet à la forme originale (32-bits) à l'exécution. Le code Thumb est plus compact comparé au même code écrit en ARM. Toutefois, la phase de décompression pendant l'exécution engendre un surcoût en terme de temps d'exécution. Krishnaswamy et al. [41] ont cherché à trouver un compromis entre le code ARM et Thumb. Cette recherche consiste à sélectionner la partie de code à garder en ARM et celle à transformer en Thumb. Le but est donc d'obtenir un code le plus réduit possible avec un moindre impact sur la performance. Dans ce sens, des algorithmes guidés par un profiler sont proposés pour générer un code mixte ARM et Thumb. Le profiler compare le code relatif à une fonction de façon plus ou moins granulaire sous les deux formes : ARM et Thumb. Quatre heuristiques ont été proposées : (*i*) moindre nombre de cycles, (*ii*) moindre nombre d'instructions, (*iii*) moindre taille de code supérieur à un seuil inférieur, et (*iv*) moindre taille et instructions avec deux seuils respectifs.

3.2.2.2 Profiler pour optimiser les changements de contexte

L'optimisation du changement de contexte est une technique qui permet de réduire les appels fréquents entre processus. Elle consiste à mettre en ligne les fonctions afin de limite les changements de contexte pénalisant en terme de temps d'exécution. Dans ce cadre, Johansson et al. [32] font intervenir un profiler pour collecter les informations concernant le récepteur de chaque instruction "d'envoi de message". Les informations collectées concernent deux éléments : le destinataire et le nombre de fois que l'instruction a été exécutée. Tout couple de processus expéditeur/récepteur est candidat à l'optimisation. Une fois une sélection de couple de processus est établie, le compilateur procède

à l'optimisation en remplaçant l'appel du processus expéditeur par le code du processus destinataire.

3.2.2.3 Profiler pour la compression sélective

La plupart du temps d'exécution est occupé par une petite portion du code [37]. Ce qui signifie que la majorité du code est généralement rarement exécuté. La compression sélective consiste à compresser la partie de code non fréquemment exécutée et à garder le reste du code à l'état initiale. Ceci engendre une réduction considérable de la taille du code avec une atténuation de la performance totale du programme due à la décompression. Debray [19] et Heydemann [31] proposent d'utiliser un profiler pour identifier les portions de code à compresser de façon à garantir un compromis entre le taux de compression et le temps d'exécution.

3.2.2.4 Profiler pour compilation sélective

Un code compilé est rapide à exécuter mais sa taille est supérieure au même code interprété. La compression partielle d'un programme - rendue possible grâce aux compilateurs AOT (Ahead-Of-Time) - permet de gagner en espace mémoire. Colin de Verdiere [59] propose un schéma de compilation pour les compilateurs AOT qui consiste en la sélection des parties de code à compiler. La sélection est basée sur un profiler dont rôle est d'enregistrer les temps d'exécution et la fréquence d'appel de chaque méthode. Le but de ce schéma de sélection est de produire un code plus rapide.

3.2.2.5 Conclusion sur l'usage du profiler dans l'optimisation

Pour une meilleure performance de l'optimisation, on peut faire appel au profiler. Son rôle est de récolter les informations pertinentes pour guider le processus d'optimisation. La différence réside dans la définition des données à collecter par le profiler et la façon de s'en servir pour contribuer à l'optimisation. Notons que chacun de ces profilers est dédié à une seule technique d'optimisation. Cette restriction au niveau des techniques impliquées dans l'optimisation limite le gain en performance escompté. Dans ce sens, l'approche de profilage que nous proposons tiendra compte de plusieurs techniques de compression à la fois. Néanmoins, certains problèmes surgissent lors de la co-application d'un ensemble de techniques de compression. Nous décrivons dans la section suivante notre profiler dédié à

la compression de fichiers *class* Java qui gère un ensemble de techniques de compression pour en tirer le meilleur gain possible.

3.3 Profiler dédié à la compression

Nous abordons dans la présente section la partie principale de notre travail. Il s'agit d'un nouveau concept de profiler pour guider le processus de compression des fichiers *class* Java. Avant de découvrir l'architecture de ce profiler, nous situons notre démarche dans son cadre général.

3.3.1 Cadre général de la démarche proposée

L'application des techniques de compression sur des programmes Java, révèlent des caractéristiques relatives à leurs performances. Ces caractéristiques concernent trois situations :

- **cas d'inflation** : la performance d'une technique de compression varie d'un programme en entrée à un autre. Par exemple, le taux de compression d'une technique donnée peut décroître au point d'engendrer l'inverse de l'effet escompté. Ainsi, le fichier résultant de l'application de la technique de compression en question peut être d'une taille plus importante que le fichier original. Dans ce cas de figure on parle d'inflation. Les techniques de compression par dictionnaire, par exemple, ne sont efficaces que lorsque les entrées ajoutées au dictionnaire sont assez fréquentes dans le fichier original ;
- **cas d'incompatibilité** : il existe une sorte d'incompatibilité entre deux ou plusieurs techniques de compression différentes. Une fois appliquées, certaines techniques ne permettent pas l'application d'autres. La nature du fichier *class* Java fait qu'il peut ne pas supporter la co-application de plusieurs techniques. Généralement, une technique de compression apporte des modifications sur un ou plusieurs composants de la structure du fichier *class* Java. Une deuxième technique, ne trouvant pas ce composant dans son état original, est donc inhibée et par conséquence, son usage est impossible ;
- **cas de dépendance** : le taux de compression relatif à une technique donnée, appliquée à un programme en entrée, peut dépendre des techniques qui le précèdent.

En effet, l'application d'une technique de compression donnée peut soit atténuer soit amplifier la performance (taux de compression) d'une deuxième technique qui suit la précédente. Ceci s'explique par le fait que la première technique apporte des modifications sur le programme en entrée qui sont favorables ou défavorables pour la deuxième technique. Le changement de structure des descripteurs de méthodes ajoute des entrées de type Utf8 à la table de constantes pour représenter le ou les paramètre(s). Lorsque ces entrées sont similaires à d'autres déjà existantes dans la table de constantes, de nouvelles redondances sont donc créées. Une technique quelconque de réduction de redondances devient par conséquence plus efficace.

Ceci montre que la mise à contribution de plusieurs techniques pour établir une méthodologie globale de compression nécessite un soin particulier. Nous constatons également que la performance d'une telle méthodologie de compression dépend essentiellement de deux facteurs primordiaux : les fichiers *class* Java à compacter et la façon dont nous utilisons et appliquons les techniques de compression.

La mise en œuvre d'une stratégie de compression des fichiers *class* Java nécessite la détermination des techniques à utiliser ainsi que leur ordre d'application. Cette tâche est particulièrement difficile étant donné le large choix de techniques dont on dispose et les dépendances révélées entre techniques de compression. La prise en compte des caractéristiques intrinsèques au code Java à compresser ne fait que compliquer sa tâche.

Afin de surmonter cette difficulté, nous proposons de mettre en place un système permettant d'établir un schéma de compression performant. Ce système fait appel à un profiler dédié pour guider le processus de compression. L'architecture de ce profiler est présentée dans la section suivante.

3.3.2 Architecture du profiler

Cette section présente l'architecture globale de notre profiler dédié à la compression [3]. Elle décrit ainsi ses différentes interactions avec l'extérieur (Entrées/Sorties) ainsi que les modules qui le composent.

3.3.2.1 Les Entrées/Sorties du profiler

L'interaction avec le profiler est assurée grâce à deux entrées et une sortie. La première entrée est l'application Java sous forme d'un ensemble de fichiers *class* Java. Notons que

l'ensemble de fichiers peut ne comporter qu'un seul fichier *class* Java. Cette entrée représente évidemment l'application sujette à la compression. La deuxième entrée représente un ensemble prédéfini de techniques élémentaires de compression comme celles décrites dans le chapitre 2 section 2.4. Ces techniques sont candidates pour figurer dans la stratégie globale de compression. Nous n'imposons aucune contrainte sur la nature ou la performance des techniques en entrée au profiler. Par ailleurs, le fait d'exiger des techniques 'élémentaires' ne constitue pas une contrainte. En effet, les sélections de techniques de compression sont exclues puisque c'est, entre autres, le but recherché par le profiler.

Le profiler fournit en sortie une ou plusieurs stratégies de compression ainsi que le gain correspondant. Il est à noter que le terme 'stratégie' est utilisé pour désigner une séquence de techniques de compression. Nous notons également que le gain, dont il est question dans ce chapitre, représente la différence entre la taille de la forme originale et la forme compressée du code Java compilé. La sortie permet, à la fois, de déterminer :

- les techniques à appliquer sur les fichiers *class* Java, données en entrée du profiler ; c'est à dire celles qui contribuent le mieux à la compaction des fichiers *class* en question. Chaque technique ne s'applique qu'une seule fois dans la sélection ;
- l'ordre d'application des techniques de compression. Le profiler détermine le séquencement qui produit le meilleur taux de compression. Notons que l'ordre peut ne concerner qu'un sous ensemble de techniques et non la totalité de celles données en entrée.

Le traitement des entrées et la génération de la sortie sont assurés par différents modules qui représentent le noyau du profiler.

3.3.2.2 Les modules du profiler

Nous définissons dans la présente section les différents modules composant le profiler ainsi que leurs rôles respectifs. Nous verrons plus loin d'avantage de détails sur leurs principes de fonctionnement. Le noyau du profiler (cf. Figure 3.2) est composé d'un scrutateur, d'un analyseur de dépendances, d'un évaluateur de performance et d'un générateur de stratégie. Les rôles de ces modules sont les suivants :

1. **Scrutateur** : il est chargé de l'examen des fichiers *class* Java à compresser. Le scrutateur peut être considéré comme un profiler standard. En effet, il fournit une description du fichier à compresser. Cependant, les données incluses dans la des-

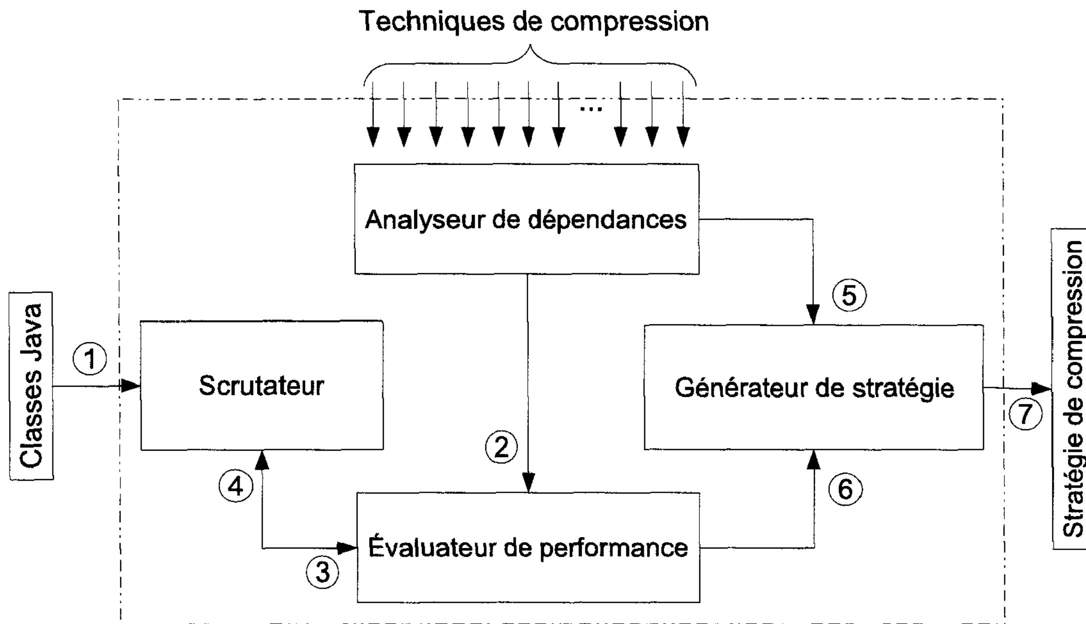

FIG. 3.2 – Architecture du profiler dédié à la compression

cription ne concernent pas, comme c'est souvent le cas, la consommation mémoire ou le nombre d'appels des méthodes. Il s'agit plutôt d'informations requises par le profiler afin d'évaluer les performances des techniques de compression ;

2. **Analyseur de dépendances** : il permet de définir les relations de dépendances qui peuvent exister entre les techniques de compression. La dépendance dont il est question dans notre cas repose sur l'effet que peut avoir une technique de compression sur la performance d'une ou plusieurs autres techniques ;

3. **Évaluateur de performance** : chaque technique de compression contribue à la réduction de l'espace mémoire requis pour représenter une application Java. Cette contribution dépend entre autres de l'entrée Java sur laquelle s'applique la technique. Il est donc nécessaire de quantifier la 'valeur ajoutée' d'une technique, facteur décisif dans l'établissement de la stratégie de compression. Et c'est en effet le rôle accompli par le module évaluateur de performance ;

4. **Générateur de stratégie** : il représente le dernier maillon de la chaîne formant le profiler. Au vu des données fournies par les autres modules, le générateur de stratégie établit la ou les séquences de techniques de compression qui contribuent le mieux à une compression efficace.

Nous découvrons dans la section suivante le principe de fonctionnement de chaque module à travers la description de la démarche de profilage.

3.4 Principe de fonctionnement du profiler

Pour profiler une application Java en vu de déterminer le schéma de compression optimal, le profiler fait appel à ses quatre modules. Nous présentons dans la suite son principe de fonctionnement.

3.4.1 Analyse de dépendances des techniques de compression

Les caractéristiques relatives aux interactions possibles entre les techniques de compression (cf. section 3.3.1) nous ont conduit à établir les relations de dépendances entre ces techniques au niveau de la performance. Ainsi, nous définissons trois types de dépendances : indépendance, dépendance totale et dépendance partielle. Nous les définissons comme suit (avec Ti et Tj deux techniques de compression avec Gi et Gj leurs gains respectifs) :

Définition 1.

Ti et Tj sont considérées comme totalement indépendantes si leurs gains restent inchangés après leur co-application. Autrement dit, le gain total de Ti et Tj est égal à la somme de Gi et Gj. Nous illustrons cette définition par la notation suivante : $Ti \dashrightarrow Tj$ ou $Tj \dashrightarrow Ti$.

Définition 2.

Tj est considérée comme totalement dépendante de Ti si son application n'est pas possible après l'application de Ti. La dépendance provient du fait que l'application de Ti entre en conflit avec Tj. La relation qui lie Tj à Ti est notée comme suit : $Ti \nrightarrow Tj$.

Définition 3.

Tj est considérée comme partiellement dépendante de Ti si son application est possible après Ti, et si la performance de Tj varie lorsqu'elle est appliquée après Ti. Autrement dit, le gain total engendré par l'application de Ti suivie par Tj est inférieur à la somme de Gi et Gj. La dépendance partielle de Tj à Ti est notée : $Ti \leadsto Tj$.

L'analyseur de dépendances dispose en entrée d'un ensemble de techniques de compression orientées vers les applications Java. L'analyse effectuée par l'analyseur est guidée par ces trois types de relations. L'analyse de dépendance est effectuée pour chaque technique par rapport au reste des techniques de l'ensemble de départ. Le profiler identifie un type de dépendance parmi les trois types possibles : indépendance, dépendance totale et dépendance partielle. Nous distinguons deux méthodes d'analyse :

Méthode numérique : cette analyse prend son origine dans la définition même de la notion de dépendance. Pour identifier le type de dépendance d'une technique quelconque T_i par rapport à une technique T_k, le module d'analyse de dépendance fait appel au module d'estimation de performance. Ce dernier lui renvoi :

- les gains G_i et G_k résultant de l'application séparée de T_i et T_k respectivement ;
- le gain G_{ki} résultant de l'application de T_k suivie de T_i. Si un problème d'incompatibilité surgit lors de l'estimation, une erreur est retournée au lieu de la valeur du gain.

Au vu des gains retournés par le module d'estimation de performance, le module analyseur de dépendances détermine la relation qui lie T_i à T_k selon les trois cas de figures suivants :

- si $G_{ki} = \text{ERREUR} \Longrightarrow T_k \nrightarrow T_i$;
- si $G_{ki} = G_i + G_k \Longrightarrow T_k \dashrightarrow T_i$;
- si $G_{ki} \neq G_i + G_k \Longrightarrow T_k \rightsquigarrow T_i$.

Méthode analytique : cette analyse est basée sur le type de traitement effectué par les techniques. Une technique de compression donnée opère sur des éléments de la structure de fichier *class* Java. La dépendance d'une technique T_i par rapport à une deuxième technique T_k n'a lieu que lorsque T_i et T_k opèrent sur des éléments communs. De ce fait, l'analyse de la dépendance revient d'abord à déterminer l'existence ou non d'éléments en commun entre T_i et T_k.

