

Université de Montréal

Génération de machines virtuelles pour l'exécution de  
programmes compressés

par

Mario Latendresse

Département d'informatique et de recherche opérationnelle  
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures  
en vue de l'obtention du grade de  
Philosophiae Doctor (Ph.D.)  
en informatique

Décembre, 1999

© Mario Latendresse, 1999



QA  
76  
U54  
2000  
v. 019

Génération de machines virtuelles pour l'exécution de  
programmes compressés

Marie Lacroix

Thèse de doctorat en informatique  
présentée à l'Université de Sherbrooke

Le jury est composé de  
M. [Nom] président  
M. [Nom] rapporteur  
M. [Nom] examinateur  
M. [Nom] examinateur

Le 15 mai 2000

A Sherbrooke, Québec



Université de Montréal  
Faculté des études supérieures

Cette thèse intitulée:

**Génération de machines virtuelles pour l'exécution de  
programmes compressés**

présentée par

**Mario Latendresse**

a été évaluée par un jury composé des personnes suivantes :

---

Rudolf Keller  
(Président-rapporteur)

---

Marc Feeley  
(Directeur de thèse)

---

Jean Vaucher

---

Charles Consel  
(Membre externe)

---

À déterminer  
(Représentant du doyen)

Thèse acceptée le \_\_\_\_\_

## SOMMAIRE

---

L'objectif général de cette thèse est de mettre au point des techniques et des outils de conception de machines virtuelles adaptées à l'exécution de programmes compressés.

Une méthode conventionnelle de la représentation des programmes de machines virtuelles utilise le code-octet : le code opérationnel est codé sur un octet et un argument occupe un multiple de huit bits. Cette codification simplifie le décodage, car toutes les instructions sont alignées sur des frontières d'octet.

Nous utilisons quatre méthodes pour compresser les programmes : la création de nouvelles instructions à partir d'instructions de base, c'est-à-dire les macro-instructions ; la compression des codes opérationnels en utilisant des codes de Huffman ; la création de formats compacts pour coder les arguments ; et le non alignement des instructions sur une frontière d'octet.

Le tout s'inscrit dans des outils de génération automatique de machines virtuelles, utilisables sous le contrôle du concepteur.

Pour le langage *Scheme*, compilé vers notre machine générale *Machina*, les résultats expérimentaux démontrent la capacité de l'approche à générer de nouvelles instructions et codifications adaptées au langage et à la compression de ses programmes. Notre approche génère des programmes très compacts, s'exécutant à des vitesses comparables à d'autres interprètes *Scheme*. Pour la machine virtuelle de Java, la *JVM*, notre méthode découvre de nouvelles instructions et formats adaptés à une bonne compression. L'ensemble des benchmarks *BYTEmark*, ainsi que la librairie *JDK 1.0.2*, a un facteur moyen de compression de 60%.

Ces méthodes sont accompagnées d'outils de génération automatique de deux types de décodeurs pour une exécution rapide sans décompression préalable. Nous appliquons de nombreux benchmarks sur ces décodeurs, utilisant deux processeurs d'architectures différentes, pour des machines virtuelles adaptées aux langages de programmation *Scheme* et *Java*. Les vitesses obtenues démontrent la viabilité de l'approche. Pour plusieurs benchmarks *BYTEmark Java*, le ralentissement est moins de 10%.

## ABSTRACT

---

The general goal of this thesis is to design techniques and tools to generate virtual machines adapted for the execution of compressed programs.

A common representation of programs for virtual machines uses byte-code: the operational code is coded on one byte and an argument uses a multiple of eight bits. This codification simplifies decoding since all instructions are aligned on a byte boundary.

We use four methods to compress programs: the creation of new instructions from sequences of basic ones, that is macro-instructions, the compression of operational codes using canonical Huffman coding, the creation of compact formats for arguments and non alignment of instructions on byte boundary.

The techniques are embedded in tools to automatically generate virtual machines under the user's guidance.

For the Scheme language, compiled towards our general virtual machine *Machina*, the experimental results demonstrate the approach to generate new instructions and codification adapted to the language and the compression of its programs. Our approach generates very compact programs executing at speeds comparable to other Scheme interpreters. For the Java virtual machine, the JVM, our method discovers new instructions and formats producing good compression. For the benchmarks BYTEmark, and the JDK 1.0.2 library, the factor of compression is 60%.

These compression methods are combined with tools to generate automatically two types of decoders for fast execution without decompression. We apply several benchmarks on these decoders, running on two architecturally different processors, for virtual machines adapted for the programming languages Scheme and Java. The speeds obtained demonstrate the viability of the approach. For several benchmarks Java ( BYTEmark ), the slowdown is less than 10%.

## TABLE DES MATIÈRES

---

Liste des Figures	iv
Liste des Tables	vi
<b>Chapitre 1: Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Travaux similaires . . . . .	3
1.2.1 Compression de programmes machines . . . . .	4
1.2.2 Compression de programmes interprétés . . . . .	6
1.2.3 Travaux théoriques . . . . .	11
1.2.4 Résumé des travaux similaires . . . . .	12
1.3 Techniques conventionnelles de décodage . . . . .	12
1.4 Vue générale du processus de génération d'une machine virtuelle . . . . .	14
1.4.1 Génération des dictionnaires de formats et de macros . . . . .	14
1.4.2 Génération de la structure du décodeur . . . . .	15
1.4.3 Génération des machines virtuelles . . . . .	17
1.5 Processus de compression d'un programme virtuel . . . . .	17
1.6 Classes de machines virtuelles . . . . .	19
1.7 Compaction binaire et alignements . . . . .	20
1.7.1 Changement du flot d'exécution <i>versus</i> compaction . . . . .	21
1.8 Mesure d'entropie des programmes . . . . .	22
1.8.1 Éléments de compression de données . . . . .	23
<b>Chapitre 2: Création des instructions virtuelles</b>	<b>25</b>
2.1 Méthodes conventionnelles de codification . . . . .	26
2.2 Attribution de codes opérationnels par codes de Huffman . . . . .	26
2.3 Construction de la base des macro-instructions . . . . .	29
2.3.1 La forme générale des macro-instructions . . . . .	30
2.3.2 La division en blocs élémentaires . . . . .	30
2.3.3 L'algorithme de recherche de séquences répétitives . . . . .	31
2.3.4 Post-traitement de la base des séquences . . . . .	34
2.4 Construction de la base des formats . . . . .	36

2.4.1	Spécialisation de type format . . . . .	38
2.4.2	La complétion d'un ensemble de formats . . . . .	39
2.4.3	Les formats généraux . . . . .	42
2.4.4	Spécialisation de type valeur . . . . .	44
2.4.5	Gain en espace d'un format . . . . .	44
2.4.6	Processus de construction de la base des formats. . . . .	47
2.5	Sélection des formats . . . . .	49
2.6	Sélection des macro-instructions . . . . .	50
2.7	Algorithme de création des instructions virtuelles . . . . .	51
2.8	Compression des programmes . . . . .	52
<b>Chapitre 3: Le décodage des instructions virtuelles</b>		<b>55</b>
3.1	Introduction aux techniques de décodage Huffman . . . . .	55
3.2	Stockage et lecture des instructions en mémoire . . . . .	55
3.3	Les décodeurs automates . . . . .	56
3.3.1	La construction de la structure d'un décodeur automate . . . . .	57
3.3.2	Espace mémoire de l'automate . . . . .	62
3.3.3	Réduction de la taille de l'automate . . . . .	63
3.3.4	Contraintes sur les codes opérationnels pour réduire la taille des automates . . . . .	64
3.3.5	Réduction de la taille d'un automate par le prédicat $Iz$ . . . . .	65
3.3.6	Génération du code C d'un décodeur automate . . . . .	65
3.4	Les décodeurs canoniques . . . . .	68
3.4.1	Les décodeurs de codes canoniques . . . . .	70
3.4.2	La génération de la structure d'un décodeur canonique . . . . .	70
3.4.3	Génération du code C d'un décodeur canonique . . . . .	79
3.5	L'accès efficace au programme en mémoire centrale . . . . .	81
3.5.1	Trois formes d'accès au programme . . . . .	82
3.6	Extraction des arguments . . . . .	85
3.7	Génération des instructions de branchement . . . . .	87
3.8	Implantation des macro-instructions . . . . .	88
3.9	Benchmark des décodeurs . . . . .	88
3.9.1	Les machines simples utilisées pour les benchmarks . . . . .	88
3.9.2	Performance des décodeurs canoniques et automates . . . . .	89