- si oui, alors T_i est dépendante de T_k. Il reste à déterminer s'il s'agit de dépendance totale ou partielle.
- si non, alors $T_i \dashrightarrow T_k$;

L'analyse des types de traitements possibles effectués par les techniques de compression sur les éléments du fichier *class* Java montre que ces traitements renseignent sur la nature de la dépendance (totale ou partielle) comme le montre le tableau 3.1.

Dépendance totale	**Dépendance partielle**
Modification structurelle	Modification du contenu
Suppression des info. de débogage	Augmentation des occurrences
Changement d'encodage	Réduction des occurrences

TAB. 3.1 – Répartition des types de traitements sur les dépendances

Le tableau 3.1 montre certains types de traitements répartis entre les deux types de dépendances possibles. Le type de traitement renseigne sur la nature de la dépendance engendrée (partielle ou totale). Notons que la méthode analytique requiert une connaissance préalable du principe de fonctionnement des techniques de compression, des éléments concernés et de la nature des traitements.

Le résultat de cette analyse est résumé dans un tableau, dit tableau de dépendances. Pour une interprétation explicite du contenu de ce tableau, nous donnons l'exemple suivant :

Soit $T = \{T1, T2, T3, T4, T5, T5, T6, T7\}$ un ensemble de techniques de compression applicables sur les fichiers *class* Java dont les dépendances sont précisées sur le tableau 3.2.

$\looparrowright$	T1	T2	T3	T4	T5	T6	T7
T1				$\leadsto$		$\leadsto$	
T2							
T3							$\nrightarrow$
T4	$\leadsto$					$\leadsto$	
T5							
T6	$\leadsto$			$\nrightarrow$			
T7			$\nrightarrow$				

TAB. 3.2 – Dépendances des techniques de compression

Le tableau 3.2 résume les dépendances décelées par le module analyseur de dépendances avec comme entrée l'ensemble T. Notons que la relation d'indépendance n'est pas représentée dans le tableau pour des raisons de clarté. Les techniques sont placées sur la première ligne et la première colonne du tableau. Chaque cellule contient le symbole (notation) qui représente la dépendance de la technique se trouvant sur la même ligne par rapport à la technique de la même colonne. La cellule située à l'intersection de la $4^{\text{ième}}$ ligne et la $6^{\text{ième}}$ colonne indique que la technique $T6$ est partiellement dépendante de $T4$.

Nous définissons une deuxième représentation que nous appelons graphe de dépendances. Ce graphe illustre mieux les relations entre les techniques. Les sommets du graphe représentent les techniques de compression et les arcs les dépendances entre les techniques relatives aux définitions 2 et 3. La transcription du tableau 3.2 sous forme de graphe de dépendances est visible sur la figure 3.3.

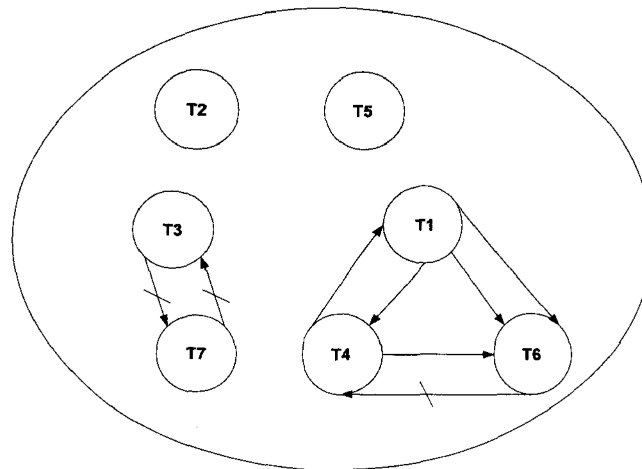

FIG. 3.3 – Graphe de dépendances des techniques de l'ensemble T

Cette représentation permet de distinguer des éventuels groupes de techniques ne présentant que des liens entre eux. Par exemple, les techniques $T3$ et $T7$ forment un groupe de techniques totalement interdépendantes. Nous pouvons discerner un deuxième groupe formé par les techniques $T1$, $T4$ et $T6$ liées par des relations de dépendance partielle et totale (respectivement identifiées sur le graphe par une flèche et une flèche barrée).

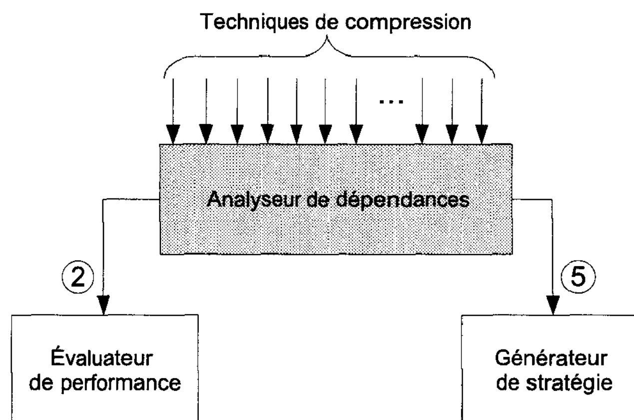

FIG. 3.4 – Le module analyseur de dépendances

Le module analyseur de dépendance transmet le tableau de dépendances vers le module évaluateur de performance et vers le module générateur de stratégie (respectivement les relations 1 et 2 de la figure 3.4). Chaque module récepteur exploite ce tableau de façon à assurer les fonctionnalités qui lui sont attribuées. L'exploitation du tableau de dépendances sera présentée plus loin dans les sections correspondantes aux modules récepteurs (cf. section 3.4.3 pour le module évaluateur de performance et la section 3.4.4 pour le module générateur de stratégies.)

3.4.2 Scrutation des fichiers *class* Java

Cette section décrit le module scrutateur et son mode de fonctionnement en interaction avec les autres modules du profiler. Ce module permet de caractériser l'entrée Java en identifiant les données nécessaire pour établir, par la suite, une stratégie de compression efficace. En d'autres termes, il définit la faculté du profiler à adapter la stratégie de compression aux caractéristiques intrinsèques au code Java à compresser.

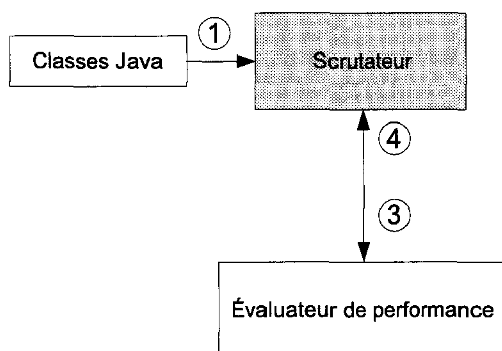

FIG. 3.5 – Le module scutateur

La figure 3.5 représente le module scrutateur en interaction avec le module évaluateur de performance et les fichiers *class* Java. Le module de scrutation examine les fichiers Java (relation 1) en entrée sous format *class*, afin d'en extraire les informations utiles pour guider le processus de compression. Ces informations sont déterminées par le module évaluateur de performance qui élabore, selon ses besoins, la liste des éléments à scruter et les communique au module scrutateur (relation 4). Le résultat de la scrutation est transmis (relation 3) au module d'évaluation de performance afin d'évaluer la performance des techniques de compression. La section 3.4.3 décrit les interactions entre le scrutateur

et l'évaluateur de performance. Les données scrutées sont relatives à :

- l'ensemble des classes Java : elles sont représentées par une fiche unique commune à toutes les classes. Nous retrouvons dans cette fiche des informations nécessaires au calcul du taux de compression, comme par exemple le nombre et la taille totale des classes.
- chaque classe Java : le scrutateur collecte les informations relatives à chaque classe Java candidates à la compression. L'ensemble des classes étant organisé en packages, nous adoptons une structure arborescente pour représenter les fiches relatives à chaque classe : une fiche par classe. Comme exemple d'informations contenues dans ces fiches, nous citons la taille de la classe, la taille de ses composants ou encore le nombre de fois qu'elle est appelée par les autres classes du package.

Notons que c'est le module d'évaluation de performance qui indique au module de scrutation quelles informations doit il extraire des fichiers à scruter (relation 4 de la figure 3.4).

3.4.3 Module d'évaluation de performance

Rappelons que le module évaluation de performance a pour rôle d'estimer le gain engendré par l'application d'une ou plusieurs techniques de compression. Comme son nom l'indique, en aucun cas nous n'appliquons une quelconque technique sur le code donné en entrée pour déterminer le gain qui en résulte. D'ailleurs, le module en question n'interagit point avec l'entrée Java (cf. figure 3.6). Il s'agit uniquement d'une évaluation statique.

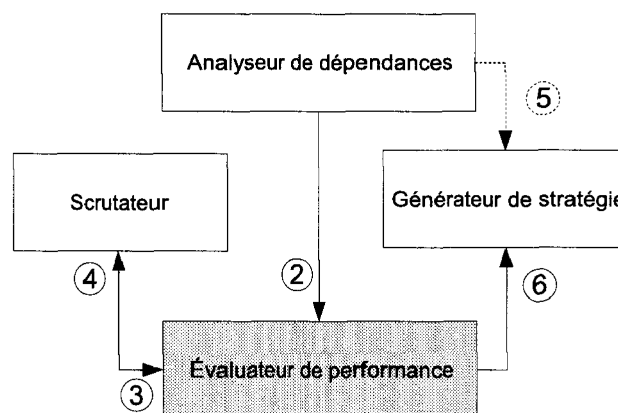

FIG. 3.6 – Le module évaluateur de performance

Pour ce faire, il est nécessaire de disposer d'une connaissance préalable sur le principe de fonctionnement de chaque technique de compression pour pouvoir en évaluer le gain. D'une part, ceci nous permet de déterminer les informations à extraire pour pouvoir calculer le gain d'une technique. Ces informations concernent des éléments contenus dans l'entrée Java sur lesquelles agit la technique en question. D'autre part, cette connaissance nous permet de déterminer le gain engendré. Selon la technique à évaluer, nous établissons la formule ou l'algorithme de calcul de son bénéfice.

Nous distinguons deux types d'évaluation de performance. Le premier type traite une technique de compression sans tenir compte des autres techniques. Autrement dit, l'évaluation porte sur le gain de la technique en question comme si elle était la seule à être appliquée. Le second type sert à évaluer le gain d'une technique en tenant compte de l'influence des autres techniques dont elle dépend partiellement. Les deux sections suivantes présentent, dans l'ordre, ces deux types d'évaluation.

3.4.3.1 Évaluation séparée d'une technique de compression

L'évaluation séparée d'une technique de compression nécessite l'élaboration préalable de la formule de calcul de gain ensuite, l'exécution des étapes suivantes :

1. Créer une fiche FT_i qui contient les éléments intervenant dans le calcul du gain de la technique T_i à évaluer ;
2. Envoyer la fiche FT_i vers le scrutateur qui la remplie en affectant à chaque élément de la fiche la valeur qui lui correspond ;
3. Récupérer du scrutateur la fiche FT_i. De retour à l'évaluateur de performance, le contenu de la fiche FT_i est exploité pour évaluer le gain engendré par la technique en question.

Le gain retourné n'est valable que lorsque la technique est la seule ou la première à être appliquée. En effet, si elle est appliquée à la suite d'une technique dont elle dépend partiellement, le gain est affecté. Dans ce genre de cas, nous faisons appel à l'évaluation dépendante présentée dans la section suivante.

3.4.3.2 Évaluation dépendante d'une technique de compression

L'évaluation dépendante tient compte du contexte dans lequel se trouve la technique à évaluer. Le changement de contexte est traduit au niveau de la valeur des éléments conte-

nus dans sa fiche et intervenant dans le calcul de son gain. En effet, les valeurs initialement obtenues par le scrutateur (contenues dans la fiche FT_i) ne sont plus valables lorsque la technique à évaluer est précédée par d'autres techniques dont elle dépend partiellement. Nous devons donc réévaluer le contenue de la fiche FT_i pour tenir compte de l'influence des autres techniques.

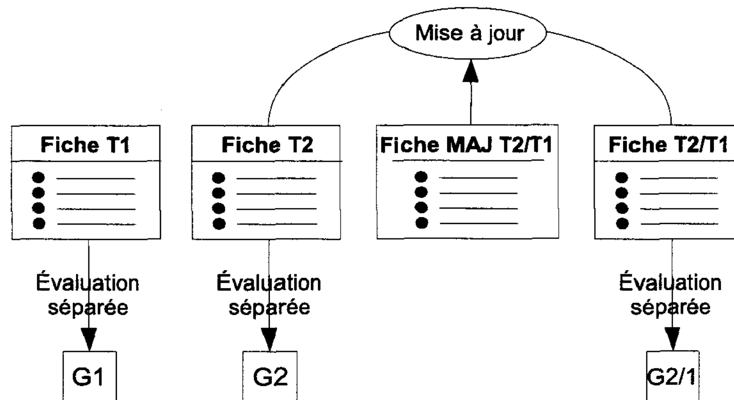

FIG. 3.7 – Exploitation des fiches pour l'évaluation de performance

Nous proposons de mettre à jour les fiches des autres techniques de compression qui sont partiellement dépendantes de la technique en question. Supposons que l'on dispose de deux techniques $T1$ et $T2$ dont les gains respectifs sont $G1$ et $G2$ et tel que $T1 \leadsto T2$. Nous faisons donc appel à l'évaluation dépendante pour calculer le gain de $T2$ lorsqu'elle est appliquée après $T1$ (cf. figure 3.7). Pour effectuer la mise à jour, nous introduisons une nouvelle fiche $FT2/1$ qui contient les éléments nécessaires pour réévaluer les éléments de la fiche $FT2$ (cf. figure 3.7). Il ne reste plus qu'à utiliser cette fiche ($FT2/1$) pour retrouver le $G2/1$ correspondant au seul gain de $T2$ appliquée à la suite de $T1$. Le calcul de $G2/1$ à partir de $FT2/1$ se déroule comme dans le cas d'une évaluation séparée.

Le passage par une étape de mise à jour des fiches permet de tenir compte des dépendances entre techniques de compression. Mais une fois la réévaluation des fiches effectuée, la suite du processus se déroule comme dans le cas d'une évaluation séparée. De ce fait, nous n'avons pas à nous soucier des techniques déjà appliquées. En plus, nous maintenons exploitable la formule de calcul de gain.

3.4.4 Génération de stratégies de compression

Le module générateur de stratégies est le "dernier maillon" du profiler. Il est en interaction directe avec deux autres modules, à savoir l'analyseur de dépendances et l'évaluateur de performance. Il exploite les données fournies par ces deux modules pour élaborer les stratégies de compression qui représentent la sortie du profiler (cf. figure 3.8).

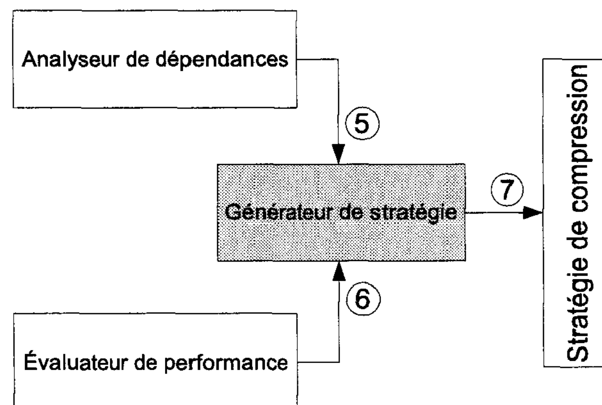

FIG. 3.8 – Le module générateur de stratégies

De son côté, le module analyseur de dépendances transmet au générateur de stratégies le graphe de dépendances (relation 5). Quant au module évaluateur de performance, il lui fournit les valeurs correspondant aux gains engendrés par chaque technique de compression appliquée séparément et en combinaison avec d'autres techniques (relation 6). À partir de ces données, le générateur de stratégies est censé donner en sortie (relation 7) l'ensemble de techniques de compression susceptible de fournir le meilleur gain possible. En plus, il doit déterminer un ordre de séquencement engendré par les éventuelles dépendances qui peuvent exister entre les techniques de compression au niveau de leurs performances.

Il existe une similarité entre notre problème de recherche de séquence de techniques de compression offrant le meilleur gain et le problème de recherche du plus long chemin dans un graphe orienté. En effet, le graphe de dépendances provenant du module analyseur de performance se présentent comme un espace formé d'un nuage de points (techniques de compression) interconnectés selon une orientation bien définie (ordre d'application dans le cas de techniques dépendantes). Par ailleurs, le module évaluateur de performance permet de placer des poids sur les liens (gain relatif à une technique ou une suite de techniques). Ces données permettent donc de former un graphe orienté valué dans lequel s'effectue la

recherche du plus long chemin (meilleur gain d'espace mémoire possible).