<b>Chapitre 4: Application à la machine JVM</b>	<b>95</b>
4.1 La machine virtuelle JVM de Java . . . . .	95
4.2 Compression sans macro-instructions . . . . .	96
4.3 Compression avec macro-instructions . . . . .	98
4.4 Vitesses d'exécution de benchmarks Java compressés . . . . .	109
4.4.1 Modification de l'implantation Harissa . . . . .	109
4.4.2 Les benchmarks Java BYTEmark . . . . .	110
4.4.3 Vitesses d'exécution sans macro-instructions . . . . .	110
4.4.4 Vitesses d'exécution avec macro-instructions . . . . .	116
4.5 Conclusion pour ce chapitre . . . . .	117
<b>Chapitre 5: Machina, une machine virtuelle générale</b>	<b>120</b>
5.1 La structure générale de Machina . . . . .	120
5.2 La compilation de Scheme vers Machina . . . . .	121
5.3 L'édition des liens . . . . .	124
5.4 La codification binaire des programmes compressés Machina . . . . .	124
5.5 L'instruction JSR . . . . .	125
5.6 Les programmes benchmarks Scheme . . . . .	127
5.7 Compression des benchmarks Scheme sans macro-instructions . . . . .	127
5.7.1 Vitesse d'exécution des benchmarks compressés . . . . .	130
5.8 Conception d'instructions virtuelles pour Scheme . . . . .	133
5.8.1 Compression des benchmarks Scheme avec macro-instructions . . . . .	136
5.8.2 Comparaison des facteurs de compression avec le système BIT . . . . .	136
5.8.3 Vitesse d'exécution des benchmarks avec macro-instructions . . . . .	138
<b>Chapitre 6: Travaux futurs et conclusion</b>	<b>141</b>
6.1 Conclusion . . . . .	141
6.2 Travaux futurs . . . . .	142
6.2.1 Décodeur hybride automate/canonique . . . . .	142
6.2.2 Modèles de Markov d'ordre supérieur . . . . .	142
6.2.3 Application au langage C . . . . .	142
<b>Annexe</b>	<b>144</b>
<b>Références</b>	<b>148</b>

## LISTE DES FIGURES

---

1.1	Génération des dictionnaires de formats et de macro-instructions .	16
1.2	Processus de génération de la structure d'un décodeur . . . . .	17
1.3	Processus de génération du code C et d'une machine virtuelle exécutable pour programmes compressés . . . . .	18
1.4	Processus de compression d'un programme . . . . .	19
2.1	L'arbre canonique de Huffman pour Zipf-20 . . . . .	28
2.2	Algorithme de création de la base de séquences . . . . .	33
2.3	Génération des séquences $s$ de longueur $i + 1$ . . . . .	34
2.4	Un ensemble de formats $P$ et sa complétion $P^c$ . . . . .	40
2.5	Algorithme de complétion des formats . . . . .	43
2.6	Processus de création d'un dictionnaire de formats $D_f$ . . . . .	49
2.7	Algorithme de création de versions spécialisées d'instructions, la sélection de macro-instructions et d'attribution de codes opérationnels	53
3.1	Technique de base d'accès au programme compressé . . . . .	56
3.2	Formes générales des séquences . . . . .	58
3.3	Algorithme de construction des automates décodeurs . . . . .	59
3.4	Codes opérationnels ayant des automates compacts . . . . .	63
3.5	Longueurs moyennes de Zipf-200 avec contraintes . . . . .	65
3.6	Instructions combinées par <i>concaténation</i> pour un décodeur automate	67
3.7	Arbres canoniques croissant et décroissant . . . . .	69
3.8	Décodeur très compact pour un code canonique croissant . . . . .	72
3.9	Algorithme de construction des décodeurs canoniques . . . . .	75
3.10	Deux arbres décodeurs $A_1$ et $A_2$ pour Zipf-200 . . . . .	78
3.11	Structure générale, en C, d'un décodeur de codes canoniques . . . .	79
3.12	Code assembleur du Pentium pour le chargement de quatre octets .	82
3.13	Exemple d'accès mémoire forme-b accompagné d'un décodeur automate . . . . .	84
3.14	Exemple de l'accès mémoire forme-c accompagné d'un décodeur canonique . . . . .	86
3.15	Instructions virtuelles des machines benchmarks . . . . .	90

3.16	Un décodeur simple de code-octet non compressé . . . . .	91
5.1	Le calcul de l'adresse de retour pour JSR . . . . .	125
1	Le programme fib en Scheme . . . . .	145
2	Le code assembleur du programme fib . . . . .	146
3	Le code Machina du programme fib . . . . .	147

## LISTE DES TABLES

---

1.1	Résumé des travaux similaires . . . . .	13
2.1	Exemples de macros pour deux machines virtuelles . . . . .	30
2.2	Instructions ne pouvant être au milieu d'une macro-instruction . .	31
3.1	Exemple de mnémoniques et de leur code canonique . . . . .	69
3.2	Codes Zipf-200 et les temps de décodage pour deux décodeurs . .	77
3.3	Temps absolu du décodeur code-octet de Zipf-20, pour trois machines virtuelles sans paramètres, sur deux processeurs . . . . .	89
3.4	Temps absolu du décodeur code-octet de Zipf-20, pour trois machines virtuelles avec paramètres, sur deux processeurs . . . . .	89
3.5	Temps relatif de décodage de Zipf-20 par des décodeurs automates, selon trois formes d'accès mémoire, pour trois machines virtuelles sans paramètres, sur deux processeurs . . . . .	93
3.6	Temps relatif de décodage de Zipf-20 par trois décodeurs canoniques, pour trois machines virtuelles sans paramètres, sur deux processeurs . . . . .	93
3.7	Temps relatif de décodage de Zipf-20 par des décodeurs automates, selon trois formes d'accès mémoire, pour trois machines virtuelles avec paramètres, sur deux processeurs . . . . .	94
3.8	Temps relatif de décodage de Zipf-20 par trois décodeurs canoniques, selon trois formes d'accès mémoire, pour trois machines virtuelles avec paramètres, sur deux processeurs . . . . .	94
4.1	Tailles des codes opérationnels et des opérandes pour le code-octet JVM des bibliothèques de JDK 1.1 . . . . .	97
4.2	Facteur de compression Huffman, sur les codes opérationnels, pour les bibliothèques de JDK 1.1 . . . . .	98
4.3	Fréquences des mnémoniques pour l'ensemble des groupes awt, io, lang, net, security, sql, util et text . . . . .	99
4.4	Nouveaux formats générés par l'algorithme de spécialisation pour les groupes AWT, IO, LANG, NET, SECURITY, SQL, TEXT et UTIL . . . . .	100

4.5	Nouveaux formats et codes opérationnels de 82 instructions JVM .	101
4.6	Facteur de compression des opérandes en utilisant les formats de la table 4.4 . . . . .	102
4.7	Facteur de compression en utilisant les formats de la table 4.4 et les codes de Huffman . . . . .	102
4.8	Facteur de compression, sans macros, des benchmarks BYTEmark en utilisant les formats de la table 4.5 et les codes de Huffman . .	103
4.9	Nombre de séquences de la base de macro-instructions, selon leur longueur ; au total, il y a 484 séquences . . . . .	104
4.10	Une partie du dictionnaire préliminaire de macro-instructions . . .	104
4.11	Une partie du dictionnaire final de macro-instructions . . . . .	106
4.12	Une partie des 362 instructions et de leur code opérationnel de la machine virtuelle avec macro-instructions et nouveaux formats . .	107
4.13	Facteur de compression de la librairie JDK 1.0.2 en utilisant des macros et formats de la table 4.12 et les codes de Huffman . . . . .	108
4.14	Facteur de compression des benchmarks BYTEmark en utilisant des macros et formats de la table 4.12 et les codes de Huffman . .	108
4.15	Description des dix benchmarks BYTEmark . . . . .	111
4.16	Profils dynamiques des benchmarks BYTEmark Java . . . . .	112
4.17	Temps absolus et relatifs des benchmarks Java BYTEmark, sans macros, pour deux décodeurs canoniques, sur deux processeurs, par l'implantation Harissa modifiée . . . . .	113
4.18	Temps relatifs des benchmarks Java BYTEmark, sans macros, pour trois décodeurs automatés, sur deux processeurs, par l'implantation Harissa modifiée . . . . .	114
4.19	Temps théorique moyen de décodage des benchmarks BYTEmark Java pour deux décodeurs canoniques . . . . .	114
4.20	Structures des sept décodeurs canoniques . . . . .	116
4.21	Temps relatifs des benchmarks BYTEmark compressés avec macros, pour sept décodeurs canoniques, sur Pentium, par l'implantation Harissa modifiée . . . . .	118
4.22	Temps relatifs des benchmarks BYTEmark compressés avec macros, pour sept décodeurs canoniques, sur Sparc, par l'implantation Harissa modifiée . . . . .	118

5.1	Les instructions de base de Machina . . . . .	122
5.2	Les formats de base de Machina . . . . .	123
5.3	Les benchmarks utilisés pour l'expérimentation . . . . .	126
5.4	Les fréquences des mnémoniques de base de Machina pour l'ensemble des benchmarks de la table 5.3 . . . . .	127
5.5	Les formats et fréquences des instructions de base pour les bench- marks . . . . .	128
5.6	Les codes opérationnels des nouveaux formats et des instructions de base standard . . . . .	129
5.7	Tailles absolues (code-octet, non compressé) et relatives (compressé) des benchmarks Scheme sans macro-instructions . . . . .	130
5.8	Temps d'exécution des benchmarks Scheme compressés, sans ma- cros, avec décodeurs canoniques pour le processeur Pentium . . . . .	131
5.9	Temps d'exécution des benchmarks Scheme compressés, sans ma- cros, avec décodeurs canoniques pour le processeur Sparc . . . . .	131
5.10	Temps d'exécution des benchmarks Scheme compressés, sans ma- cros, avec décodeurs automates pour le processeur Pentium . . . . .	132
5.11	Temps d'exécution des benchmarks Scheme compressés, sans macro- instructions, avec décodeurs automates pour le processeur Sparc . . . . .	132
5.12	Une partie du dictionnaire final de macro-instructions . . . . .	134
5.13	Facteur de compression des benchmarks Scheme . . . . .	136
5.14	Tailles des benchmarks Scheme pour BIT et Machina . . . . .	138
5.15	Temps d'exécution en seconde des benchmarks pour BIT, Gambit et Machina sur Pentium . . . . .	138
5.16	Temps relatifs d'exécution des benchmarks Scheme, compressés avec macro-instructions, pour des décodeurs canoniques forme-c sur pro- cesseur Pentium . . . . .	139
5.17	Temps relatifs d'exécution des benchmarks Scheme, compressés avec macro-instructions, pour des décodeurs canoniques forme-c sur pro- cesseur Sparc . . . . .	139
5.18	Temps relatifs d'exécution des benchmarks Scheme, compressés avec macro-instructions, pour des décodeurs canoniques forme-b sur pro- cesseur Pentium . . . . .	140

5.19 Temps relatifs d'exécution des benchmarks Scheme, compressés avec macro-instructions, pour des décodeurs canoniques forme-b sur processeur Sparc . . . . .	140
---	-----

## REMERCIEMENTS

---

Mille remerciements,

À Marc Feeley, mon directeur de recherche, pour sa patience, son aide technique et financière. Il a poussé sa générosité jusqu'à me laisser occuper son bureau personnel pour terminer la réalisation de ce Ph.D. ;

Aux collègues du laboratoire de parallélisme, en particulier à Danny Dubé, pour avoir conversé sur de nombreux détails hétéroclites du sujet de cette thèse et de son système BIT ;

À Ericsson, en particulier à Joe Armstrong et au laboratoire de Suède, pour leur aide financière opportune ;

À Céline, pour son grand amour, et pour m'avoir accompagné dans les deux premiers tiers de la montée du col ;

À Maria, avec ses paroles convaincantes, par une belle soirée d'été, de clore ce Ph.D. ;

À Grace, avec ses encouragements de longue distance ;

À saint Jean et à ses 153 gros poissons ...

## Chapitre 1

### INTRODUCTION

---

*Research is to see what everybody else has seen, and to think what nobody else has thought.*

*Albert Szent-Gyorgyi*

Ce travail aborde le problème de la création d'instructions virtuelles afin de former des programmes très compacts, directement exécutables en un temps raisonnable. Elle le fait, en majeure partie, sur la codification des instructions et, à un degré moindre, sur la sémantique. Ma thèse est qu'il est possible de concevoir automatiquement des instructions complexes à partir d'instructions de base ; et que celles-ci peuvent être codées en utilisant des codes de Huffman pour les codes opérationnels, et des champs de bits de longueur non multiple de huit ; le tout stocké en mémoire sans tenir compte des frontières d'octets.

La conception d'une machine virtuelle est un processus complexe. La tâche du concepteur consiste à définir un ensemble de services utiles au besoin d'un langage spécifique de programmation. L'une des sous-tâches est la conception des instructions virtuelles. Dans la plupart des cas, celles-ci sont choisies pour faciliter l'implantation de la compilation du langage source. L'élaboration de ces instructions se subdivise en plusieurs activités :

1. la spécification de la structure générale : pile, monceau, variables globales, variables locales, gestion de la mémoire, etc. ;
2. la définition de la sémantique des instructions ;
3. la codification des instructions sous forme de chaînes de bits : choix des codes opérationnels, du nombre de bits des paramètres, la spécialisation de certaines instructions pour des arguments fixes et des formats compacts.

Notre approche touche principalement aux activités 2 et 3. Nous n'abordons pas la tâche de définir la structure générale d'une machine virtuelle.

Nous proposons la combinaison de quatre méthodes afin de rendre un programme compact. L'une des méthodes utilise des macro-instructions, c'est-à-dire des instructions faisant référence à des séquences d'instructions de base. Ces macro-instructions sont implantées dans la machine virtuelle. Cette méthode est reliée à la tâche 2, car le choix de macro-instructions

définit de nouvelles instructions. Une deuxième méthode utilise des codes opérationnels de différentes longueurs à préfixe unique, plus précisément les codes de Huffman. La construction de ces codes est fondée sur les fréquences des instructions. Une troisième méthode spécialise les instructions afin de diminuer l'espace occupé par les opérandes. Ces activités sont similaires à la tâche 3. Assurément, il faut ajouter une quatrième méthode pour rendre effectives les trois premières, soit le non-alignement des instructions sur une frontière d'octet. Ces quatre méthodes sont mutuellement avantageuses pour les raisons suivantes.

La codification par des codes de longueurs différentes à préfixe unique permet la compression des programmes, en attribuant des codes opérationnels courts aux instructions les plus fréquentes. Toutefois, le temps de décompression peut s'avérer être substantiel relativement au temps d'exécution des instructions. La création de macro-instructions permet de remplacer des séquences d'instructions fréquentes par un seul code opérationnel. Ce remplacement a deux impacts positifs : le temps d'exécution des instructions augmente par rapport au temps de décodage, et le code est plus compact. La création de nouveaux formats courts permet non seulement de réduire l'espace mais augmente la rapidité, car l'extraction de longs arguments ralentit le processus de décodage.

Ces méthodes sont accompagnées d'algorithmes de génération de deux types de décodeur, automate et canonique, pour une exécution rapide sans décompression préalable.

La compression des programmes apporte plusieurs difficultés d'exécution, notamment le décodage des instructions, l'extraction de leurs arguments et le changement du flot d'exécution. Il est important de fournir des outils et des techniques pour résoudre ces difficultés. Nous le faisons par des algorithmes de conception de nouvelles instructions, de nouveaux formats, et par deux algorithmes de construction de décodeurs. Ainsi, la création d'instructions complexes, de nouveaux formats et d'un décodeur devient un processus automatisé où le concepteur intervient à un niveau élevé. Essentiellement, celui-ci spécifie un ensemble de programmes échantillons et l'espace à consacrer au décodeur.

Le tout s'inscrit dans un objectif de développement d'outils, utilisables sous le contrôle du concepteur, produisant automatiquement du code C pour implanter des machines virtuelles.

Les résultats de nombreux tests expérimentaux démontrent la viabilité des méthodes en produisant des machines virtuelles adaptées aux langages de programmation Scheme et Java<sup>MD</sup>.

## 1.1 Motivation

La taille des programmes est devenue une préoccupation avec l'avènement des processeurs de haute performance, des programmes transférés sur réseau longue distance et des systèmes embarqués à mémoire réduite.

Certains systèmes matériels ont une forte contrainte sur leur mémoire centrale. Par exemple, la *Java Ring* [Cur98] a 6Ko de RAM et 32Ko de ROM. Pour permettre de stocker des programmes complexes, c'est l'approche d'une machine virtuelle qui est utilisée (une JVM réduite pour ce système). Cette approche peut, effectivement, donner des programmes très compacts.

Si les programmes à exécuter sont dynamiquement chargés, une décompression préalable utiliserait le peu d'espace mémoire RAM disponible. L'exécution sans une phase de décompression est donc très utile pour ces systèmes.

Dans le cas où les programmes ne sont pas chargés dynamiquement, c'est la mémoire ROM qui est utilisée pour les stocker. Cette situation requiert la conception d'une machine virtuelle accompagnée des programmes compressés. Il est donc important, dans ce cas, de tenir compte, à la fois, de la taille de l'interprète de la machine virtuelle et des programmes compressés.

Le transfert de programmes sur réseau, pour une exécution locale, requiert une technologie de machines virtuelles pour permettre la portabilité et la sécurité. Ces programmes sont transférés sous une forme intermédiaire, souvent sous la forme de code-octet (byte-code, p.e. JVM). La diminution de la taille des programmes permet un transfert plus rapide. C'est une situation où la taille de la machine virtuelle est moins importante que la taille des programmes à transférer. Dans ce cas, le concepteur est prêt à augmenter la taille de la machine virtuelle afin de diminuer les tailles des programmes.

Il est donc important de concevoir une approche générale où ces différentes situations peuvent être abordées ; c'est-à-dire la génération de machines virtuelles dont la taille peut être augmentée pour réduire les tailles des programmes compressés, ou pour les cas dont la taille de la machine virtuelle doit être maintenue basse pour obtenir une taille totale basse.

D'autre part, la conception des instructions pour une machine virtuelle est souvent effectuée sans mesure précise de l'impact de leur codification. Ces instructions sont souvent choisies pour faciliter l'implantation de la compilation. Conventionnellement, leur codification est faite à la main. Il y a intérêt à automatiser ce processus pour en faciliter la modification et l'évaluation. Notamment, le décodeur d'instructions.

La codification code-octet est limitée à une codification des codes opérationnels sur un octet : de là une limite de 256 instructions. L'utilisation de codes opérationnels de différentes longueurs permet d'éviter cette limite parfois gênante.

## 1.2 Travaux similaires

Les travaux similaires ont été divisés en trois sections majeures : l'une pour les programmes compressés s'exécutant au niveau matériel, une autre pour les interprètes logiciels, et

une troisième section pour les travaux théoriques. Une quatrième section résume le contenu de ces travaux.