Nous adoptons donc cette solution issue du domaine de l'optimisation combinatoire pour trouver une solution au problème de recherche de stratégies. Cependant, nous devons définir l'espace de recherche et établir la formulation qui permettent sa résolution comme étant un programme linéaire.

3.4.4.1 Construction de l'espace de recherche de stratégies

La construction de l'espace de recherche consiste à spécifier les sommets du graphe, les arcs entre les sommets et les poids sur les arcs. Le générateur de stratégies récupère le graphe de dépendances qui contient l'ensemble des techniques de compression incluses dans le profiler. Les techniques indépendantes sont exclues de l'ensemble T puisque, par définition, elles n'ont pas un lien de dépendance basé sur le gain avec le reste du groupe. Cependant, elles sont rajoutées ultérieurement à la stratégie optimale de compression si leur gain s'avère positif. Le reste des techniques forment les nœuds du graphe. Nous ajoutons par la suite les connections entre les sommets. Chaque couple de sommets (i, j) est relié par un arc allant de i vers j et inversement. Nous obtenons alors un graphe orienté complet (cf. figure 3.9).

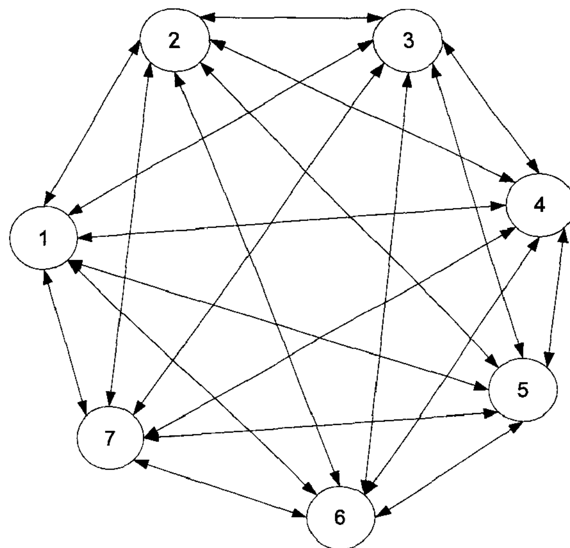

FIG. 3.9 – Graphe de recherche de stratégies

La figure 3.9 montre un exemple de graphe formé par un ensemble de sept sommets (indice de 1 à 7). Chaque sommet représente une technique de compression. Par ailleurs, les arcs entre les nœuds indiquent l'ordre de séquencement des techniques. Notons qu'il existe plusieurs chemins possibles pour explorer le graphe et par conséquence autant de séquences de techniques. L'objectif est de trouver parmi toutes les séquences possibles, celle qui réduit le mieux la taille du code Java. Nous présentons dans la section suivante la modélisation du problème de recherche de stratégies de compression.

3.4.4.2 Formalisation du problème de recherche de stratégies

Nous donnons dans ce qui suit une formulation qui nous permet de résoudre le problème de recherche de stratégies. Sa définition consiste à établir la fonction objectif et les contraintes à respecter. Commençons par donner quelques notations :

- $T = \{T_1, \ldots, T_n\}$ l'ensemble des techniques de compression incluses dans le profiler ;
- $s = < T_i, \ldots, T_k >$ une *séquence* de techniques ($T_i, \ldots, T_k \in T$) définie comme étant une liste de techniques telle que :
  - une technique est appliquée une et une seule fois ;
  - la liste des techniques est ordonnée ;
  - toutes les techniques de T ne sont pas nécessairement incluses dans la liste ;
- S l'ensemble des séquences de techniques ;
- $G(T_j|T_i, \ldots, T_k)$ le gain d'une technique T_j précédée par la séquence de techniques $< T_i, \ldots, T_k >$. Par convention, si T_j est totalement dépendante de T_i, on pose $G(T_j|\sigma(I)) = -\infty$ pour toute permutation σ des techniques de $I \subseteq T - \{T_j\}$. Nous avons choisi une valeur fortement négative pour exclure toute séquence contenant T_i précédée de T_j ;
- $\varphi(s)$ la fonction d'évaluation d'une séquence s de l'ensemble S. Pour une séquence $s = < T_1, \ldots, T_k >$, $\varphi(s)$ est définie comme suit :

$$\varphi(s) = G(T_1) + G(T_2|T_1) + \cdots + G(T_k|T_1, \ldots, T_{k-1})$$

La recherche de la meilleure stratégie de compression revient à trouver parmi toutes les séquences de techniques de compression possibles, celle qui offre le meilleur gain. Selon les notations ci-dessus, nous définissons alors la fonction objectif donnée par l'équation 3.1.

$$\underset{s \in S}{Max} \quad \varphi(s) \tag{3.1}$$

Pour décrire la taille de ce problème, nous donnons le nombre de séquences possibles et son évolution en fonction du nombre de sommets du graphe. Lorsque tous les sommets représentent des techniques partiellement dépendantes les unes des autres, le nombre de séquences possibles est la $\sum_{p=0}^{n} A_n^p$, où n est le nombre de sommets du graphe et A_n^p est l'arrangement de p éléments parmi n. Dans le cas où $n = 7$, le nombre de séquences est égal à 13700. Lorsque la valeur de n passe de 7 à 8, le nombre de séquences possibles augmente considérablement pour atteindre 109601. Le problème est NP-Complet. Étant donnée l'explosion combinatoire, il semble plus judicieux d'utiliser des méthodes approchées pour résoudre le problème. Ainsi, nous proposons d'utiliser des méthodes gloutonnes dont le principe est de construire la solution incrémentalement en rajoutant à chaque pas un élément selon un critère glouton, i.e. celui qui nous paraît localement le meilleur choix à "court terme". Si cette vision à court terme nous donne toujours une solution optimale, on parle alors d'algorithme glouton exact sinon d'heuristique gloutonne. Dans le cas de la recherche de la meilleure séquence de compression, notre critère de choix à chaque étape est déterminé par le gain en terme d'espace mémoire. Nous proposons dans la suite deux heuristiques gloutonnes qui diffèrent au niveau de la prise en compte du gain :

1. **Heuristique gloutonne basée sur le gain normal** : à chaque itération de la construction de solution, une technique, parmi celles possibles, est ajoutée à la séquence. L'approche peut être comparée à la recherche du plus proche voisin dans le cas du problème du voyageur de commerce [46]. En effet, la technique sélectionnée est celle qui offre le meilleur gain ; jusqu'à ce qu'il ne reste plus de techniques dans l'ensemble initial (N). Cette heuristique est définie par les quatre étapes suivantes :
   - **Étape 1** : initialiser l'arbre et considérer la racine comme le nœud courant ;
   - **Étape 2** : comparer les gains des nœuds sous-jacents au nœud courant ;
   - **Étape 3** : pointer le nœud courant sur le nœud correspondant au gain maximum ;
   - **Étape 4** : tant qu'il y a des nœuds à explorer, revenir à l'étape 2 ; sinon sortir.

Soient N l'ensemble initial de sommets (techniques de compression), Π la stratégie à chercher et C_{ij} la distance (gain) de i vers j. L'algorithme est décrit comme suit :

Algorithme 1 : Algorithme de la 1ere heuristique gloutonne

$\Pi =< 0 >$,i=0

$N = \{T_1, T_2, \ldots, T_n\}$

tantque $\Pi \neq \phi$ **faire**

$C_{ij^*} \leftarrow \max\{C_{ij} : j \in N - \Pi\}$

$\Pi \leftarrow \Pi + < j^* >$

$N \leftarrow N - < j^* >$

$i \leftarrow j^*$

fin tantque

La figure 3.10 illustre cette première heuristique gloutonne appliquée sur un ensemble de trois techniques $\{T1, T2, T3\}$. Le nœud racine O représente les fichiers Java à compresser à l'état initial. La solution (nœuds marqués par un fond gris) est construite par l'ajout de T1, ensuite T2 et enfin T3. Notons que cette heuristique permet d'élaguer une grande partie de l'espace d'exploration (espace hachuré de la figure 3.10). Dans notre exemple, 9 nœuds sur 15 n'ont pas été explorés.

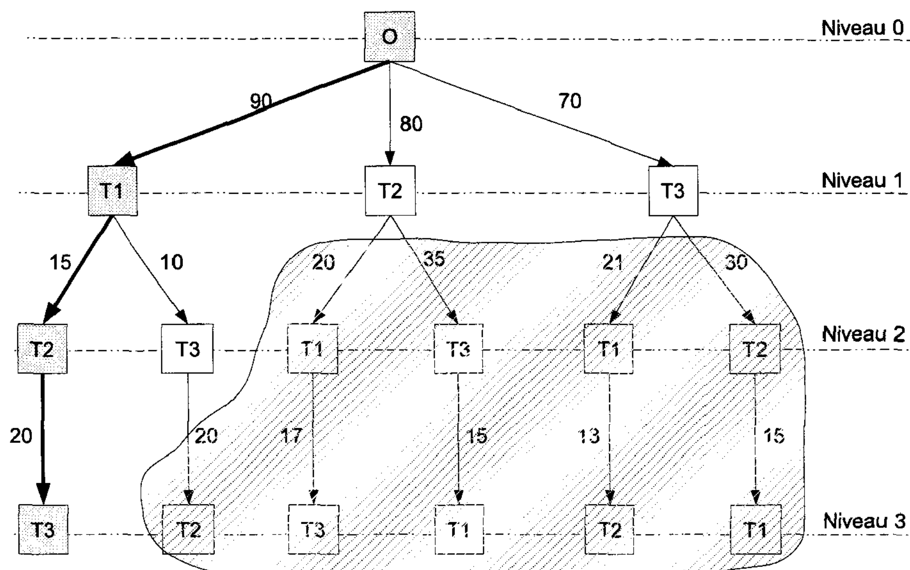

FIG. 3.10 – Heuristique gloutonne basée sur le gain normal

2. **Heuristique gloutonne basée sur le gain biaisé** : le gain biaisé consiste à considérer les gains au niveau suivant (niveau $i + 1$) pour décider du choix de la

technique à ajouter à la séquence au niveau courant (niveau i). Cette heuristique diffère de la première au niveau de la deuxième étape qui sera substituée par l'**étape 2'** suivante :

– **Étape 2'** : comparer les gains biaisés des nœuds sous-jacents au nœud courant ;

Ainsi, l'algorithme peut être décrit comme suit :

Algorithme 2 : Algorithme de la 2^{ieme} heuristique gloutonne

$\Pi =< 0 >$,i=0

$N = \{T_1, T_2, \ldots, T_n\}$

tantque $|\Pi| \neq 1$ **faire**

$\delta_{ij^*} \leftarrow \max\{\delta_{ij} : j \in N - \Pi\}$

$\Pi \leftarrow \Pi + < j^* >$

$N \leftarrow N - < j^* >$

$i \leftarrow j^*$

fin tantque

Où $\delta_{ij^*} = \max\{C(\Pi) + C_{jk} : k \in N - \Pi - \{j\}\}$.

Sur l'exemple de la figure 3.11, le calcul du gain biaisé de T1 (niveau 1) est donné par la formule suivante :

$$\tilde{G}(T_1) = \underset{k \neq 2}{MAX} \quad G(T_k|T_1) + G(T_1) = \underset{k \neq 3}{MAX} \quad (90 + 15, 90 + 10) = 105$$

Cette anticipation dans la recherche permet de détecter de nouvelles séquences avec un meilleur gain. En effet, la meilleur séquence (marquée avec des nœuds gris sur la figure 3.11) trouvée par cette heuristique est $< T2, T3, T1 >$ avec un gain de 130. Alors que le gain de la solution donnée par la première heuristique est égal à 125.

Comparée à la première méthode, l'heuristique gloutonne basée sur le gain biaisé explore plus de nœuds (seulement 5 nœuds ont été élagués contre 15 pour la première heuristique). Ceci explique le fait que la solution obtenue par cette méthode est meilleure que celle basée sur le gain normal ; avec un surcoût de temps de recherche.

De façon générale, la recherche de la solution avec le premier algorithme nécessite l'exploration d'un nombre de nœuds calculés comme suit :

$$n + (n-1) + (n-2) + \cdots + 1$$

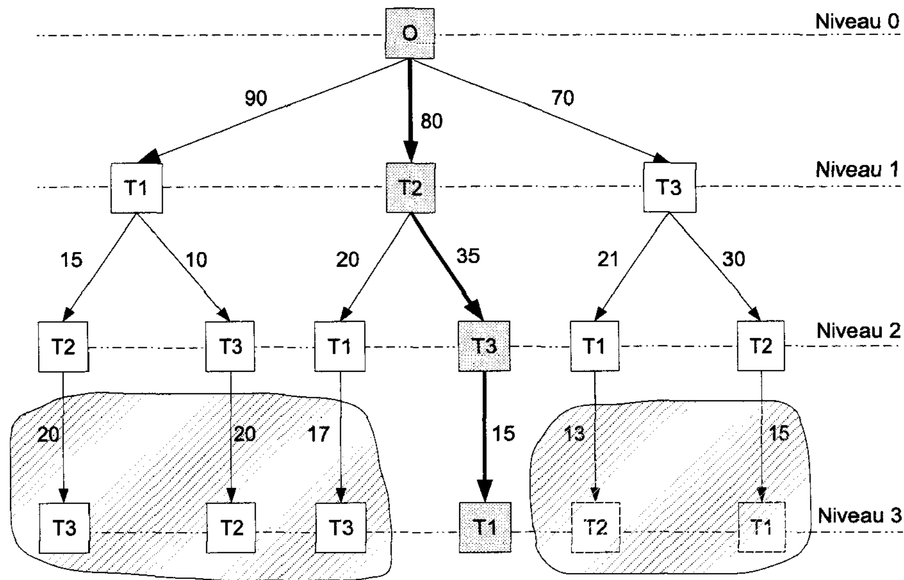

FIG. 3.11 – Heuristique gloutonne basée sur le gain biaisé

De ce fait, le premier algorithme a une complicité en $O(n^2)$. Quant à la complexité du deuxième algorithme, elle est calculée comme suit :

$$n^2 + (n-2)^2 + (n-4)^2 + \cdots + 1$$

D'où une complexité en $O(n^3)$.

La séquence trouvée par les heuristiques gloutonnes contient uniquement des techniques liées avec des relations de dépendance totale et partielle. Nous ajoutons à cette séquence les techniques indépendantes que nous avons exclues de l'ensemble de départ. La séquence finale représente l'arrangement de techniques qui fournie le meilleur gain d'espace mémoire. Le générateur de séquence fournie également le gain correspondant à la séquence finale. Ceci nous permet d'avoir une idée sur l'intérêt de compresser le code Java donné en entrée du profiler.

3.5 Conclusion

Nous avons proposé dans ce chapitre un système de compression basé sur un profiler pour la compression des fichiers *class* Java. L'originalité de ce travail est de chercher parmi les techniques de compression disponibles celles qui permettent d'offrir un meilleur taux de compression toute en tenant compte des caractéristiques de l'application Java à compresser.

Dans la première section, nous avons montré la nature de la corrélation qui existe entre le profilage et l'optimisation ; notamment la compression. Néanmoins, cette relation est initialement indirect : elle nécessite l'intervention d'un testeur d'application ou d'un programmeur afin d'exploiter les résultats fournis par le profiler et décider des remaniements nécessaires pour améliorer la performance de l'application testée. Par la suite, cette relation est devenue plutôt directe : le profiler collecte des informations spécifiques pour perfectionner une technique d'optimisation donnée. Notre approche se distingue par prise en compte du profil du code et des interactions entre les techniques impliquées dans la compression.

La deuxième section a exposé l'architecture de notre profiler dédié à la compression. Dans un premier temps, nous avons caractérisé la performance des techniques élémentaires de compression. Outre le fait que la performance d'une technique dépend du fichier à compresser, la co-application de deux ou plusieurs techniques de compression peut s'avérer impossible ou ne pas fournir le gain escompté. Ces observations représentent les raisons qui sont derrière notre contribution. Ainsi, dans un deuxième temps, nous avons proposé un nouveau concept de profiler dédié à la compression qui permet à la fois de gérer les dépendances entre les techniques et de sélectionner celles qui offrent le meilleur gain en terme d'espace mémoire par rapport au profil du code Java compilé à compresser. L'architecture de notre profiler est basée sur quatre modules. Le premier module est appelé scrutateur chargé de la scrutation des fichiers *class* java et de l'extraction des informations nécessaires pour l'évaluation du gain. Le deuxième module est l'analyseur de dépendance, il permet de définir la nature des relations de dépendance qui peuvent exister entre les techniques de compression. Le troisième module, évaluateur de performance, calcule le gain relatif à une technique de compression seule ou combinée avec d'autres. Le quatrième module, générateur de stratégies, utilise les données fournies par les autres modules pour établir la ou les séquences de techniques de compression qui permettent d'offrir le meilleur taux de compression.

Dans la troisième section, nous avons décrit en détail le principe de fonctionnement des modules qui composent le profiler. En plus, nous avons montré la façon dont les modules interagissent entre eux pour mener à terme la tâche confiée au profiler. L'analyse de dépendances identifie les relations entre les techniques de compression. Pour ce faire, nous avons défini trois types de relations de dépendances basés sur la performance : indépendance, dépendance totale et dépendance partielle. Le module analyseur de dépendances se base sur ces trois relations pour générer une table et un graphe de dépendances. Deux méthodes ont été proposées pour construire la table et le graphe de dépendances : numérique et analytique. La table et le graphe sont transmis d'une part et d'autre vers le module évaluateur de performance et le module de génération de stratégies. Par ailleurs, nous avons mis en place au niveau de l'évaluateur de performance une méthodologie qui lui permet de fournir le gain sans appliquer réellement la ou les techniques de compression. Cette méthodologie passe par une formulation du gain et l'identification des éléments contenus dans les fichiers *class* Java et intervenant dans le calcul du gain. Le module évaluateur de performance collabore avec le module de scrutation pour quantifier les gains engendrés par les techniques de compression. Enfin, nous avons mis en œuvre une méthode d'optimisation combinatoire pour permettre au module générateur de stratégies d'exploiter les données récoltées par les autres modules et de fournir en sortie du profiler une stratégie efficace pour la compression du code Java profilé.

Ce nouveau concept de profiler dédié à la compression permet de tirer le meilleur profit d'un ensemble de techniques de compression donné. L'assurance de fournir le meilleur profit passe par la recherche de l'adéquation entre le profil des fichiers *class* Java à compresser et la stratégie de compression à appliquer. Afin de montrer la faisabilité et la validité de notre modèle, nous avons développé un prototype du profiler. Les résultats de l'application de ce prototype sur un ensemble de benchmarks font l'objet du chapitre suivant.

Chapitre 4

Mise en œuvre du profiler

4.1 Introduction

Nous avons présenté dans le chapitre précédent un nouveau concept de profiler dédié à la compression du code Java compilé. Ce profiler prend en entrée un ensemble de techniques de compression et des fichiers *class* Java à compresser. En sortie, il fournit la meilleure stratégie de compression en terme de performance. Le noyau du profiler est formé de quatre modules qui interagissent ensemble pour assurer l'adéquation entre le code à compresser et la stratégie de compression. Sur cette base, un prototype a été développé sur la base des principes énoncés dans le chapitre précédent, et sera décrit dans les paragraphes suivants :

Ce prototype intègre un ensemble de techniques de compression applicables sur les fichiers *class* Java. Cette sélection se veut représentative des classes de techniques de compression présentées dans la section 2.3.4 du chapitre 2. De plus, les techniques choisies permettent de couvrir toutes les relations de dépendances étudiées dans la section 3.4.1 du chapitre 2. La première section de ce chapitre donne une brève description de chaque technique.

La deuxième section s'intéresse au module analyseur de dépendances. La méthode analytique est utilisée pour déterminer la nature des relations qui existent entre les techniques de compression. Ainsi les dépendances sont établies selon l'effet que peut avoir une technique compression sur le gain des autres techniques. À la fin de cette section, nous dressons le tableau de dépendance obtenu grâce à cette analyse.

La mise en œuvre du module d'évaluation de performance fait l'objet de la troisième section. Nous déterminons ainsi de façon statique le gain et la fiche de scrutation relatifs aux techniques incluses dans le profiler.

L'implémentation du profiler fait l'objet de la quatrième section. Cette dernière décrit la librairie BCEL utilisée pour analyser les fichiers *class* Java. Elle présente aussi le diagramme de classes du prototype.

La cinquième et dernière section est dédiée à la présentation des fonctionnalités intégrées dans le prototype à travers la description de son interface. Parmi ces fonctionnalités, il y a la visualisation du contenu des fichiers *class* Java, la configuration du profilage et la génération de la stratégie de compression.

4.2 Techniques de compression à implémenter dans le profiler

Nous présentons dans cette section un ensemble de techniques élémentaires de compression applicables sur les fichiers *class* Java. Ces techniques figurent parmi celles précédemment présentées (cf. chapitre 2 section 2.4). Nous avons opté pour un choix qui couvre les trois classes de techniques de la nouvelle classification présentée dans le chapitre 2 (cf. section 2.3.4). Nous avons également sélectionné les techniques de sorte que toutes les relations de dépendances soient illustrées.

Nous donnons dans ce qui suit la liste des techniques incluses dans le profiler avec une description de chacune d'entre elles :

Rétrécissement des noms (RN) : cette technique consiste à renommer les noms de packages, classes, méthodes et variables de façon à réduire l'espace mémoire nécessaire à leur représentation. Ainsi, chaque nom est remplacé par un caractère ; le code résultant est incompréhensible, mais le résultat de l'exécution demeure le même ;

Restructuration des descripteurs de méthodes (RDM) : la restructuration des descripteurs de méthodes offre une représentation compact par rapport à celle définie par le format standard. Chaque descripteur de méthode est remplacé par des indices vers des entrées de la table de constantes qui représentent le type des variables données en entrée ou récupérées en sortie d'une méthode donnée ;

Factorisation Bytecode (FB) : la factorisation du bytecode Java consiste à remplacer les séquences de code redondantes (les patterns) dans le bytecode par de nouvelles instructions, appelées aussi macros-instructions. De ce fait, le code est plus compact et occupe moins d'espace qu'il ne l'était auparavant ;

Représentation arborescente de noms (RAN) : comme plusieurs autres techniques de compression, la représentation arborescente des noms permet d'éliminer quelques redondances dans la table des constantes. Cette redondance provient du fait que les noms des classes sont représentés par son chemin en entier. Sachant que plusieurs classes peuvent appartenir au même package, alors nous pouvons retrouver des noms de packages qui se répètent entre les chemins des classes. La représentation arborescente permet d'éviter cette redondance ;

Partitionnement de la table de constantes (PTC) : cette technique s'applique également sur la table des constantes. Elle regroupe les entrées de même type dans une

même partition dans la table des constantes. Ceci permet de supprimer les étiquettes qui définissent le type de chaque entrée ;

Suppression des informations de débogage (SID) : cette technique supprime les informations superflus. Les informations de débogage en font partie du fait qu'elles ne sont pas utiles pour le bon déroulement de l'exécution ;

Indices implicites (II) : cette technique représente une alternative à l'indexage explicite présent dans le format original. Elle permet de supprimer l'indice nécessaire pour identifier l'entrée ciblée en plaçant cette dernière selon un ordre prédéfini ;

Les techniques qui viennent d'être citées se répartissent sur les trois classes de techniques (cf. tableau 4.1) avec un avantage pour celles relatives à la table de constantes vu que cette dernière occupe une part très importante d'un fichier *class* Java.

Techniques	Globales	Relatives à la table de constantes	Relatives au bytecode
RN		X	
RDM		X	
FB			X
RAN		X	
PTC		X	
SID	X		
II		X	

TAB. 4.1 – Répartition en classes des techniques de compression

4.3 Analyse de dépendances des techniques de compression

Nous avons présenté dans la section 3.4.1 du chapitre 3 deux méthodes d'analyse de dépendances des techniques de compression. La première méthode, dite numérique, identifie le type de relation selon le résultat du gain de deux techniques T_k et T_i évaluées de façon séparées et combinées. Cette méthode fait appel au module évaluation de performance. Le résultat de cette méthode dépend fortement de la qualité de l'estimation faite au niveau de ce module d'évaluation de performance. Par conséquent, nous avons opté pour la deuxième méthode, dite analytique, qui est indépendante des autres modules.

Rétrécissement des noms (RN) : cette technique agit sur les constantes Utf8 en représentant des noms de packages, classes, méthodes et variables avec des caractères. Elle réduit la taille des Utf8 sans changer sa structure. De ce fait, elle n'affecte pas la performance des techniques de partitionnement de la table de constantes, d'indices implicites, de suppression des informations de débogage et de factorisation (agit sur le bytecode). Elles sont donc totalement indépendantes de cette techniques. Par contre les performances des autres techniques sont affectées. En effet, la taille totale de la constante Utf8 représentant le chemin d'une classe est réduite, ce qui affecte le gain engendré par des techniques qui en sont dépendantes lorsqu'elles sont appliquées à la suite de la technique de rétrécissement des noms.

Le tableau 4.2 représente les relations de dépendances des techniques de compression par rapport à la technique de rétrécissement des noms (RN). Ainsi, nous distinguons deux techniques (RDM et RAN) qui dépendent partiellement de RN et quatre techniques (FB, PTC, SID et II) qui sont totalement indépendantes de RN.

Techniques	RDM	FB	RAN	PTC	SID	II
Rétrécissement des noms (RN)	$\rightsquigarrow$	$\dashrightarrow$	$\rightsquigarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$

TAB. 4.2 – Dépendances avec la technique de rétrécissement des noms

Restructuration des descripteurs de méthodes (RDM) : cette technique remplace les paramètres dans les descripteurs de méthodes par des indices vers des entrées qu'elle ajoute à la table des constantes. Parmi ces entrées, nous pouvons trouver de nouvelles constantes représentant des noms de classes à rétrécir ou à transformer sous forme arborescente. Cette restructuration a un effet négligeable sur le gain de la technique de partitionnement de la table de constantes. Nous la considérons, avec la technique d'indices implicites, comme étant totalement indépendantes de la restructuration des descripteurs de méthodes. De même pour les techniques de factorisation et de suppression des informations de débogage puisqu'elles ne sont pas concernées par la forme et le contenu des descripteurs de méthodes ;

Techniques	RN	FB	RAN	PTC	SID	II
Restructuration des descripteurs de méthodes (RDM)	⇝	⇢	⇝	⇢	⇢	⇢

TAB. 4.3 – Dépendances avec la technique de restructuration des descripteurs de méthodes

Le tableau 4.3 représente les relations de dépendances des techniques de compression par rapport à la technique de restructuration des descripteurs de méthodes (RDM). Ainsi, nous distinguons deux techniques (RN et RAN) qui dépendent partiellement de RDM et quatre techniques (FB, PTC, SID et II) qui en sont totalement indépendantes.

Factorisation Bytecode (FB) : la factorisation agit sur le bytecode Java en remplaçant les séquences de code redondantes (les patterns) par de nouvelles instructions. De ce fait, aucun autre composant de la structure du fichier *class* Java n'est affecté. Par conséquence, les autres techniques de l'ensemble de départ peuvent être appliquées après la factorisation sans aucun effet sur leur performance.

Le tableau 4.4 montre que toutes les techniques sont totalement indépendantes de FB.

Techniques	RN	RDM	RAN	PTC	SID	II
Factorisation du Bytecode (FB)	⇢	⇢	⇢	⇢	⇢	⇢

TAB. 4.4 – Dépendances avec la technique de factorisation du bytecode

Représentation arborescente de noms (RAN) : cette technique découpe chaque constante Utf8 représentant une classe selon les packages formant le chemin. Il est donc impossible d'appliquer la technique de rétrécissement des noms puisque la structure sur laquelle elle s'applique est transformée. Quand à la technique de restructuration de descripteurs de méthodes, elle demeure applicable, mais son gain est affecté. Les techniques restantes (FB, PTC, SID et II) sont cependant totalement indépendantes de la technique en question.

Techniques	RN	RDM	FB	PTC	SID	II
Représentation arborescente des noms (RAN)	$\not\rightarrow$	$\rightsquigarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$

TAB. 4.5 – Dépendances avec la technique de représentation arborescente des noms

Le tableau 4.5 représente les relations de dépendances des techniques de compression par rapport à la technique de représentation arborescente des noms (RAN). Nous distinguons une technique (RDM) qui dépend partiellement de RAN et une autre (RN) qui en est totalement dépendante. Quand au reste des techniques (FB, PTC, SID et II) elles sont liées à RAN avec une relation d'indépendance totale.

Partitionnement de la table de constantes (PTC) : cette technique n'effectue aucun changement sur le contenue des constantes à part la suppression des étiquettes qui indiquent le type de la constante. De ce fait, son application sur le code à compresser n'affecte pas le gain des techniques. En revanche, elle entre en conflit avec la technique d'indices implicites (II) ; ce qui rend impossible l'application de cette dernière.

Le tableau 4.6 représente les relations de dépendances des techniques de compression à la technique de partitionnement de la table de constantes (PTC). Seule la technique d'indices implicites (II) est totalement dépendante de PTC. En revanche, les autres techniques (RN, RDM, FB, RAN, et SID) en sont totalement indépendantes.

Techniques	RN	RDM	FB	RAN	SID	II
Partitionnement de la table de constantes (PTC)	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\not\rightarrow$

TAB. 4.6 – Dépendances avec la technique de partitionnement de la table de constantes

Suppression des Informations Débogage (SID) : tout comme la technique de factorisation du bytecode, cette technique n'affecte pas le gain des autres techniques du fait que ces données ne sont pas concernées par les informations de débogage.

Techniques	RN	RDM	FB	RAN	PTC	II
Suppression des Informations de Débogage (SID)	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$

TAB. 4.7 – Dépendances avec la technique de suppression des informations de débogage

Le tableau 4.7 montre que toutes les techniques sont totalement indépendantes de la technique de suppression des informations de débogage.

Indices implicites (II) : au niveau de la dépendance, cette technique se comporte de façon réciproque avec les autres techniques : cinq techniques (RN, RDM, FB, RAN et SID) en sont totalement indépendantes et la sixième (PTC) est exclue (dépendance totale) puisque l'indexage implicite impose un ordre de placement de certaines entrées dans la table de constantes.

Le tableau 4.8 montre une indépendance totale des techniques de compression par rapport à la technique d'indices implicites (II). Une exception est faite pour la technique PTC à cause d'une dépendance totale.

Techniques	RN	RDM	FB	RAN	PTC	SID
Indices Implicites (II)	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\dashrightarrow$	$\nrightarrow$	$\dashrightarrow$

TAB. 4.8 – Dépendances avec la technique d'indices implicites

Cette analyse permet de construire la table de dépendance de l'ensemble des techniques de compression en entrée du module d'analyse de dépendances (cf. table 4.9). Pour faciliter la lecture de ce tableau de dépendances nous y avons représenté uniquement les relations de dépendances partielles et totales.

Techniques	RN	RDM	FB	RAN	PTC	SID	II
Rétrécissement des Noms (RN)		$\leadsto$		$\leadsto$			
Restructuration des Descripteurs de Méthodes (RDM)	$\leadsto$			$\leadsto$			
Factorisation du Bytecode (FB)							
Représentation arborescente des noms (RAN)	$\nrightarrow$	$\leadsto$					
Partitionnement de la table de constantes (PTC)							$\nrightarrow$
Suppression des Informations de Débogage (SID)							
Indices Implicites (II)					$\nrightarrow$		

TAB. 4.9 – Table de dépendances des techniques de compression

Cette table de dépendances est transmise au module d'évaluation de performance. La mise en œuvre de ce module fait l'objet de la section suivante.

4.4 Évaluation de performances

Nous étudions dans cette section le gain en espace mémoire résultant de la compression des fichiers *class* Java donnés en entrée du profiler. Ce gain provient de l'application d'une ou plusieurs techniques de compression. Le principe de fonctionnement de chaque technique de compression est détaillé afin de déterminer statiquement son gain. À cet effet, une fiche relative à chaque technique de compression est générée et est transmise au

module scrutateur.

4.4.1 Rétrécissement des noms (RN)

Le gain engendré par cette technique est égale à la différence de taille entre la longueur des noms avant et après rétrécissement. La scrutation des classes Java permet de retrouver la longueur des noms et de calculer par la suite le gain. Pour chaque classe d'indice i, le scrutateur effectue les opérations suivantes :

- extraire les noms de la classe courante et des classes (du même package) appelées et retourner leurs tailles TNC_{ij} et leur nombre NC_i respectifs ;
- extraire les noms des champs de la classe courante et retourner leurs tailles TNF_{ik} et leur nombre NF_i ;
- extraire les noms des méthodes de la classe courante et retourner leurs tailles TNM_{ih} et leur nombre NM_i.