### 1.2.1 Compression de programmes machines

Plusieurs travaux appliquent des techniques de compression sur des programmes exécutables dont la décompression s'effectue à l'exécution au niveau matériel [WC92, KW94, KW95, ARM95, BWN97, BNW98, LW98, LBCM97, ACCP98]. Bien que ces travaux se situent au niveau matériel, les techniques employées s'apparentent aux méthodes utilisées au niveau logiciel. De plus, les résultats obtenus sont des indications des possibilités. Voici une description concise de leurs techniques et résultats.

Wolfe, Kozuch et Chanin [WC92, KW94, KW95] ont analysé l'utilisation du code de Huffman [Huf52] 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 compressé par un code de Huffman, préétabli une fois pour toute (modèle statique). (Les auteurs ont utilisé des lignes d'antémémoire de 32 octets.) 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 voit pas le code compressé, mais bien les octets décompressés. Pour adresser correctement la mémoire centrale lors des branchements, une table de correspondance entre les adresses des blocs compressés et non compressés est utilisée (une LAT, Line Address Table). Leur facteur de compression est de 70%<sup>1</sup> pour le Mips R2000. De plus, dans [WC92] ce facteur diminue quelque peu en utilisant d'autres architectures. Il faut noter que ces travaux s'adressent à des processeurs RISC, le code des programmes étant au départ assez volumineux. L'une des conclusions des auteurs est qu'il serait probablement plus avantageux d'appliquer la compression de code au niveau du compilateur.

De même, pour les travaux [LW98, BNW98, BWN97, BJS97] le décodage des instructions est effectué entre la mémoire centrale et l'antémémoire. Les auteurs de [BNW98, BWN97] ont considéré en détail la réalisation d'un décodeur de Huffman, à modèle statique, au niveau matériel. Une attention particulière a été apportée au temps de décodage. Le facteur de compression est essentiellement celui rapporté par [KW94], soit environ 73%. Lekatsas et Wolf [LW98] utilisent plutôt une méthode de compression en utilisant des séquences de codes opérationnels adjacents ou la spécialisation d'une instruction (cette méthode porte le nom de SADC, « *Semiadaptive Dictionary Compression* »; une autre méthode combinant le codage arithmétique et des modèles de Markov (SAMC) s'avère moins performants). Les instructions

---

1. Dans cette thèse le facteur de compression est exprimé par le rapport *taille compressé / taille originale*.

machines, soient des processeurs Mips ou x86, sont préalablement divisées en plusieurs flots de bits, séparant ainsi les codes opérationnels des registres, opérandes immédiats, etc. Dans le cas de Mips, il y a quatre flots : op-codes, registres, immédiats 16 bits, immédiats 26 bits. Un dictionnaire de séquences de codes opérationnels est construit. Au plus, 256 codes sont utilisés pour référencer les instructions de base ou les nouvelles instructions formées par des séquences d'instructions de base. Finalement, le flot de codes opérationnels est codé à la Huffman. La décompression et la reconstruction des instructions sont effectuées lors de leur chargement dans l'antémémoire. Cette méthode appliquée au processeur Mips obtient le meilleur facteur de compression, similaire au logiciel `compress` de UNIX, pour les dix-huit programmes benchmarks Spec95 ; soit un facteur moyen de 50%. Pour le processeur x86 le facteur est de 60%.

Les travaux [DEM99, CM99, LDK99] compressent des programmes exécutables tout en conservant la capacité d'exécution sans décompression préalable (voir aussi [LBCM97]). Pour ces trois travaux, la méthode de base utilisée est le remplacement de séquences d'instructions par des appels à des procédures provenant d'abstraction de séquences. Naturellement, la difficulté majeure est la découverte des séquences fréquentes pouvant être avantageusement abstraites. Cooper et McIntosh appliquent aussi un remplacement par « cross jumping » où aucune abstraction n'est effectuée (un « cross jumping » est la transformation d'une séquence de la forme  $M; \text{goto } L \dots M; L; X$  ; en  $\text{goto } Lb \dots Lb; M; X$ ). Toutefois, cette méthode ne semble pas très avantageuse. Le facteur de compression obtenu se situe au environ de 95%, ce qui est finalement assez peu. Debray *et al.* utilisent des méthodes de transformation plus poussées (déplacement d'instructions) sur le code intermédiaire pour identifier des séquences équivalentes. Ils rapportent un facteur de compression moyen de 78% pour l'ensemble des huit programmes SpecInt95. Liao *et al.* [LDK99] ne rapportent pas de meilleurs facteur de compression.

L'architecture Thumb [ARM95, Tur95] a l'objectif de diminuer l'espace occupé par les instructions machines d'un processeur RISC. La technique principale est l'ajout d'instructions, où le nombre de bits pour la spécification des registres et des opérandes immédiats est réduit par rapport au nombre de bits nécessaires. Ce travail s'inscrit dans le contexte des processeurs pour système embarqué. Le facteur de compression, par rapport à l'architecture originelle, est d'au maximum 60% avec une perte de vitesse de 20%.

Lefurgy *et al.* [LBCM97] utilisent un dictionnaire de séquences d'instructions avec décompression à l'exécution au niveau matériel. La construction du dictionnaire de séquences se fait après la compilation et peut inclure des séquences d'une seule instruction ce qui, selon les auteurs, augmente la compression par rapport aux travaux de [Lia96]. Une implantation au niveau matériel décode les références au dictionnaire pour fournir les instructions machines au décodeur originel du processeur. Les meilleurs facteur de compression sont obtenus

en utilisant des codes de référence ayant un multiple de 4 bits. Le processeur PowerPC obtient le meilleur facteur de compression ; il est d'une moyenne de 61% pour les huit programmes benchmarks *SpecInt95*.

Une équipe de chercheurs [KMH<sup>+</sup>98, GB98, IBM98], du groupe de micro-électronique chez IBM, a mis au point un système de compression de programmes exécutables pour le PowerPC. Pour compresser un programme, les instructions de 32 bits sont divisées en deux parties de 16 bits. Six formats sont utilisés pour coder la partie haute, et six autres formats pour la partie basse. La partie haute, contenant toujours le code opérationnel, a des formats de longueurs 5, 7, 9, 10, 11 et 19 bits. Par exemple, le format de 5 bits a un champ de deux bits identifiant sa longueur, et trois bits pour coder un symbole. Chaque symbole est associé à un mot de 16 bits. La raison pour diviser les instructions en deux parties est simple : statistiquement, certains mots de 16 bits sont beaucoup plus fréquents dans la partie basse que dans la partie haute, et vice versa. La décompression se fait au niveau matériel, entre l'antémémoire et la mémoire centrale. Une table associant les codes aux mots de 16 bits est stockée dans ce décompresseur. Cette table est construite par analyse statistique sur le programme à compresser. Des facteurs de compression de 60% sont relativement fréquents. Le temps d'exécution des programmes n'augmente pas substantiellement et peut même diminuer.

Les travaux de Araujo *et al.* [ACCP98] sont les plus élaborés ; ils obtiennent un facteur moyen de compression de 43% pour les benchmarks *SpecInt95* sur le processeur Mips R2000. C'est, sans contredit, le meilleur résultat des travaux publiés. De tels résultats s'obtiennent par une combinaison de plusieurs techniques. La première consiste à séparer les codes opérationnels des opérandes pour mieux déceler la présence de répétitions de patrons. Les patrons les plus fréquents sont utilisés pour former un dictionnaire. Le code compressé réfère à ce dictionnaire par des indices de longueurs variables, construit par codage de Huffman.

### 1.2.2 Compression de programmes interprétés

Cette section s'intéresse aux travaux similaires effectués au niveau logiciel.

La compression du code-octet d'un interprète a été abordée par Wilner pour le langage SDL de l'ordinateur Burroughs B1700 [Wil72]. Deux techniques de compression ont été comparées : une méthode *ad hoc* utilisant des champs de 4, 6 et 10 bits, et la codification de Huffman. La première méthode produit, pour un programme fixe appelé MCP (*Master Control Program*), un facteur de compression de 61%, et la codification de Huffman un facteur de 57%. En général, une instruction SDL utilise 14 bits pour spécifier son opérande : un entier pour indiquer le niveau sur la pile (16 niveaux possibles), et un indice ( $2^{10}$  possibles) de la variable pour ce niveau. Une analyse statistique a montré qu'il serait préférable de créer

des instructions spécialisées pour certains cas, résultant en des formats de 8, 11, 13 ou 16 bits. Lors de l'interprétation, le temps de décodage pour la première méthode augmente de 2.6%, et de 17% pour la codification de Huffman. Il faut noter que le décodage se fait par microcode ; un bit à la fois. Les auteurs concluent qu'il est possible, en utilisant des codes de Huffman, d'obtenir des facteurs de compression entre 25% et 75%, selon le langage de programmation.

Zastre [Zas95] réutilise la technique de Fraser *et al.* [FMW84] et Marks [Mar80] qui est essentiellement la recherche de séquences d'instructions similaires répétitives (macros). Marks a tenté de découvrir des sous-routines de niveau supérieur par l'analyse du code machine, mais sans grand succès. C'est plutôt dans la recherche de séquences répétitives au niveau machine qu'une diminution de 15% en espace, pour l'IBM-360, a été obtenue. Fraser mentionne la technique pour découvrir les séquences identiques répétitives pour le PDP-11 : l'arbre suffixe. La diminution en espace obtenue est de 7%. Zastre améliore la technique de Fraser en recherchant des séquences similaires, à l'aide de l'arbre suffixe, introduisant ainsi des paramètres. Les diminutions en espace demeurent modestes, de 1 à 9%. La méthode de l'arbre suffixe, pour découvrir des macros, ne semble pas convaincante comme méthode ; elle est lente et complexe.