Ces données, récupérées grâce au scrutateur, permettent de calculer le gain selon la formule suivante :

$$G_{RN} = \sum_{i=1}^{N}(\sum_{j=1}^{NC_i} TNC_{ij} + \sum_{k=1}^{NF_i} TNF_{ik} + \sum_{h=1}^{NM_i} TNM_{ih} - NC_i - NF_i - NM_i) \quad (4.1)$$

Avec :
- N : nombre de classes incluses dans le package ;
- NC_i : nombre de classes incluses dans le package et appelées par la classe i ;
- TNC_{ij} : taille du nom de la $j^{\text{ième}}$ classe incluse dans le package et appelée par la classe i ;
- TNF_{ik} : taille du $k^{\text{ième}}$ nom du champ inclu dans la classe i ;
- TNM_{ih} : taille du $h^{\text{ième}}$ nom de méthode invoquée dans la classe i.

4.4.1.1 Évaluation dépendante relative à (RDM)

L'application de la technique RDM ajoute à la table de constantes des noms de classes (correspondant aux paramètres de méthodes) dans le cas où ces derniers n'y figurent pas. Prendre en compte de l'influence de RDM sur RN revient à appliquer la formule de calcul du gain G_{RN} sur les noms non dupliqués en appliquant l'équation 4.2.

$$G_{RN/RDM} = \sum_{i=1}^{N} \sum_{j=1}^{NM_i} TNC_NDup_{ij} - NNC_NDup_{ij} \quad (4.2)$$

Avec :
- N : nombre de classes incluses dans le package ;
- NM_i : nombre de méthodes appelées par la $i^{\text{ième}}$ classe ;
- TNC_NDup_{ij} : taille des noms non dupliqués générés à partir des paramètres de la $j^{\text{ième}}$ méthode appelée par la $i^{\text{ième}}$ classe ;
- NNC_NDup_{ij} : nombre des noms non dupliqués générés à partir des paramètres de la $j^{\text{ième}}$ méthode appelée par la $i^{\text{ième}}$ classe ;

4.4.2 Restructuration des descripteurs de méthodes (RDM)

La technique de restructuration des descripteurs consiste à ajouter pour chaque méthode j de la classe i la nouvelle structure DM_Struct suivante :

```
MethodDescriptor
{
    u2 parameter_count;
    u2 parameter_indices[parameter_count];
    u2 return_type_index;
}
```

Avec :
- *parameter_count* : nombre de paramètres de la méthode en question ;
- *parameter_indices* : tableau de taille *parameter_count* contenant les indices des entrées représentant les paramètres de la méthode en question ;
- *return_type_index* : indice de la constante contenant le type de retour de la méthode en question.

La taille TDM_Struct de cette structure DM_Struct est égale à :

$$TMD_Struct = 2 + 2 * parameter_count + 2$$

De plus, cette technique ajoute des entrées à la table des constantes pour représenter les paramètres. Les constantes qui occupent un espace mémoire significatif correspondent aux paramètres et résultats de type classe.

Le calcul du gain G_{RDM} représente la différence de la taille en mémoire entre l'ancienne et nouvelle représentation des descripteurs de méthodes. L'ancienne représentation correspond à la taille de l'ensemble des constantes de type Utf8 représentant les descripteurs de méthode. En revanche, la taille de la nouvelle représentation est obtenue en additionnant la taille des constantes Utf8 représentant les paramètres des méthodes et celles des nouvelles structures de type descripteur de méthodes.

$$G_{RDM} = \sum_{i=1}^{N} \sum_{j=1}^{NM_i} (TUtf8_DM_{ij} - TDM_Struct_{ij} - TMP_{ij} + TMP_Dup_{ij}) \quad (4.3)$$

Avec :

- N : nombre de classes incluses dans le package;
- NM_i : nombre de méthodes appelées par la $i^{\text{ième}}$ classe;
- $TUtf8_DM_{ij}$: taille de la constante Utf8 représentant le descripteur de la méthode j de la classe i ;
- TDM_Struct_{ij} : taille de la structure représentant un descripteur de la méthode j de la classe i ;
- TMP_{ij} : taille du résultat et des paramètres d'une méthode j de la classe i après transformation.
- TMP_Dup_{ij} : taille du résultat et des paramètres dupliqués (déjà existants) d'une méthode j de la classe i après transformation.

4.4.2.1 Évaluation dépendante relative à (RAN)

La technique RAN transforme la représentation des noms de classes. De ce fait, les noms ajoutés à la table de constantes par la technique RDM ne trouvent pas des constantes qui leurs sont identiques. Ils sont donc gardés et leurs tailles comptabilisées lors de l'évaluation dépendante de la technique (RDM) par rapport à (RAN).

$$G_{RDM/RAN} = \sum_{i=1}^{N} \sum_{j=1}^{NM_i} (TUtf8_DM_{ij} - TDM_Struct_{ij} - TMP_{ij}) \quad (4.4)$$

Avec :

- N : nombre de classes incluses dans le package;
- NM_i : nombre de méthodes appelées par la $i^{\text{ième}}$ classe;

- $TUtf8_DM_{ij}$: taille de la constante Utf8 représentant le descripteur de la méthode j de la classe i ;
- TDM_Struct_{ij} : taille de la structure représentant un descripteur de la méthode j de la classe i ;
- TMP_{ij} : taille du résultat et des paramètres d'une méthode j de la classe i après transformation.

4.4.2.2 Évaluation dépendante relative à (RN)

Contrairement à la technique (RAN), la technique (RN) ne transforme que les noms de classes internes (dans le package). Lors de l'évaluation dépendante du gain de (RDM) par rapport à (RN), il faut donc tenir compte des noms de classes externes qui peuvent créer des duplications. Ainsi, le gain est calculé comme suit :

$$G_{RDM/RN} = \sum_{i=1}^{N} \sum_{j=1}^{NM_i} (TUtf8_DM_{ij} - TDM_Struct_{ij} - TMP_{ij} + TMPE_Dup_{ij}) \quad (4.5)$$

Avec :

- N : nombre de classes incluses dans le package ;
- NM_i : nombre de méthodes appelées par la $i^{\text{ième}}$ classe ;
- $TUtf8_DM_{ij}$: taille de la constante Utf8 représentant le descripteur de la méthode j de la classe i ;
- TDM_Struct_{ij} : taille de la structure représentant un descripteur de la méthode j de la classe i ;
- TMP_{ij} : taille du résultat et des paramètres d'une méthode j de la classe i après transformation.
- $TMPE_Dup_{ij}$: taille du résultat et des paramètres externes d'une méthode j de la classe i dupliqués (déjà existant) après transformation.

4.4.2.3 Évaluation dépendante relative à (RN/RAN)

Les transformations effectuées par la technique (RN) suivie par la technique (RAN) sur les noms de classes empêchent la possibilité de trouver des redondances entre les noms déjà existants et ceux ajoutés par la technique (RDM). Ainsi, l'évaluation dépen-

dante $G_{RDM/RN-RAN}$ de la technique (RDM) par rapport aux techniques (RN) et (RAN) s'effectue de la même manière que $G_{RDM/RAN}$ (cf. Équation 4.4).

4.4.3 Factorisation du bytecode (FB)

L'évaluation du gain de la factorisation passe par la détection des séquences de code qui se répètent dans le bytecode. La prise en compte du chevauchement entre les patterns [2] permet de réduire l'espace de recherche des patterns et d'aller le plus loin possible dans notre recherche afin de trouver les meilleures séquences. Pour calculer le gain de cette technique, nous identifions, pour chaque classe i, les séquences de code redondantes et le nombre d'occurrence. Notons que les séquences les plus fréquentes sont de taille 2 ou 3. Le calcul du gain est donné par la formule suivante :

$$G_{FB} = \sum_{i=1}^{N} \sum_{j=1}^{NP_i} (TP_{ij} * (NOP_{ij} - 1) \tag{4.6}$$

Avec :
- N : nombre de classes incluses dans le package;
- NP_i : nombre de patterns de la classe i.
- TP_{ij} : Taille du pattern j de la classe i.
- NOP_{ij} : nombre d'occurrence du pattern j de la classe i.

4.4.4 Représentation arborescente des noms (RAN)

Cette technique transforme les constantes Utf8 représentant des noms de classes en un arbre dont les nœuds symbolisent les éléments du chemin menant à la classe en question. Chaque nœud a la structure suivante :

```
Element
{
    u2 index_parent;
    u2 index_courant;
}
```

Le scrutateur établi la liste des éléments (LEN_i) d'une classe i et retourne la taille de chaque élément de cette liste. Le gain est obtenu en calculant la différence entre la taille

des constantes Utf8 représentant des noms de classes et celle des éléments constituant la nouvelle représentation arborescente des chemins des classes. Le calcul du gain engendré par cette technique est donné par la formule suivante :

$$G_{RAN} = \sum_{i=1}^{N}(\sum_{j=1}^{C_i} TCUtf8_{ij} - \sum_{k=1}^{Q_i} TEN_{ik}) \quad (4.7)$$

Avec :

- N : nombre de classes incluses dans le package;
- C_i : nombre des constantes de type Utf8 représentant les noms de classes dans la classe i;
- Q_i : nombre des éléments de la classe i.
- $TCUtf8_{ij}$: taille de la $j^{\text{ième}}$ constante Utf8 représentant les noms de classes dans la classe i;
- TEN_{ik} : taille du $k^{\text{ième}}$ élément appartenant à la liste LEN_i;

4.4.4.1 Évaluation dépendante relative à (RN)

Afin retrouver le gain de la technique RAN appliquée à la suite de RN, il faut tenir compte du fait que les noms des classes et des packages internes sont réduits à cause de la technique RN. Ainsi, les valeurs des $TCUtf8_{ij}$ sont réduites de la taille du chemin des classes internes (sans compter les caractères de substitution). Il en va de même pour les valeurs des TEN_{ik}.

$$\begin{aligned} G_{RAN/RN} = \sum_{i=1}^{N}(\sum_{j=1}^{C_i} TCUtf8_{ij} - \sum_{k=1}^{CI_i}(TChemin_{ik} - NNChemin_{ik}) \\ + \sum_{l=1}^{Q_i} TEN_{il} - \sum_{m=1}^{QI_i}(TENI_{im} - 1)) \end{aligned} \quad (4.8)$$

Avec :

- N : nombre de classes incluses dans le package;
- C_i : nombre des constantes de type Utf8 représentant les noms de classes dans la classe i;
- $TCUtf8_{ij}$: taille de la $j^{\text{ième}}$ constante Utf8 représentant les noms de classes dans la classe i;

- CI_i : nombre des classes internes appelées par la $i^{\text{ième}}$ classe ;
- $TChemin_{ik}$: taille du chemin de la $k^{\text{ième}}$ classe appartenant au package appelée par la $i^{\text{ième}}$ classe ;
- $NNChemin_{ik}$: nombre de noms (de classes ou de packages) dans le chemin de la $k^{\text{ième}}$ classe appartenant au package appelée par la $i^{\text{ième}}$ classe ;
- Q_i : nombre des éléments de la classe i.
- TEN_{il} : taille du $l^{\text{ième}}$ élément appartenant à la liste LEN_i ;
- TNI_{im} : taille du nom de la $m^{\text{ième}}$ classe (ou package) interne appelée par la $i^{\text{ième}}$ classe ;
- QI_i : nombre des éléments internes (appartenant au package) et appelés par la $i^{\text{ième}}$ classe ;
- $TENI_{ik}$: taille d'un élément représentant le $k^{\text{ième}}$ nom (de classe ou de package) interne appartenant à la $i^{\text{ième}}$ classe ;

4.4.4.2 Évaluation dépendante relative à (RDM)

La technique RDM ajoute à la table de constantes les noms de classes non dupliqués contenus dans les paramètres des méthodes. La technique RAN les transforme en une forme arborescente. Les nœuds forment la liste (LEN_Dup) des noms des éléments formant les chemins des classes non dupliquées. Le gain engendré par cette transformation est donné par la formule

$$G_{RAN/RDM} = \sum_{i=1}^{N}(\sum_{j=1}^{NM_i} TMP_Dup_{ij} - \sum_{k=1}^{QD_i} TEN_Dup_{ik}) \tag{4.9}$$

Avec :

- N : nombre de classes incluses dans le package ;
- NM_i : nombre de méthodes appelées par la $i^{\text{ième}}$ classe ;
- QD_i : nombre d'éléments de la liste LEN_Dup ;
- TMP_Dup_{ij} : taille du résultat et des paramètres dupliqués (déjà existants) d'une méthode j de la classe i après transformation.
- TEN_Dup_{ik} : taille du $k^{\text{ième}}$ élément de la liste LEN_Dup.

4.4.5 Partitionnement de la table de constantes (PTC)

Le gain engendré par la technique de partitionnement provient de la suppression des étiquettes de chaque constante appartenant à la table de constantes. Comme la taille d'une étiquette est d'un octet, l'espace mémoire gagné est équivalent à la somme des nombres d'entrées NE_i de la classe C_i. Il faut néanmoins soustraire les 12 compteurs (12 types des constantes) d'entrées de chaque partition. Le calcul du gain G_{PTC} revient à appliquer la formule suivante :

$$G_{PTC} = \sum_{i=1}^{N}(NE_i - 12) \tag{4.10}$$

Avec :
- N : nombre de classes incluses dans le package ;
- NE_i est le nombre d'entrées de la table des constantes de la classe i.

4.4.6 Suppression des informations de débogage (SID)

Le gain engendré par cette technique est égal à l'espace occupé par les informations de débogage. Dans le code compilé Java, nous retrouvons trois types d'informations de débogage : *LocalVariableTable*, *LineNumberTable* et *SourceFile*.

- **SourceFile** : Cet attribut respecte la structure suivante :

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

  Á ces huit octets, nous ajoutons les quatorze octets qu'occupe la constante Utf8 qui contient la chaîne "SourceFile". Nous y ajoutons également la taille de la constante Utf8 qui représente le nom du fichier source Java (3+ taille de la chaîne "NomClass.java").
- **LineNumberTable** : Cet attribut respecte la structure suivante :

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
```

```
        u2 line_number_table_length;
        {   u2 start_pc;
            u2 line_number;
        } line_number_table[line_number_table_length];
    }
```

La taille de cette structure est égal à :

$$2 + 4 + 2 + (2 + 2) * line_number_table_length$$

À ceci s'ajoute la taille de la constante Utf8 qui désigne un attribut *LineNumberTable* (3 + taille de la chaîne "LineNumberTable".)

- *LocalVariableTable* : La structure qui définie cet attribut est comme suit :