Piumarta et Riccardi de [PR98] ont comme objectif de réduire le temps d'exécution d'un programme d'une machine virtuelle pour la technique du code-adresse (« threaded-code »). Lors de la traduction du code virtuel vers le code-adresse, de nouvelles instructions sont créées. Celles-ci sont des macros correspondant à des séquences d'instructions de base. Un bloc élémentaire devient une macro, et les instructions du bloc élémentaire sont concaténées pour former une liste d'instructions à exécuter, sans décodage, par la machine virtuelle. Ce processus est effectué au chargement des instructions par la machine virtuelle. Le deuxième aspect positif de cette technique est la réduction de l'espace du programme, si des blocs élémentaires se répètent. Cette méthode est une extension proposée par Bell [Bel73] : où chaque instruction virtuelle était traduite en une adresse sans création de macros.

Le travail de Pugh [Pug99] s'applique 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 compresser, à la fois, le code-octet et la table des constantes. Pour le code-octet, la technique générale est de séparer les codes opérationnels des opérandes en deux flots d'octets (voir aussi [Fra99]). Ces flots sont compressés séparément par différentes méthodes dont celle utilisée par `gzip`. Des résultats impressionnants sont obtenus par une technique de prédiction de codes opérationnels en utilisant le contexte de l'état de la pile. L'auteur rapporte des facteurs de compression de 17% à 49% par rapport au format `Jar`. Notez que

ces très bons facteurs de compression sont possibles à cause de la compression du bassin de constantes contenant de nombreuses informations redondantes, dont une multiplicité de sous-chaînes identiques. D'autre part, les classes compressées ne peuvent être exécutées avant décompression.

La recherche de macros pour améliorer la rapidité a aussi été abordée par Proebsting [Pro95]. Ce travail, effectué dans le cadre d'un interprète pour le langage C, rapporte des améliorations de temps d'exécutions ainsi que des codes compilés plus compacts. L'heuristique de recherche de macros débute avec 109 instructions de base pour augmenter cet ensemble à un maximum de 256 instructions ; un maximum de 147 macros peuvent ainsi être créées. Une macro est essentiellement une suite d'instructions de base ou de macros. Les 109 instructions de base correspondent aux instructions apparaissant dans les arbres abstraits produits par le compilateur `lcc`. Une fois les macros construites, un interprète est généré. Les instructions de base ont leur propre implantation en langage machine, et chaque macro est implantée en concaténant les implantations des instructions de base la composant. Ainsi, une macro remplace le décodage de chacune des instructions de base par un seul décodage. Ceci diminue le temps d'exécution et permet un code plus compact.

Proebsting et Fraser ont étendu le travail de [FP95]. Le contexte est semblable : à partir d'un programme source C un compilateur produit un interprète à pile pour les opérations produites par `lcc`. L'exécution de cet interprète, avec le code-octet, émule l'exécution du programme. Un détail technique diffère : l'interprète généré est en C et non en assembleur. Pour faciliter l'utilisation des bibliothèques systèmes, chaque fonction C compilée en code-octet est préfixée d'un prologue en langage machine. Le facteur de compression du code est approximativement de 50%. Le résultat négatif majeur est le temps d'exécution qui est 20 fois moins rapide que le code compilé. C'est deux fois plus lent que plusieurs interprètes C.

Wegdam [Weg98] applique les techniques de Proebsting et Fraser [FP95] : langage source C, génération d'un interprète écrit en C, utilisation de la forme intermédiaire de `lcc` comme point de départ. Les résultats obtenus sont pour le processeur R3000 de Mips. L'interprète généré utilise une pile pour manipuler les opérandes. Le programme interprété est sous la forme de code-octet, donc un maximum de 256 instructions. L'idée centrale est de générer de nouveaux arbres à l'aide de `lburg` couvrant des formes répétitives du programme généré. Les meilleurs résultats mentionnés sont des facteurs de compressions variant de 30% à 36%, où les interprètes générés ne peuvent adéquatement exécuter qu'un seul programme. Des facteurs de 41% à 61% sont obtenus si un ensemble fixe de nouvelles instructions est utilisé. L'auteur mentionne des essais de compression du code-octet à l'aide de codes de Huffman, mais l'augmentation des facteurs de compression ne dépassait pas 2%. Ainsi, aucune méthode de décodage rapide des codes de Huffman n'a été conçue.

Lars Raeder *et al.* [CSCM98] appliquent une méthode de création de macros pour com-

presser le code-octet de programmes Java. Cette méthode consiste à construire de nouveaux codes opérationnels pour des séquences fréquentes de code-octet ; 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 quantité 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 machine JVM (c'est sans compter les 23 instructions « quick » et l'instruction `breakpoint`). Dans le cas de la machine virtuelle JavaCard, 152 codes opérationnels sont disponibles. Toutefois, pour les tests rapportés, il y a toujours au moins 60 macros, et dans certains cas, plus de 160 ; dans ces cas les codes opérationnels peuvent avoir deux octets. Les auteurs considèrent que les techniques de compression Huffman ou Lempel-Ziv ne peuvent résoudre le problème du peu d'espace mémoire RAM des systèmes embarqués. Ainsi, la codification de Huffman n'a pas été utilisée ou analysée. Des facteurs de compression de 80% à 70% sont atteints pour plusieurs benchmarks. Ils utilisent l'implantation Harissa[MMBC97] pour mesurer la perte de vitesse d'exécution. Celle-ci varie selon les benchmarks utilisés : environ 2% dans le meilleur cas et 27% dans le pire cas.

Ernst *et al.* [EFE<sup>+</sup>97] proposent deux techniques, sans les combiner, pour compresser des programmes C : la première méthode nécessite la décompression avant l'exécution, mais la seconde peut exécuter sans décompression. Le code intermédiaire produit par `lcc` est utilisé dans les deux cas. La première méthode est une combinaison des techniques de Lempel-Ziv et des codes de Huffman. Pour obtenir de très bons facteurs de compression, le code intermédiaire des programmes C, généré par `lcc`, est compressé en séparant les opérandes des opérations. Cette technique permet à la compression de Lempel-Ziv de rencontrer plus fréquemment des répétitions. Ainsi, cette méthode nécessite une décompression avant l'exécution. La seconde méthode utilise essentiellement la spécialisation d'instructions pour certains arguments, et la combinaison de paire de codes opérationnels adjacents. C'est donc des formes particulières de macros. Toutes les instructions sont alignées sur des frontières d'octet. Les résultats des facteurs de compression comparent l'espace occupé par le code intermédiaire compressé, et le code exécutable machine produit pour le processeur Pentium. Ces facteurs varient de 53% à 69%.

Michael Franz et Thomas Kistler [FK97, FK96] ont conçu un système complet pour la compression d'une représentation intermédiaire pour le langage Java et sa compilation. La thèse des auteurs suggère qu'il est préférable de transférer la représentation intermédiaire compressée plutôt que le code-octet Java, car la représentation intermédiaire permet une compilation ayant des optimisations mieux adaptées au processeur, et cette représentation est plus compacte que le code-octet. Le facteur de compression par rapport au code-octet

est de 41%. Tout ce système repose sur la nécessité d'un compilateur et d'une décompression préalable à l'exécution ; un contexte qui n'est pas celui de notre travail. Toutefois, ce travail a le bénéfice de démontrer qu'une représentation intermédiaire a le potentiel d'offrir un facteur de compression plus élevé que le code-octet. Un objectif intéressant, et loin d'être résolu, serait de mettre au point une méthode d'exécution rapide et directe d'une telle représentation intermédiaire compacte.

Le travail pertinent le plus récent, est sans nul doute, celui de Hoogerbrugge *et al.* [HATvdW99]. Leur étude est fort intéressante. Elle s'effectue au niveau logiciel et tente de concurrencer les processeurs à deux modes, où il y a un passage d'un mode d'exécution compressé à non compressé, et vice versa, comme l'ARM Thumb et le MIPS16 [Tur95, Kis97]. Le contexte est celui de la compilation de programmes. Une partie du programme est compressée et le reste ne l'est pas. La partie non-compressée est choisie par le programmeur et devrait correspondre à la partie critique pour la rapidité d'exécution du code. Une machine virtuelle générale à pile est employée pour permettre la génération d'un interprète relativement à la partie du code à compresser. Cette machine a 113 instructions de base. Pour augmenter la compression, de nouvelles instructions, correspondant aux séquences les plus fréquentes du programme à compiler, sont générées dans l'interprète. Quelques autres techniques mineures de compression sont utilisées. Le facteur de compression, par rapport aux 113 instructions de base, est environ 70% pour les huit benchmarks choisis parmi `Speclnt92` et `Speclnt95`. Les macro-instructions ont rarement plus de quatre instructions de base. Pour atteindre le facteur de compression de 70%, environ 150 macro-instructions sont utilisées. Par rapport au code natif, le facteur de compression peut atteindre 20%. La compression s'effectue à la granularité de la fonction. Le ralentissement d'exécution, incluant la partie non compressée, n'est perceptible qu'à partir d'un facteur de compression de 50%. À 20%, le temps d'exécution total augmente en moyenne huit fois par rapport au code natif. Si la granularité de compression est le bloc élémentaire, le facteur de compression peut atteindre 30% avec une perte de temps d'exécution négligeable pour la majeure partie des benchmarks.

Ce travail a été effectué sur un TriMedia1000 ; un processeur VLIW. Pour exploiter son parallélisme, l'interprète généré utilise un décodage en pipeline de profondeur trois. Les auteurs ajoutent à leur conclusion que l'utilisation de codes de Huffman, pour les codes opérationnels, ne ferait qu'augmenter la taille de l'interprète et diminuer sa rapidité. Toutefois, aucun benchmark ou analyse ne vient confirmer cette affirmation. Notez que l'algorithme de construction des nouvelles instructions ne considère que des séquences d'instructions exactes, c'est-à-dire sans paramètres.

### 1.2.3 Travaux théoriques

Certains travaux ont analysé les possibilités de la compression de programmes sans avoir l'objectif d'implantations particulières. Toutefois, ces travaux apportent à la fois des données et des idées de base pertinentes.

Pittman [Pit87] fait une analyse générale des possibilités de compression de code à l'aide d'interprètes. Aucun résultat précis n'est mentionné, mais il rapporte l'utilisation d'interprète dans le milieu industriel pour réduire l'espace occupé par les programmes. La codification de Huffman est évoquée, mais en indiquant que son usage rend le décodeur complexe. Mentionnons un résultat intéressant, attribué à Earle [Ear82] et Turner [Tur86] par quelques expériences pratiques : l'utilisation d'un interprète peut réduire de deux à cinq fois la taille d'un programme codé en langage machine.

Dans le travail de Fraser [Fra99], la technique de séparation en plusieurs flots de bits est la méthode de base utilisée. Ce travail explore l'automatisation de la mise au point de modèles statistiques pour la compression de programmes compilés utilisant cette technique, plus particulièrement sur des formes intermédiaires produites par `lcc`. L'aspect intéressant est l'automatisation de la recherche de bons modèles utilisant différents critères contextuels, dont la hauteur de l'arbre dans la représentation intermédiaire, le type des opérandes, les derniers codes opérationnels, etc.

Eric Hehner [Heh77, Heh76] a exploré la compression de programmes pour l'IBM/360. Ses travaux sont similaires à ceux de Foster et Gonter [FG71] pour le CDC-3600, mais font une analyse théorique plus poussée. L'idée centrale des « codes opérationnels conditionnels » de Foster et Gonter est simple : la probabilité d'occurrence d'une instruction machine dépend des  $n$  dernières instructions. Leur analyse porte, en fait, sur le contexte d'une instruction seulement. Ainsi, une instruction crée un contexte ; ce contexte étant conservée par la machine. Dans le cas de Foster et Gonter, 53 contextes sont utilisés, où plus d'une instruction peuvent utiliser le même contexte. Le code opérationnel, de trois bits, est interprété selon le contexte précédent. Un code est réservé pour une interprétation hors contexte. Ce nombre de trois bits provient d'une analyse statistique effectuée sur trois programmes majeurs, totalisant 37000 instructions. Cette analyse démontre que 74% des instructions successeurs, étant donné un contexte, faisaient partie d'un groupe de sept instructions. Les auteurs présentent un mécanisme simple pour effectuer le décodage au niveau de la micro-programmation, mais sans appliquer aucune implantation réelle. Hehner ne fixe pas la longueur du code opérationnel ; un codage de Huffman est appliqué. De plus, il fait une analyse sur un contexte allant d'une à trois instructions. Pour l'IBM/360, il en résulte une longueur moyenne des codes opérationnels de 2.1, 2.7 et 1.6 bits, respectivement. Toutefois, l'espace nécessaire pour représenter des contextes de trois instructions n'est pas analysé, et cet espace n'est pas

négligeable! Hehner fait aussi l'analyse d'une technique de macros qu'il appelle « *iterative pairing* », car les macros sont construites par la combinaison récursive de paires d'instructions. À partir de 47 instructions de base, 131 nouvelles instructions sont créées. Par codage Huffman, la longueur moyenne des codes opérationnels devient 1.8 bits par rapport aux instructions de base.

Bennett [BS89, Ben87] a fait une étude afin d'automatiser la conception de nouvelles instructions à partir d'un ensemble de base, dans le but de réduire l'espace ou le temps d'exécution. Ce travail ne s'applique pas aux machines virtuelles, mais aux implantations matérielles et utilise principalement deux langages de programmation comme objet d'études : BCPL et POLY. Un ensemble d'instructions de base est amélioré par la création de nouvelles instructions en fixant des paramètres diminuant le nombre de bits des paramètres, ou en combinant deux instructions adjacentes. L'auteur fait une analyse de l'entropie des codes opérationnels, mais aucune compression n'est adoptée. Une étude intéressante, effectuée dans la thèse, est l'utilisation de modèles de Markov d'ordre supérieur pour la compression des codes opérationnels. Toutefois, les instructions générées utilisent toujours un nombre multiple de huit bits. Ainsi, aucune technique de décodage d'instructions compressées n'est analysée.

#### 1.2.4 Résumé des travaux similaires

La table 1.1 expose brièvement les travaux de compression de programmes mentionnés aux sections précédentes.

En résumé, les travaux précédents de compression de programmes ont utilisé des techniques bien connues : codes de Huffman, codage arithmétique, dictionnaire de séquences d'instructions, modèle de Markov, Lempel-Ziv. Si la méthode Lempel-Ziv est utilisée, une décompression préalable à l'exécution est effectuée. Tous ces travaux font une « décompression » lors de l'exécution du code, à l'exception de [Pug99] nécessitant une décompression préalable avant l'exécution. Aucun travail au niveau logiciel, et pour une exécution directe, n'emploie des codes de Huffman ou le codage arithmétique ; dans le cas d'interprète logiciel, c'est la solution du dictionnaire de séquences d'instructions qui est retenue.

### 1.3 Techniques conventionnelles de décodage

Une partie essentielle d'un interprète est le décodage des instructions virtuelles. Un tel décodage peut utiliser une part substantielle, ou négligeable, du temps d'exécution selon la granularité des instructions à émuler. Plus la tâche effectuée prend du temps machine, moins la part du temps de décodage est importante.

Travaux	Niveau	Méthodes	Langages	UCT	Facteur
[KMH <sup>+</sup> 98]	matériel	Statistique	machine	PowerPC	60%
[DEM99, CM99, LDK99]	logiciel	macros	machine	—	78%
[BWN97, BNW98, BJS97]	matériel	Huffman	machine	PowerPC	73%
[LW98]	matériel	Huffman, macros	machine	Mips, x86	50%, 60%
[LBCM97]	matériel	macros	machine	PowerPC	61%
[ACCP98]	matériel	Huffman, macros	machine	R2000	43%
[WC92, KW94, KW95]	matériel	Huffman	machine	R2000	70%
[BS89, Ben87]	théorique	Markov, macros	machine	—	—
[Wil72]	microcode	Huffman	machine	B1700	57%
[Heh77, Heh76, FG71]	théorique	Huffman, macros	machine	IBM-360	—
[HATvdW99]	interprète	macros	C	TM1000	70%
[Pug99]	interprète	gzip	Java	—	17%–49%
[PR98]	interprète	macros	C, CAML	Pentium	—
[Weg98]	interprète	macros	C	R3000	30%–40%
[CSCM98]	interprète	macros	Java	—	70%–86%
[EFE <sup>+</sup> 97]	interprète	Huffman, macros	C	Pentium	53%–69%
[Pro95]	interprète	macros	C	—	—
[FP95]	interprète	macros	C	—	50%

TAB. 1.1 –. Résumé des travaux similaires

Deux techniques de décodage conventionnelles sont bien connues : le code-octet (« byte-code ») et le code-adresse (« threaded-code ») (voir [Kli81] pour des variations).

1. Le code-octet utilise un code opérationnel d'un octet, celui-ci servant d'indice à un vecteur d'adresses. Chacune de ces adresses correspond à l'implantation de l'opération associée au code opérationnel. C'est une méthode plus compacte que le code-adresse, mais moins rapide sur la plupart des processeurs.
2. Le code-adresse consiste à utiliser directement, comme code opérationnel, les adresses des routines des opérations. Cela évite un niveau d'indirection pour atteindre le code implantant l'opération. C'est moins compact que la méthode du code-octet, car, pour la plupart des machines, une adresse mémoire utilise plus d'un octet.

On peut aussi avoir des variations de ces méthodes. Par exemple, pour le code-adresse, l'adresse est remplacée par une instruction d'appel de sous-routine.

Pour les tests expérimentaux présentés dans les derniers chapitres, les vitesses d'exécution des programmes compressés sont comparés à des implantations utilisant la première technique.

#### **1.4 Vue générale du processus de génération d'une machine virtuelle**

Cette section présente un résumé des méthodes, des algorithmes et des outils logiciels pour la génération d'une machine virtuelle. Nous présentons une vue générale des relations entre les processus généraux de compression de programmes, de génération de dictionnaires de macro-instructions, de formats et de génération du code C d'une machine virtuelle.