```
LocalVariableTable_attribute {
u2 attribute_name_index;
u4 attribute_length;
u2 local_variable_table_length;
    {   u2 start_pc;
        u2 length;
        u2 name_index;
        u2 descriptor_index;
        u2 index;
    }   local_variable_table[local_variable_table_length];
}
```

La taille de cette structure est de :

$$2 + 4 + 2 + 10 * local_variable_table_length$$

En plus, nous comptabilisons l'espace mémoire occupé par la constante Utf8 représentant le nom de l'attribut (3+ taille de la chaîne "LocalVariableTable".)

Le calcul de l'espace mémoire relatif à chacune de ces trois structures nous permet d'établir la formule de calcul du gain de la technique de suppression des informations de débogage :

$$G_{SID} = 14 * N + \sum_{i=1}^{N} \sum_{j=1}^{NM_i} TNLM_{ij} + TVL_{ij} \tag{4.11}$$

Avec :

- N : nombre de classes incluses dans le package ;
- NM_i : nombre de méthodes de la classe i ;
- $TNLM_{ij}$: table des numéros de lignes de la méthode j de la classe i ;
- TVL_{ij} : table des variables locales de la méthode j de la classe i.

4.4.7 Indices implicites (II)

Lorsque cette technique est appliquée sur les champs et méthodes pour indexer implicitement la classe à laquelle ils appartiennent, le champ *class_index* est supprimé des structures *CONSTANT_Fieldref_info*, *CONSTANT_Methodref_info* et *CONSTANT_-InterfaceMethodref_info*. L'espace économisé par cette technique est égale à l'espace occupé par le champs class_index dans toutes les classes. Le calcul de ce gain G_{II} est donné par la formule suivante :

$$G_{II} = 2 * \sum_{i=1}^{N} (NM_i + NC_i + NI_i) \quad (4.12)$$

Avec :

- N : nombre de classes incluses dans le package ;
- NM_i : nombre de constantes de type *CONSTANT_Methodref* dans la classe i ;
- NC_i : nombre de constantes de type *CONSTANT_Fieldref* dans la classe i ;
- NI_i : nombre de constantes de type *CONSTANT_InterfaceMethodref* dans la classe i.

4.5 Implémentation du profiler

L'implémentation du profiler a été effectuée en langage Java. Le choix de ce langage permet de bénéficier de l'existante de librairies Java qui peuvent être réutilisées. Le profilage du code Java compilé passe par l'analyse des fichiers en entrée du profiler. Comme ces fichiers sont structurés selon le format du fichier *class* Java, il est donc nécessaire de respecter ce format lors de la lecture des fichiers en question.

Il existe plusieurs librairies d'ingénierie dédiées à l'analyse des fichiers *class* Java. Nous

citons à titre d'exemple les librairies ASM[9], Javassist[10] et BCEL [47]. Nous avons décidé d'exploiter les possibilités offertes par ces librairies en codant les modules de scrutation et d'évaluation en langage Java. Ainsi, nous appelons directement les objets et les fonctions incluses dans la librairie à partir de notre code source. Nous avons choisi d'utiliser la librairie BCEL pour développer notre prototype. Nous consacrons la section suivante à la présentation de cette librairie et des raisons qui nous ont mené à ce choix.

4.5.1 La librairie d'ingénierie BCEL

BCEL (Byte Code Engineering Library) est une librairie d'ingénierie de bytecode. Elle est conçue pour donner aux utilisateurs une possibilité commode d'analyser, créer et manipuler les fichiers *class* Java. Les classes sont représentées par les objets qui contiennent toute l'information symbolique d'une classe donnée : les méthodes, des champs et en particulier, les instructions bytecode. De tels objets peuvent être lus à partir d'un fichier existant, être transformés par un programme (un chargeur de classes par exemple), et déposés dans un nouveau fichier. La librairie d'Ingénierie de bytecode (BCEL) peut être aussi utile pour l'étude de la Machine Virtuelle Java (JVM) et le format des fichiers *class* Java. Cette librairie contient un vérificateur de bytecode appelé Justice, qui donne généralement beaucoup plus d'informations sur ce qui n'est pas correct dans le bytecode par rapport aux messages standards donnés par une Machine Virtuelle Java.

Le choix de BCEL s'explique par la richesse de sa documentation et le large panel de possibilités offertes par cette librairie. D'ailleurs, elle est plus utilisée que les autres librairies. En effet, BCEL a déjà été employée, avec succès, dans plusieurs projets comme des compilateurs, des optimiseurs, des générateurs de code et des outils d'analyse[11]. Apache témoigne de ce succès en reprenant le code de BCEL pour l'intégrer dans son projet Jakarta[12].

[9] `http://asm.objectweb.org`
[10] `http://www.jboss.org/products/javassist`
[11] Des exemples de projets utilisant BCEL sont disponibles à l'adresse suivante : http ://bcel.sourceforge.net/, rubrique "Projects".
[12] Disponible à l'adresse suivante : http ://jakarta.apache.org/bcel/.

4.5.2 Conception du profiler

La couche logicielle relative au profiler interagit avec les fichiers *class* Java à travers l'API BCEL. Les informations collectées sont utilisées pour quantifier le gain et établir la stratégie de compression. Nous donnons dans ce qui suit le diagramme de classes du profiler selon la modélisation UML.

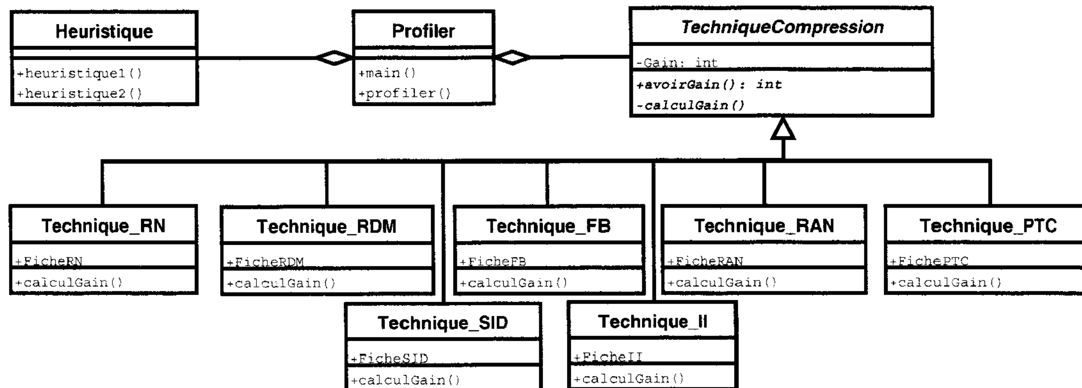

FIG. 4.1 – Diagramme de classes

La figure 4.1 représente le diagramme de classes du profiler. Sur cette figure, nous retrouvons les sept techniques de compression incluses dans le profiler. La classe *TechniqueCompression* représente la classe mère de toute technique de compression. Elle contient l'attribut *Gain* et une méthode *AvoirGain()* qui correspondent au gain obtenu grâce à une technique de compression donnée. Ce gain est calculé grâce à la méthode *CalculGain()*. Cette méthode est redéfinie au niveau des classes qui représentent les techniques de compression (*Technique_RN*, *Technique_RDM*, *Technique_FB*, *Technique_RAN*, *Technique_PTC*, *Technique_SID* et *Technique_II*). Ces classes contiennent chacune un attribut regroupant la liste éléments nécessaires à l'évaluation du gain. La classe *Heuristique* permet de trouver la stratégie selon l'une des deux heuristiques présentées dans la section 3.4.4.2 du chapitre 3. La classe *Profiler* contient la fonction principale qui enclenche le processus de profilage en appelant la méthode *profiler()*.

4.6 Interface du prototype

Le profiler proposé a été implémenté sous la forme d'un outil avec une interface graphique. Cette dernière est représentée par une fenêtre principale avec deux onglets.

- **Fichiers *class* Java :** cet onglet (cf. figure 4.2) présente les fichiers *class* Java à compresser. À gauche, un arbre représente l'arborescence des fichiers qui composent le code Java compilé. Son contenu est initialisé à la suite du choix de l'ensemble des fichiers représentant le code Java à compresser. Ce choix s'effectue à travers le menu "Fichier/Ouvrir". Sur la partie droite de l'onglet, une zone texte sert à afficher les informations contenues dans chaque *class* Java. Parmi ces informations, il existe les entrées de la table de constantes, les attributs, le bytecode, etc.

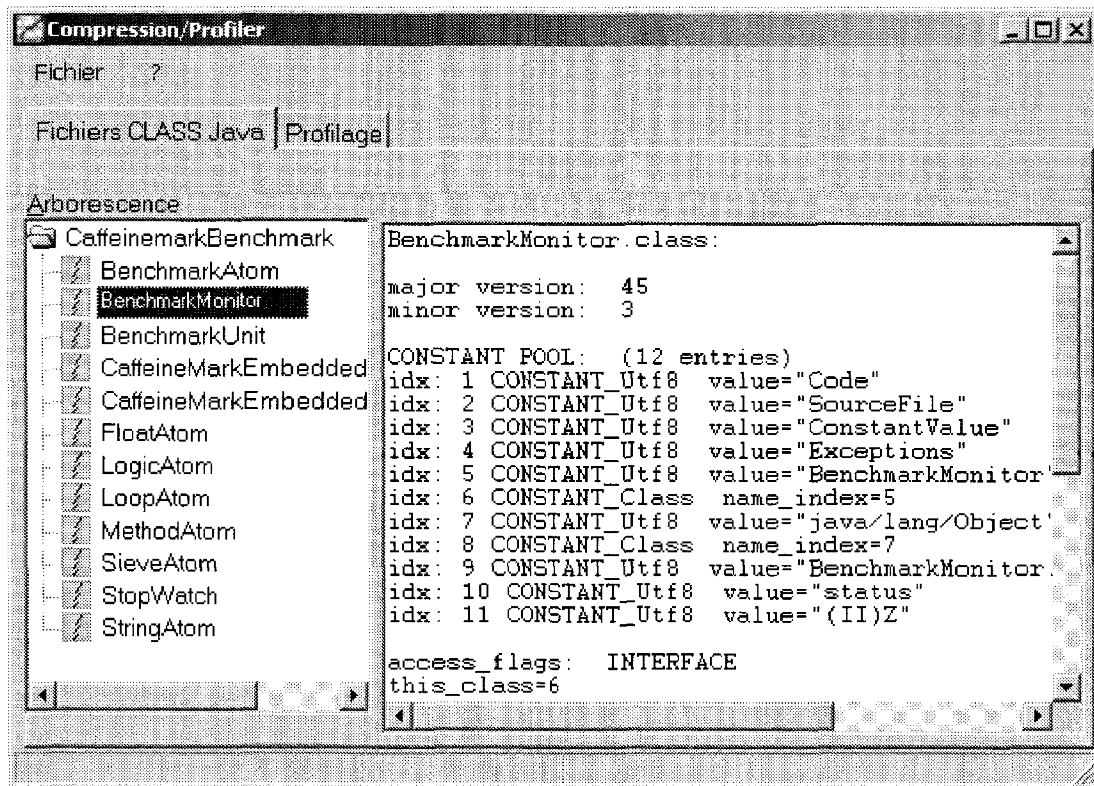

FIG. 4.2 – Interface graphique : premier onglet

- **Profilage** : le deuxième onglet (cf. figure 4.3) permet d'effectuer le profilage dédié à la compression. À travers des boutons à options, l'utilisateur choisi l'heuristique de recherche. Les techniques de compression à inclure dans la procédure de profilage en cauchant les cases correspondantes.

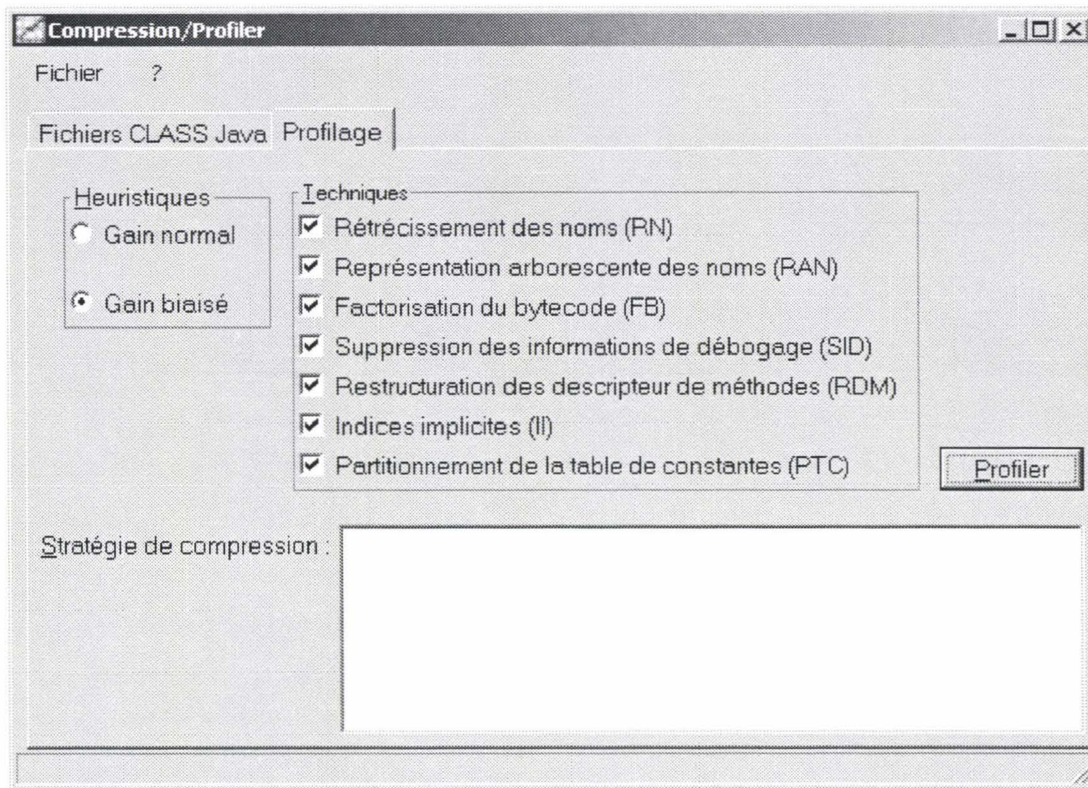

FIG. 4.3 – Interface graphique : deuxième onglet

La configuration de profilage (options) et les résultats (stratégie et taux de compression) peuvent être enregistrés dans un fichier texte grâce au menu "Fichier/Enregistrer Sous".

BIBLIOTHÈQUE UNIV. VALENCIENNES

4.7 Conclusion

Nous avons présenté dans ce chapitre la mise en œuvre du profiler dédié à la compression du code Java compilé.

Dans la première section, nous avons intègré un ensemble de techniques de compression applicables sur les fichiers *class* Java. La sélection des techniques est basée sur deux critères. Le premier critère veuille à ce que toutes les classes de techniques de compression (techniques relatives à la table de constantes, au bytecode et à l'ensemble de la structure du fichier *class* java) soient représentées. Le deuxième critère est la couverture de l'ensemble des relations de dépendance (totale, partielle et de non dépendance).

Dans la deuxième section, nous avons analysé les dépendances entre ces techniques de compression. Nous avons utilisé la méthode analytique pour identifier les types de relation. Nous avons privilégié la méthode analytique sur la méthode numérique vu que la première est indépendante des autres modules du profiler. Nous avons donc étudié l'effet de chaque technique de compression sur le gain du reste des techniques appartenant à l'ensemble. Ceci nous a permis de dresser la table de dépendance relatives aux techniques incluses dans le profiler. Notons que cette table conforte bien le choix des techniques puisque tous les types de dépendance y figurent.

La troisième section concerne le module d'évaluation de performance. Ce dernier permet de quantifier le gain engendré par une ou plusieurs techniques de compression. Nous avons donc développé le mode de fonctionnement de chaque technique de façon à établir la formule de calcul du gain. De la même façon, nous avons construit la fiche contenant les éléments intervenant dans le calcul du gain. Cette fiche est envoyée vers le module scrutateur afin de déterminer la valeur de chacun de ses éléments.

L'implémentation du profiler a fait l'objet de la quatrième section. Nous avons utilisé une librairie de classes Java appelée BCEL. Cette dernière permet l'extraction des différents éléments qui composent la structure d'un fichier *class* Java. Outre le fait qu'elle soit extensible, BCEL permet la manipulation et la modification du code Java compilé. Ceci peut être utile pour une application effective des techniques de compression sur les fichiers *class* Java. Les classes Java implémentant le profiler se situent au dessus de la couche logicielle représentée par la librairie BCEL.

La dernière section a présenté l'interface graphique que nous avons implémentée afin de facilité l'utilisation du profiler. À travers cette interface, l'utilisateur peut charger les

fichiers *class* Java à compresser, observer leur contenu, configurer les options de profilage et générer les résultats du profilage avant de les sauvegarder à la fin.

Nous avons appliqué le prototype de ce profiler sur du code Java compilé. Les résultats obtenus suite à ces expérimentations sont donnés dans le dernier chapitre dans lequel nous présentons quelques ouvertures pour des travaux à venir.

Chapitre 5

Expérimentations et perspectives

5.1 Introduction

Nous avons présenté dans le chapitre précédent la mise en œuvre d'un profiler dédié à la compression le code Java compilé. Ce profiler gère un ensemble de techniques de compression applicables sur les fichiers *class* Java.

La première section de ce chapitre présente les expérimentations effectuées dans l'objectif de valider le concept de profilage tel qu'il a été défini dans le chapitre 3. La validation concerne essentiellement les heuristiques et la stratégie de compression. La validation des heuristiques injectées dans le module générateur de stratégie passe par des temps de recherche raisonnables et des solutions suffisamment proches de l'optimum. Quant à la stratégie de compression, sa validation réside dans le gain obtenu grâce au profilage.

La seconde section ouvre des perspectives de recherche relatives aux travaux présentés dans ce mémoire. Ces perspectives couvrent des améliorations, extensions et réutilisations applicables sur le concept du profilage. Les améliorations proposées concernent l'enrichissement de l'ensemble des techniques et la prise en compte du temps d'exécution ou de l'énergie consommée. Elles permettent de mieux perfectionner et exploiter le concept de profilage dédié à l'optimisation (compression).

5.2 Expérimentations

La fiabilité du profiler repose, en partie, sur une maîtrise du principe de fonctionnement des techniques de compression appliquées sur le code Java compilé. Ces connaissances interviennent par exemple lors de l'établissement de la table de dépendances avec la méthode analytique ou encore pour définir les formules de calcul de gain. Cependant, les heuristiques proposées pour la recherche du meilleur gain nécessite une expérimentation étant donnée l'aspect "approché" des solutions. Dans le même sens, nous donnons quelques résultats de l'application du profilage sur des différents benchmark. La validation des heuristiques et des stratégies de compression font l'objet de cette section.

5.2.1 Évaluation des heuristiques

Le premier volet de l'expérimentation concerne l'évaluation des algorithmes de recherche du chemin le plus efficace (ou de stratégie). Deux heuristiques ont été proposées (Cf. chapitre 3 section 3.4.4). Elles désignent respectivement l'heuristique selon le gain normal et le gain biaisé ; exprimés en terme de pourcentage. Une étude comparative de la qualité des résultats obtenus par ces heuristiques est donnée dans le tableau 5.1.

Instances	Heuristique 1 (%)	Heuristique 2 (%)
5	15.60	4.16
6	15.70	6.45
7	16.59	5.65
8	15.26	6.20
9	15.13	5.36
10	16.22	4.96

TAB. 5.1 – Évaluation du gain des heuristiques

Le tableau 5.1 permet de comparer les solutions données par les deux heuristiques par rapport à la solution optimale (obtenue grâce à une résolution exacte). La première colonne représente les instances sur lesquelles ont été appliquées les heuristiques. Ces instances ont été généré aléatoirement avec des cardinalités (nombre de techniques de compression) allant de 5 à 10. La deuxième et troisième colonnes représentent respectivement le pourcentage de la solution approchée par rapport à la solution optimale (meilleur gain).

La première heuristiques donne des résultats à 15% ou 16% de l'optimum. Le caractère basique de l'heuristique 1 fait que sa performance n'est pas très proche de l'optimum. Quant à la deuxième heuristique, la solution qu'elle fournit est à environ 5% de la solution optimale. Ceci s'explique par un critère d'exploration (gain biaisé) plus sophistiqué comparé à la première heuristique.

Nous effectuons à présent une évaluation temporelle de ces deux heuristiques. Il s'agit de mesurer le temps mis par une heuristique pour trouver une solution. Ces mesures ont été effectuées sur un AMD Athlon 3.4 Ghz. Plusieurs instances ont été utilisées pour observer l'impact de la cardinalité sur le temps de résolution. Les résultats expérimentaux sont donnés par le tableau 5.2.

Instances	Heuristique 1 (Sec)	Heuristique 2 (Sec)
5	0.75	2.9
6	1.06	7.46
7	1.42	15.45
8	1.82	25.5
9	2.28	38.92
10	2.79	54.87

TAB. 5.2 – Évaluation temporelle des heuristiques

Les temps d'exécution des heuristiques 1 et 2 sur des petites instances (de l'ordre de 5 ou 10) sont nuls ou insignifiants. Nous introduisons donc un temps de latence symbolique pour chaque nœud exploré afin de pouvoir observer l'évolution du temps d'exécution en fonction de la taille des instances.

Au vu de ces résultats, nous suggérons l'utilisation de la méthode exacte pour les instances à faible cardinalité (par exemple entre 2 et 6). Ceci garantie une solution optimale dans des temps raisonnables. Dans les autres cas (instances relativement grandes), les heuristiques peuvent être utilisées; avec une recommandation de la deuxième heuristique pour avoir des solutions à environ 5 ou 6% de l'optimum.

La deuxième phase de l'évaluation concerne les stratégies fournies par le profiler; la section suivante lui est consacrée.

5.2.2 Évaluation des stratégies

La deuxième série d'expérimentations concerne l'évaluation des stratégies générées par le profiler. L'objectif est de montrer l'influence des relations de dépendances sur le gain et par conséquent la capacité du profiler à trouver l'adéquation entre l'application à profiler et la stratégie de compression à appliquer. Un ensemble de librairies (Cf. tableau 5.3) a été utilisé pour les expérimentations.

Librairies	Nombre de classes	Taille (Ko)
Sun JDK (Java.*)	713	1665
JEODE (awt)	160	257
Exerces	887	2333
Swing	1275	3164
MIDP (J2ME)	303	729
Scimark2lib	24	52

TAB. 5.3 – Librairies Java