Les différents programmes exécutant ces processus ont tous été écrits en **Scheme**.

##### *1.4.1 Génération des dictionnaires de formats et de macros*

C'est le processus le plus complexe de la part du concepteur de la machine virtuelle. La figure 1.1 montre l'essentiel de la génération. Un ensemble de programmes échantillons doit être soigneusement conçu pour représenter les programmes typiques exécutés sur la machine virtuelle.

C'est aussi assez coûteux, en terme de temps machine<sup>2</sup>. La construction des dictionnaires nécessite la conservation d'une grande quantité d'information, dont les emplacements des séquences d'instructions répétitives et les formats avec fréquence de toutes les instructions des programmes.

---

2. Ces temps proviennent de l'usage de cet algorithme pour la JVM et Machina/Scheme. C'est une évaluation très approximative car, de toute façon, les programmes effectuant ce travail étaient utilisés en mode interprété.

Cette génération produit deux dictionnaires finaux pour la génération de la machine virtuelle. Un dictionnaire contient les macro-instructions à implanter dans la machine virtuelle. On peut percevoir ces macros comme de nouvelles instructions. Le second dictionnaire contient les mnémoniques, formats et codes opérationnels de toutes les instructions, incluant les macros.

De plus, le concepteur doit fournir des paramètres pour contrôler les tailles de ces dictionnaires. Par exemple, celui-ci spécifie la fréquence d'occurrence minimum des séquences à conserver, l'espace occupé par l'implantation d'une instruction virtuelle, etc. Ces paramètres permettent d'élaguer l'information conservée et d'augmenter la rapidité de l'étape subséquente de création des instructions virtuelles.

Nous procédons en deux étapes.

La première fait une analyse des fréquences des instructions et de leur format, ainsi que des séquences d'instructions répétitives. Les paramètres du concepteur contrôlent la quantité de cas à conserver. Essentiellement, le contrôle se fait sur la fréquence minimum à atteindre. Cela produit deux bases d'informations sur les macros et formats potentiellement utiles pour la compression. La section 2.4 présente la construction de la base des formats et la section 2.3 la base des macros.

Dans la deuxième phase, il y a une évaluation des nouveaux formats et séquences candidats. Ceux permettant une compression sont placés dans les dictionnaires finaux.

Le problème de la génération optimum de ces deux dictionnaires est NP-complet ; nous utilisons donc un heuristique. Les détails de cet heuristique sont présentés au chapitre 2 ; et la figure 2.7 expose l'algorithme général de génération de ces deux dictionnaires. C'est en fait l'algorithme de création des instructions virtuelles, car le résultat est un ensemble de nouvelles instructions et de nouveaux formats.

#### *1.4.2 Génération de la structure du décodeur*

La génération du décodeur est relativement simple pour le concepteur. Deux types de décodeurs sont utilisés dans ce travail : les décodeurs automates et les décodeurs canoniques. Leur génération est effectuée automatiquement à partir des codes opérationnels et des formats. Les algorithmes de génération sont passablement différents ; il y a donc, en fait, deux outils différents pour cette génération. La figure 1.2 montre le déroulement de la génération.

Le concepteur de la machine virtuelle fait donc un choix entre ces deux décodeurs et précise, explicitement ou implicitement, l'espace maximum à utiliser pour les représenter.

Dans le cas des décodeurs automates, leur structure a une implication directe sur le code C de l'interprète. En effet, ce type de décodeur peut reconnaître plus d'une instruction par cycle de décodage, ce qui se traduit par l'introduction d'instructions combinées à implanter

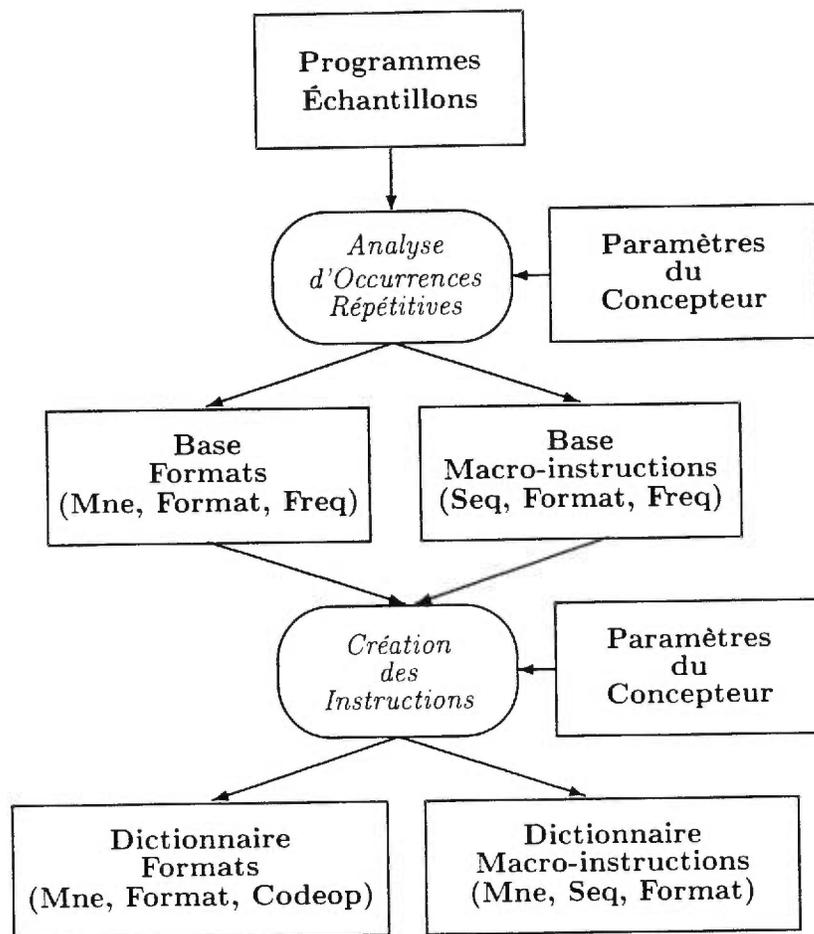


FIG. 1.1 – Génération des dictionnaires de formats et de macro-instructions

dans l'interprète. L'espace maximum n'est pas explicitement spécifié par le concepteur. Celui-ci fournit plutôt le nombre  $k$  de bits de décodage de la racine du décodeur. Cette valeur détermine explicitement la taille de la table de décodage, qui est dans  $O(2^k)$ . De plus, cette valeur a une implication sur l'espace du code de l'interprète, car plus  $k$  est élevée, plus d'instructions combinées sont reconnues par cycle.

Pour les décodeurs canoniques, c'est un espace explicite maximum qui est spécifié par le concepteur. Le générateur va nécessairement produire un décodeur satisfaisant cette contrainte, si un tel décodeur existe. La partie interprète ne dépend pas de la structure du décodeur canonique.

Le chapitre 3 traite de la génération des décodeurs automates et canoniques.

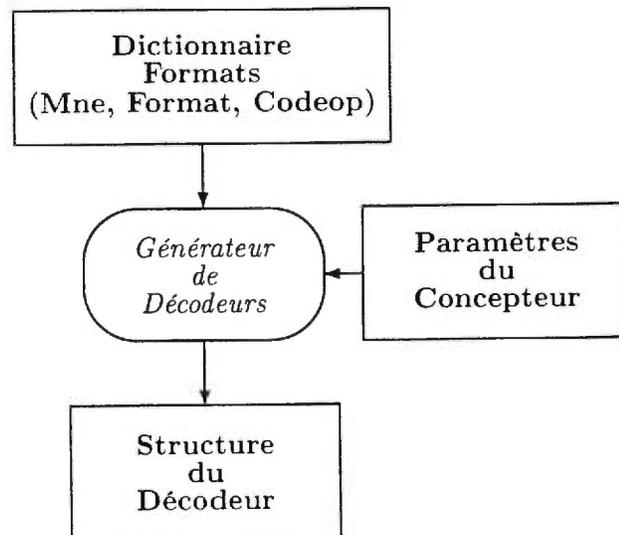


FIG. 1.2 -. Processus de génération de la structure d'un décodeur

### 1.4.3 Génération des machines virtuelles

Les machines virtuelles générées dans ce travail, le sont à partir des composantes générées par les processus décrits dans les sections précédentes. La figure 1.3 présente la relation de ces sources avec la machine virtuelle résultante exécutable.

Le générateur de code C produit automatiquement l'interprète et le décodeur en deux parties : l'une contenant les tables de décodage (.h), et l'autre (.c) le code effectuant le décodage à partir de ces tables.

Chaque instruction virtuelle, incluant les macro-instructions, est implantée dans l'interprète ainsi que l'extraction de ses paramètres au niveau du bit.

Les instructions de base, implantées en C, sont écrites sous la forme de macros C. Elles sont référencées par le code C de l'interprète.

Les fonctions utilitaires forment la partie restante pour le fonctionnement de la machine virtuelle. Cela inclut toutes les fonctions référencées par l'interprète, et des composantes comme le GC, le chargeur de programmes, etc.

Le chapitre 3 traite de la génération du code C des décodeurs et des interprètes.