Le tableau 5.3 donne une description des librairies qui composent le benchmark. Le nombre de classes appartenant aux librairies est assez varié ; allant de 24 (Scimark2lib) jusqu'à 1275 (Swing). Les expérimentations consistent à appliquer le profilage sur les librairies et de donner la stratégie de compression adéquate à chacune d'entre elles. Afin d'expliciter les raisons qui ont conduit à ces stratégies, nous relevons les résultats obtenus grâce à l'évaluation des gains. Le tableau 5.4 donne les pourcentages que représentent les gains des techniques de compression (évalués séparément) par rapport à la taille des fichiers non compressés.

Techniques	Sun JDK	JEODE	Exerces	Swing	MIDP	Scimark2lib
RN	3.73 %	3.87%	4.35%	4.69%	4.97%	2.25%
RDM	5.48 %	8.63%	10.99%	7.78%	5.25%	4.01%
FB	4.14%	4.43%	5.46%	4.07%	6.97%	6.11%
RAN	1.65%	3.21%	4.16%	3.65%	3.27%	2.16%
PTC	2.98%	3.79%	3.32%	3.48%	3.13%	4.13%
SID	16%	3.16%	1.92%	14.59%	2.10%	15.2%
II	1.92%	4%	2.71%	3.07%	2.81%	3.76%

TAB. 5.4 – Évaluation du gain des techniques isolées

D'une part, le tableau 5.4 montre que le gain d'une technique de compression peut varier d'une façon remarquable, telle que la technique SID qui voit son gain passer de 1.92% (avec Exerces) à 16% (avec Sun JDK). Le profiler peut donc nous renseigner sur l'intérêt d'appliquer une technique de compression donnée. D'autre part, ce même tableau montre que les gains des techniques PTC et II ne peuvent pas être ordonnés de façon universelle. En effet, la technique PTC devance II dans certains cas (avec MIDP par exemple) et enregistre un moindre gain comme avec JEODE. Vu la dépendance totale entre PTC et II, seule la technique la plus efficace est gardée par le profiler.

Étant donnée les relations de dépendantes partielles qui existent entre certaines techniques de compression (RN, RDM et RAN), nous en donnons les évaluations dépendantes du gain relatif à ces techniques dans le tableau ci-dessous :

Stratégies	Sun JDK	JEODE	Exerces	Swing	MIDP	Scimark2lib
RN - RDM	7.77%	11.3%	13.26%	10.53%	8.33%	5.95%
RDM - RAN	7.51%	12.56%	16.19%	12.04%	8.71%	5.34%
RN - RAN	5.38%	7.08%	8.52%	8.34%	8.24%	4.43%
RDN - RN	8.37%	12.2%	14.18%	11.71%	9.6%	6.24%
RAN - RDM	5.69%	8.87%	12.46%	8.73%	6.64%	6.32%
RN - RDM - RAN	11.27%	17.88%	20.57%	16.89%	14.56%	10.01%
RN - RAN - RDM	12.5%	12.74%	16.81%	13.42%	11.61%	7.59%
RDN - RN - RAN	9.42%	22.11%	26.46%	18.89%	13.87%	11.66%

TAB. 5.5 – Évaluation dépendante du gain

Le tableau 5.5 montre l'importance de l'ordre d'application des techniques dépendantes pour avoir un meilleure performance de la compression. En effet, aucun ordre prédéfini ne garantie le meilleur gain. Par exemple, le profilage de *Scimark2lib* montre que l'application de [RAN - RDM] est plus avantageux que [RDM - RAN] ; ce qui n'est pas valable pour *MIDP*. Il en va de même pour les stratégies composées de trois techniques puisque la meilleure stratégie change d'un benchmark à un autre : [illegible]N - RDM - RAN] pour *MIDP*, [RN - RAN - RDM] pour *Sun JDK* et [RDN - RN - RA[illegible] pour *Exerces*.

En déterminant la meilleure stratégie possible, le générateur de stratégie permet d'améliorer la performance de la compression de 5.56% en moyenne. Cette valeur est obtenue en faisant la moyenne des écarts entre la meilleure et la pire des stratégies (composées de trois techniques) appliquées sur ces benchmarks.

5.3 Perspectives

Certaines propositions présentées dans cette section ont pour objectif d'améliorer les performances du profiler. Les effets de ces propositions peuvent se refléter sur le mode de fonctionnement du profiler ou sur les résultats qu'il fournit (stratégie et gain). Les autres propositions représentent des réutilisations possibles du concept du profilage dans d'autres domaines d'optimisation. Dans ce qui suit sont présentés quelques améliorations possibles du profiler.

5.3.1 Enrichir l'ensemble des techniques de compression

Le profilage, tel que nous l'avons présenté, a été validé par un prototype avec un ensemble de techniques de compression. Toutefois cet ensemble peut être étendu avec d'autres techniques de compression. L'ajout d'une technique de compression a un effet sur les performances du profiler qui nécessitent d'être analysé. Pour ce fait, nous commençons par décrire la procédure à suivre. L'inclusion d'une technique dans le profiler débute par l'étude de sa dépendance par rapport aux autres techniques déjà existantes. Une des deux méthodes proposées (numérique ou analytique) permettent de déterminer la nature de ses relations de dépendances. Cette étape est suivie par l'évaluation du gain engendré par cette technique en identifiant les éléments et la formule de calcul de gain. La nouvelle technique devient donc habilitée à intégrer le profiler. De ce fait, il est possible de la retrouver dans la stratégie de compression proposée par le profiler.

L'intégration d'une nouvelle technique de compression a un effet sur les performances du profiler. Nous distinguons deux cas de figures qui dépendent de la nature de la dépendance qui existe entre cette technique et celles déjà intégrées dans le profiler :

- Absence de dépendance : Il s'agit du cas où la technique ajoutée a une relation d'indépendance totale avec les autres techniques. Autrement dit, le gain engendré par cette technique s'ajoute au gain des autres techniques. L'ajout de telles techniques est bénéfique pour le profiler puisqu'il y a une grande probabilité de les retrouver dans la stratégie finale proposée. Ceci s'explique par le fait que le gain de la nouvelle technique est cumulé avec le gain de la stratégie ; indépendamment des techniques qui composent cette dernière. Les rares cas où le gain de cette technique est négatif n'affecte point la performance du profiler puisque ce dernier ne considère que les techniques qui fournissent un gain positif.

- Existence de dépendance : La technique ajoutée a des relations de dépendances partielles ou totales. La nature de cette relation se répercute par définition sur le gain des autres techniques de compression. L'évaluation de gain de la technique tient également compte de ces relations de dépendance. Notons que l'ajout de ce genre de techniques entraîne, d'une part, l'augmentation des choix de stratégies à explorer. D'autre part, ceci peut engendrer un accroissement du gain résultant du profilage si la technique ajoutée figure dans la stratégie fournie. L'effet de l'enrichissement de l'ensemble des techniques de compression sur le gain ne peut être que positif ou nul. Ceci est garanti par le profiler qui se charge d'exclure la technique ajoutée (si elle ne contribue pas à l'augmentation du gain) de la stratégie finale par le biais des algorithmes des heuristiques

5.3.2 Profilage pour l'optimisation de l'énergie

L'optimisation de la consommation de l'énergie est un domaine de recherche aussi important que la compression de code notamment pour les systèmes embarqués. Cependant, nous remarquons le manque de travaux sur ce sujet, en particulier l'optimisation de l'énergie du code Java compilé (code à pile). La difficulté de ce domaine réside en premier lieu dans la quantification de l'énergie consommée, phase nécessaire, pour pouvoir évaluer l'apport d'une technique d'optimisation de l'énergie donnée.

Les rares études qui existent se focalisent sur la quantification de l'énergie. Il en ressort un constat maigre et non convainquant. Bien que ces travaux sont basiques, ils ont le mérite de proposer des points de départ ou des repères pour des travaux complémentaires à venir. La proposition de Lafond [42] [43] s'inscrit dans ce registre. Ce dernier donne une quantification basée sur les opcodes. En effet, il associe à chaque opcode dans le bytecode Java une valeur représentant l'énergie consommée. Ces valeurs sont obtenus par une exécution répétitive de la même instruction. La moyenne de l'énergie consommée est attribuée à l'instruction exécutée. La quantification basée sur une estimation énergétique par instruction n'apporte pas une résolution exacte. De plus, certains facteurs peuvent intervenir, notamment le défaut de cache et les environnements d'exécution.

Le même concept de profilage dédié à la compression peut être appliqué sur l'optimisation de l'énergie. Nous pouvons également combiner la compression avec l'optimisation de l'énergie (ou temps d'exécution) dans un même profiler. Pour ce faire, la fonction de maximisation devient donc à objectif double et la stratégie obtenue est un compromis

entre les deux.

5.3.3 Autres perspectives

À ce stade de finalisation, le profiler se charge de la génération de la stratégie permettant d'obtenir le meilleur gain compte tenu des techniques incluses dans le profiler et du code Java compilé à compresser. L'application des techniques de compression est déléguée à l'utilisation de cet outil de profilage. L'amélioration proposée réside dans la mise en pratique de la compression selon la stratégie générée par le profiler.

La recherche du meilleur gain passe par l'application des heuristiques pour de grandes instances. Les meilleurs résultats obtenus jusqu'à présent sont garantis par la deuxième heuristique et avoisinent les 5%. Cependant, il est possible de perfectionner la recherche avec de nouvelles heuristiques. Des techniques d'optimisation classiques peuvent être envisagées. Un critère de recherche encore plus sophistiqué peut donner de meilleurs résultats. Une alternative consiste à définir un gain biaisé couvrant trois niveaux au lieu de deux (comme c'est le cas de la deuxième heuristique) pourrait donner des solutions plus proches de l'optimal.

5.4 Conclusion

Dans la première section de ce chapitre, nous avons effectué une série d'expérimentations afin de valider le concept de profilage dédié à la compression des fichiers *class* Java. Ce processus de validation concerne les heuristiques de recherche du meilleur gain et les stratégies de compression. L'application des heuristiques sur plusieurs instances donne des solutions proches de l'optimum, notamment la deuxième heuristique qui fournit des solutions à 5% de la meilleure solution. Les temps d'exécution sont tout à fait acceptables. Dans le cas réel, le nombre de techniques est généralement faible, ce qui rend une résolution exacte possible et du coup préférable. Pour de plus grandes instances, la deuxième heuristique est la mieux appropriée. De ce fait, aucune limite sur le nombre de techniques intégrées dans le profiler n'est imposée.

Dans la deuxième section sont présentées les perspectives qu'ouvre ce travail de recherche. Nous y proposons quelques idées qui permettent d'améliorer la performance du profiler, tel que l'enrichissement du profiler avec de nouvelles techniques de compression. D'autres perspectives qui concernent les possibilités de réutilisation du concept de profilage pour l'optimisation d'autres ressources (temps CPU ou énergie) sont également présentées.

Conclusion générale

Le thème principal abordé dans ce mémoire concerne la compression du code Java pour les systèmes embarqués. Le premier chapitre présente donc à la fois le format des fichiers *class* Java et les systèmes embarqués. À partir de l'étude des caractéristiques relatives au code compilé Java et aux systèmes sur lesquels il est embarqué, nous avons pu remarquer que la viabilité du code Java sur un support embarqué passe par la résolution de la problématique soulevée par ses inconvénients (lenteur d'exécution et encombrement mémoire) et profiter par conséquence de ses avantages (portabilité, sécurité). La présentation des spécifications de la Machine Virtuelle Java a ressorti les difficultés notamment techniques entourant ce sujet.

Le second chapitre est, quant à lui, centré sur l'état de l'art des techniques de compression du code Java. Après avoir présenté le cadre général, nous avons étudié trois classifications existantes des techniques de compression. Il s'est avéré qu'aucune de ces classifications n'est adéquate à notre contexte. Nous avons donc proposé une nouvelle classification qui tient compte des spécificités des fichiers *class* Java. Elle distingue trois classes de techniques qui s'appliquent respectivement sur la table de constantes, le bytecode et sur l'ensemble du fichier *class* Java. Ensuite, nous avons classé les techniques élémentaires de compression selon ces trois classes. De nouveaux formats de code Java compilé existants ont été exposés. Ils réutilisent plusieurs techniques élémentaires de compression. Néanmoins, il n'existe pas auparavant une étude préalable de l'adéquation des techniques utilisées avec le code à compresser.

Notre contribution est présentée dans le chapitre trois. Elle consiste en la conception et la mise en oeuvre d'un système basé sur un profiler pour guider la compression des fichiers class Java. Ce profiler permet d'établir une stratégie de compression efficace offrant le meilleur taux de compression tout en tenant compte des caractéristiques du code en entrée et des dépendances entre les techniques de compression employées. Le profilage passe par l'identification des dépendances entre les techniques de compression. Ainsi, nous avons

défini trois types de relations (indépendance, dépendance totale et dépendance partielle) basées sur l'influence entre les techniques au niveau du gain.

Notre démarche s'appuie sur quatre modules chargés des tâches suivantes :

- l'examen des fichiers en entrée en vue de l'extraction d'informations utiles pour le guidage du processus de compression.
- l'analyse des dépendances des opérations de compression en terme d'interaction mutuelle des unes avec les autres. Pour ce faire, nous avons mis au point deux méthodes, l'une numérique basée sur l'estimation des performances, l'autre analytique permettant de déterminer la nature des dépendances entre les opérations de compression.
- l'évaluation statique des performances permettant le choix de stratégie de compression. Nous avons, à ce propos, établi les différents paramètres caractérisant chacune des méthodes traitées permettant cette évaluation.
- la construction du chemin de compression le plus efficace dans l'espace de recherche représenté par un graphe orienté et l'application d'heuristiques appropriées pour aboutir à des résultats intéressants.

Dans le quatrième chapitre, nous avons présenté un prototype de profiler basé sur un ensemble de techniques de compression applicables sur les fichiers *class* Java. À travers la présentation du prototype, nous avons procédé à l'application du principe de fonctionnement de chaque module du profiler à la fois sur les techniques de compression et le code Java compilé. Ainsi, la table de dépendances relative à ces techniques de compression a été établie. De même, nous en avons donné l'évaluation statique du gain dans les cas d'évaluation séparée et dépendante. Nous avons obtenu en résultat une graphe sur lequel est appliqué l'une des heuristiques de recherche du chemin le plus efficace.

La validation expérimentale de ce prototype a fait l'objet du cinquième chapitre. Elle a porté essentiellement sur les heuristiques et la stratégie de compression. D'une part, vous avons montré que les heuristiques donnent des solutions proches de l'optimum, notamment la deuxième qui fournit des solutions à 5% de la meilleure solution. Les temps d'exécution sont tout à fait acceptables. D'autre part, les tests menés sur des benchmarks, formés d'exemples réels de code Java compilé, ont montré l'existante de plusieurs stratégies possibles et la capacité du profiler à determiner la meilleure. La deuxième section de ce chapitre a pour objectif de donner les perspectives qu'ouvre ce travail de recherche. Des perspectives telles que, l'enrichissement de l'ensemble des techniques de compression ou l'application effective de la stratégie finale de compression, permettent d'améliorer la

performance du profiler. D'autres perspectives représentent des exemples de réutilisation du concept de profilage dans d'autres domaines d'optimisation des resources (temps CPU ou énergie).

Annexe

Pour ne pas encombrer le chapitre 1, seules quelques structures ont été présentées (Cf. section 1.3.3.2). Cet annexe est donc consacré à la définition de certaines structures qui composent le format de fichiers *class* Java. Ces structures sont celles des champs, méthodes et attributs.

Les champs

Chaque champs est décrit par une structure *field_info* et est identifié par un nom et un descripteur uniques dans une même classe. Sa structure se présente comme suit :

```
field_info {
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Avec :

- *access_flags* : sa valeur représente les droits d'accès et les propriétés du champs ;
- *name_index* : représente l'index vers une entrée de la table des constantes de type *Utf8_info* contenant le nom du champs ;
- *descriptor_index* : index de la constante de type *Utf8_info* qui contient le descripteur du champs ;
- *attributes_count* : représente le nombre des attributs supplémentaires de ce champs ;
- *attributes* : un tableau d'attributs associés au champs. Chaque élément est un attribut de type *attribute_info*.

Les méthodes

```
method_info {
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Avec :

- *access_flags* : sa valeur représente les droits d'accès et les propriétés de la méthode ;
- *name_index* : représente l'index vers une entrée de la table des constantes de type *Utf8_info* contenant le nom de la méthode ;
- *descriptor_index* : index de la constante de type *Utf8_info* qui contient le descripteur de la méthode ;
- *attributes_count* : représente le nombre des attributs supplémentaires de cette méthode ;
- *attributes* : un tableau d'attributs associés à la méthode. Chaque élément est un attribut de type *attribute_info.*

Les attributs

```
attribute_info {
        u2 attribute_name_index;
        u4 attribute_length;
        u1 info[attribute_length];
    }
```

Avec :

- *attribute_name_index* : index de la constante de type *Utf8_info* représentant le nom de l'attribut ;
- *attribute_length* : représente la taille en octets de l'information qui suit dans cette structure ;
- *info* : information représentant le contenu de l'attribut.

Bibliographie

[1] Ammous, K., Benameur, N., Niar, S., and Abed, M., "Java Constant Pool Inspection". Program Acceleration by Application-driven and Architecture-driven Code Transformations Research network, PA3CT network symposium, Edegem, Belgium, september, 2004.

[2] Ammous, K., Benameur, N., Niar, S., and Abed, M., "Improving the Adequacy of Java Application to Embedded Architectures through Bytecode Optimisation", SETIT (IEEE), Sousse, Tunisia, march 15-22, 2004.

[3] Ammous, K., Benameur, N., and Abed, M., "Profile-based Optimization for Embedded Java Application" Industrial Embedded Systems (IEEE), Antibes Juan-Les-Pins, France, October 18-20, 2006.

[4] Ammous, K., and Benameur, N., "Java Virtual Machines Behavior on Embedded Systems" International Conference on Software Engineering (SE 2007), Innsbruck, Austria, february 13-15, 2007.

[5] Antonioli, D. N., Car : The Class Archive Format. Technical Report 2001.01 - January 2001.

[6] Benes, M., Wolfe, A., and Nowick, S. M., A high-speed asynchronous decompression circuit for embedded processors. In Proc. Conf. On Advanced Reserch in VLSI, September 1997.

[7] Beneš, M., Nowick, S. M., and Wolfe, A., A fast asynchronous Huffman decoder for compressed-code embedded processors. In Proc. International Symposium on advanced Research in Asynchronous Circuits and Systems, September 1998.

[8] Benitez, M. E., and Davidson, J. W., The advantages of machine-dependent global optimization. In Proceedings of the 1994 Conference on Programming Languages and Systems Architectures, pages 105–124, March 1994.

[9] Beszédes, Á., Ferenc, R., Gyimóthy, T., Dolenc, A., and Karsisto, K., Survey of

code-size reduction methods, ACM Computing Surveys (CSUR), v.35 n.3, p.223-267, September 2003

[10] Venners, B., Inside The Java Virtual Machine, 2nd Edition. ISBN : 0-07-135093-4. publisher : McGraw-Hill Osborne Media, 2000.

[11] Bradley, Q., Horspool, R. N., and Vitek, J., JAZZ : An efficient compressed format for Java archive files, in Proceedings of CASCON'98, November 1998.

[12] Briggs, P., Shillner, R., and Simpson, T. L., Dead code elimination. Technical report, Rice University, October 1993. Available via anonymous FTP.

[13] Budimlič, Z., and Kennedy, K., Optimizing Java : Theory and practice. Concurrency, Pract. Exp. (UK), 9(11) :445-63, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.

[14] Clausen, L. R., Schultz, U. P., Consel, C., and Muller, G., Java Bytecode Compression for Low-End Embedded Systems. ACM, published in TOPLAS volume 22(3), May 2000.

[15] Cooper, K. D., and McIntosh, N., Enhanced Code Compression for Embedded RISC Processors. Proc. SINGPLAN'99 Conference on Programming Language Design and Implementation, May 1999, Pages. 139-149.

[16] Corless, J. D., Compression of Java class files, MsC Thesis, Department of Computer Science, University of Victoria, Canada, 1996.

[17] Cover, T. M., and Thomas, J. A., Elements of Information Theory. John Wiley & Sons, 1991.

[18] Debray, S., Evans, W., and Muth, R., Compiler techniques for code compaction. Technical report TR00-04 march 2000.

[19] Debray, S., Evans, W., Profile-guided code compression, Proceedings SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI), p. 95-105, 2002.

[20] Engel, J., Programming for the Java Virtual Machine, ISBN-10 : 0-201-30972-6, Addison Wesley Professional, 1999.

[21] Ernst, J., Evans, W., Fraser, C. W., Lucco, S., and Proebsting, T. A., Code Compression. Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, Pages : 358 - 365, 1997.

[22] Franz, C. W., Myers, E.W., and Wendt, A. L., "Analysing and Compressing Assembly

Code", Proc. SIGPLAN'84 Symposium on Compiler Construction, June 1984, pages 117-121.

[23] Franz, M., Code generation on the fly : A key to portable software, PhD Thesis 10497, Swiss Federal Institute of Technology, Zurich, Switzerland, 1994.

[24] Franz, M. and Kistler, T., Slim binaries. Communications of the ACM 40 :12, Pages 87-94, Dec. 1997.

[25] Franz, M., Adaptive Compression of Syntax Trees and Iterative Dynamic Code Optimization : Two Basic Technologies for Mobile Object Systems. Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet, Springer LNCS vol 1222, Pages : 263 - 276, Feb 1997.

[26] Fraser, C. W., and Proebsting, T. A., Custom instruction sets for code compression. Unpublished manuscript, `http://www.cs.arizona.edu/people/todd/papers/pldi2.ps`

[27] Fraser, C. W., Automatic inference of models for statistical code compression. In Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI-99), pages 242–246, May 1999.

[28] Gosling, J., Joy, B., Steele, G., and Bracha, G., The Java Language Specification third edition, ADDISON-WESLEY edition, May 2005.

[29] Horspool, R. N., and Corless, J., Tailored compression of Java class files, Software - Practice & Experience, v. 28 n. 12, pp 1253 - 1268, October 1998

[30] Huffman, D. A., A method for the construction of minimum redundancy codes. In Proc. IRE, volume 40, pages 1089-1101, September 1952.

[31] Heydemann, K., Bodin, F., Charles, H.-P., A software-only compression system for trading-offs between performance and code size. Proceedings of the 2005 workshop on Software and compilers for embedded systems SCOPES '05, September 2005.

[32] Johansson, E., Nyström, S.-O., ProfileGuided Optimization Across Process Boundaries, ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation (Dynamo'00). Jan 18, 2000.

[33] Jones, R. and Lins, R., Garbage Collection : Algorithms for Automatic Dynamic Memory Management John Wiley & Sons, Ltd, 1996.

[34] Kadionik, P., Le dictionnaire des dêveloppeurs, 2005.

[35] Kennedy, K., A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, Program Flow Analysis : Theory and Applications, chapter 1 pages 5- 54. Prentice Hall, 1981.

[36] Kistler, T., and Franz, M., A Tree-Based Alternative to Java Byte-Codes. International Journal of Parallel Programming, 27(1) :21–34, Feb. 1999.

[37] Knuth, D. E., An Empirical Study of FORTRAN Programs. Software-Practice and Experience vol. 1, 105-13, 1971.

[38] Knuth, D. E., Computer Programming as an Art. ACM Turing Award lecture, pp 667-672, December, 1974.

[39] Kozuch, M., and Wolfe, A., Compression of embedded system programs. In pro. Int't Conf. on Computer Design, pages 270-277, 1994.

[40] Kozuch, M., and Wolfe, A., Performance analysis of the compressed code RISC processor. Technical Report CE-A95-2, Princeton University Computer Engineering, 1995.

[41] Krishnaswamy, A., Gupta, R., Profile Guided Selection of ARM and Thumb Instructions, ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems, pages 55-63, Berlin, Germany, June 2002.

[42] Lafond, S., Lilius, J., An Opcode Level Energy Consumption Model for a Java Virtual Machine, Virtual Machine Research and Technology Symposium, 2004.

[43] Lafond, S, Lilius, J., An Energy Consumption Model for Java Virtual Machine, Technical Report, Åbo Akademi University, Department of Computer Science, Finland, March 2004.

[44] Latendresse, M., Génération de machines Virtuelles pour l'exécution de programmes compressés. Thèse de doctorat, Département d'informatique et de recherche opérationnelle - faculté des arts et des sciences, décembre 1999.

[45] Lindholm, T., and Yellin, F., The Java Virtual Machine Specification (2nd Edition). Addison Wesley, 1997.

[46] Lawler, E., L., Lenstra, J., K., Rinnooy Khan, A., H., G., a[illegible]hmoys, D., B., The Traveling Salesman Problem : A Guided Tour of Combinatorial Optimization. John Wiley & Sons. ISBN 0-471-90413-9, 1985.

[47] Dahm, M., Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universitat Berlin, Institut fur Informatik, April 2001.

[48] McManus, É., JDistill, a program to shrink Java packages, `http://www.gr.osf.org/~emcmanus/jdistill.html`, April 1998.

[49] Muller, G., Moura, B., Bellard, F. and Consel, C., Harissa : a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. Proceeding of the Third USENIX Conference on Object-Oriented Technologies (COOTS), June 1997

[50] Proebsting, T. A., Optimizing an ANSI C interpreter with superoperators, in Conference record of POPL'95 : The 22th ACM SIG-Plan - SIG-Act symposium on principles of programming languages, pp 322 - 332, January 1995

[51] Pugh, W., Compressing Java class files, in Proceedings of the ACM SIG-Plan conference on programming language design and implementation (PLDI'99), ACM SIG-Plan Notices, v. 34 n. 5, pp 247-258, May 1999

[52] Rayside, D., Mamas, E., and Hons, E., Compact Java Binaries for Embedded Systems. in Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, page 9, 1999.

[53] Seal, D., ARM Architecture Reference Manual, Second Edition, Addison-Wesley Edition. ISBN-10 : 0201737191, december 2000.

[54] SUN MICROSYSTEMS. Java Card 2.11 virtual machine specification, may 2000. `http://java.sun.com/products/javacard/javacard21.html`

[55] Tip, F., Laffra, C., Sweeney, P. F., and Streeter, D., Practical Experience with an Application Extractor for Java. Proceedings of ACM/SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)'99, November 1999.

[56] Vahid, F., and Givargis, T., Embedded System Design, Prentice Hall, 2002.

[57] Vallée-Rai, R., A Java bytecode optimization framework. Master's thesis in School of Computer Science McGill University, Montreal, 2000.

[58] Van De Wiel, R., The code compaction bibliography. 2001. `http://www.extra.research.philips.com/ccb/`

[59] Verdiere, V. C., Cros, S., Fabre, C., Guider, R., Yovine, S., Speedup Prediction for Selective Compilation of Embedded Java Programs. In Proceedings of "Workshop on Embedded Software, EMSOFT'02". Grenoble, October 7-9, 2002.

[60] Wegman, M. N., Zadeck, F. K., Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 13 Issue 2 - Pages : 181 - 210 - April 1991.

[61] Welch, T. A., A technique for high-performance data compression. IEEE Computer, 17(6) :8-19, June 1984.

[62] Wolfe, A., and Chanin, A., Executing compressed programs on an embedded RISC architecture. In Wen mei Hwu, editor, proceeding fo the 25th Annual International Symposium on Microarchitecture, pages 81-941, Portland, OR, December 1992. IEEE Computer Society Press.

[63] Yourst, M. T., Inside Java Class Files, Dr. Dobb's Journal, n. 281, p. 46-52, Jan. 1998

[64] Ziv, J., and Lempel, A., A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3) :337–343, 1977.

Résumé

Notre travail de recherche se focalise sur la compression du code Java sous format de fichiers class Java. Notre contribution consiste à concevoir et mettre en œuvre un système basé sur un profiler pour guider la compression des fichiers class Java. Ce profiler permet d'établir une stratégie de compression efficace offrant le meilleur taux de compression en tenant compte des caractéristiques du code en entrée et des dépendances entre les techniques de compression.

La démarche suit quatre points : **(i)** l'examen du code Java afin d'en extraire les informations utiles au profilage, **(ii)** l'analyse des dépendances entre les techniques de compression, **(iii)** l'évaluation statistique des performances des techniques de compression et **(iv)** la définition d'heuristiques appropriées pour identifier le chemin de compression le plus efficace dans l'espace de recherche représenté par un graphe orienté.

Mots-clés :

Code Java compilé, Profilage, Dépendance, Heuristiques, Stratégie de Compression.

Abstract

Our study focuses on the compression of Java code represented by Java Class format files. Our contribution consists in designing and implementing a profiler based system in order to guide the compression of Java class files. Our profiler enables us to set up, on the basis of elementary compression techniques, an efficient compression strategy which delivers the best rate of compression. This strategy takes into consideration the features of the code given in input and dependencies between compression techniques.

Our approach is based on four points : **(i)** the study of the input files in order to extract the necessary information, **(ii)** the analysis of compression techniques dependencies in terms of effects produced by each technique to the others, **(iii)** the statistic performance assessment which allows evaluating compression rate and **(iv)** the definition of heuristics in order to identify the most efficient compression path in a research space characterized by an oriented graph.

Keywords :

Java Class Files, Profiling, Dependency, Heuristics, Compression Strategy.

Bibliothèque..Universitaire..de..Valenciennes
00900394