## 1.5 Processus de compression d'un programme virtuel

Dans un premier temps, la compression d'un programme s'effectue en appliquant les macro-instructions disponibles dans la machine virtuelle. C'est essentiellement la substitu-

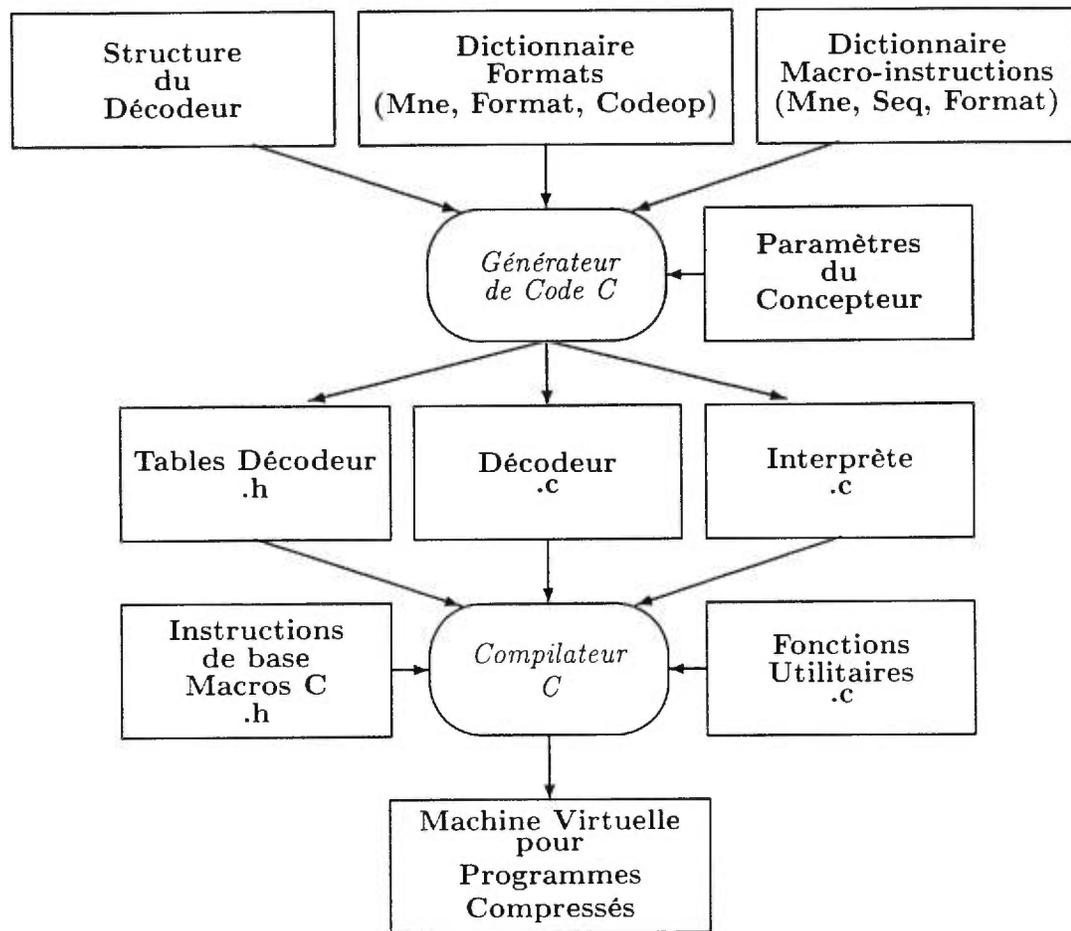


FIG. 1.3 – Processus de génération du code C et d'une machine virtuelle exécutable pour programmes compressés

tion de séquences d'instructions, potentiellement avec arguments, par des macro-instructions. Le problème de substitution optimum est NP-complet ; nous ne recherchons pas l'optimum et utilisons un heuristique. Nous appliquons séquentiellement les macro-instructions en débutant par les plus longues. Il faut vérifier qu'aucune instruction ne branche à l'intérieur de la séquence à remplacer. Toutes les instructions de branchement sont vérifiées, car à cause de l'attrition du code, dans certains cas, il faut modifier le nombre d'instructions à franchir. Notez que, préalablement, les instructions de branchement sont toutes transformées pour

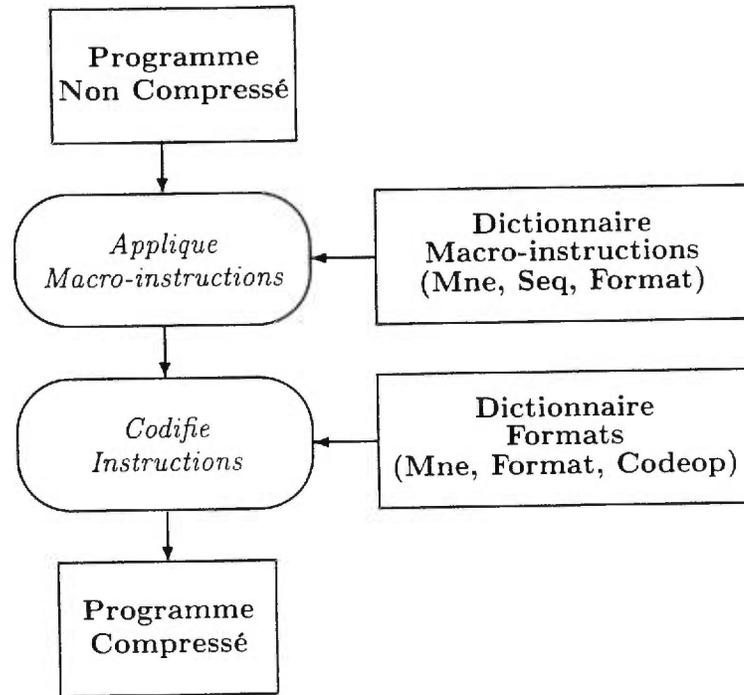


FIG. 1.4 -. Processus de compression d'un programme

indiquer le nombre d'instructions à franchir<sup>3</sup>.

Dans une seconde étape, le programme modifié est véritablement recodé en des codes opérationnels et des formats disponibles dans la machine virtuelle.

Le programme compressé ne dépend pas d'un décodeur particulier mais bien des macro-instructions, des codes opérationnels et des formats. Dans toutes nos expériences pour la JVM et la Machina, nous utilisons le même outil pour effectuer cette compression. La section 2.8 présente les détails.

### 1.6 Classes de machines virtuelles

Notre système crée une machine virtuelle en générant quatre composantes : un ensemble de macro-instructions, un codage des instructions, un décodeur et un interprète. Toutefois, il est possible de générer celles-ci de différentes façons, fournissant ainsi des machines virtuelles plus ou moins générales. C'est dans le choix des échantillons de programmes, pour la conception des instructions, que le concepteur intervient. Le choix des échantillons a un impact

3. Il faut appliquer un prétraitement pour certains langages. Par exemple, pour la JVM, les instructions de branchement spécifient le nombre d'octets à franchir, ce qui est modifié par le nombre d'instructions.

important sur l'ensemble des programmes pouvant être exécutés sur la machine virtuelle générée. Voici deux cas extrêmes.

**Spécialisée :** La machine ne peut exécuter qu'un seul programme. Elle ne peut charger un programme différent. Tout se passe comme si l'interprète généré était une compilation d'un programme. Celle-ci contient à la fois le décodeur, le programme compacté et l'interprète. Le concepteur a décidé de n'utiliser que ce programme comme échantillon. Le choix des formats et des macro-instructions implantés dans la machine virtuelle ne dépend que de celui-ci. Un cas extrême consisterait à remplacer toutes les instructions paramétrées par des macros spécialisées sans paramètres, implantées directement dans la machine virtuelle. Cela éliminerait la nécessité d'extraire les paramètres du code.

**Ouverte :** La machine permet d'exécuter une très large gamme de programmes. Sa caractéristique essentielle est que l'ensemble des formats des instructions est conçu pour accommoder ces programmes. Le concepteur doit faire un choix éclairé des échantillons pour effectuer la conception des dictionnaires de formats et de macro-instructions. Pour s'assurer la présence de certains formats, il faut, dans certains cas, démarrer l'algorithme de création des instructions avec un ensemble de formats initiaux envisagés comme les plus large possibles.

Les machines virtuelles spécialisées ont une application pratique importante. Elles permettent une génération de programmes exécutables très compacts. Les systèmes embarqués peuvent bénéficier de cette classe de machines. Leur génération devrait tenir compte, à la fois, des tailles de la machine virtuelle et du programme.

Dans le cas de la machine ouverte, la taille de la machine virtuelle a probablement moins d'importance. Le concepteur intervient ici pour juger de la pertinence de l'évaluation de sa taille. Pour plus de détails techniques, vous pouvez consulter la fin de la section 2.4.5, et l'algorithme de création des instructions à la section 2.7.

Bien entendu, il est possible de générer différentes machines pour différents échantillons ; celles-ci peuvent varier sur le décodeur et l'interprète. Cela permet d'avoir des machines plus rapides ou plus compactes, selon l'ensemble des programmes à exécuter. Par exemple, on pourrait envisager d'avoir des machines virtuelles différentes selon le type d'application : calcul numérique, traitement de matrices, télécommunication, traitement symbolique, etc. Il suffit d'employer un échantillon approprié pour reconnaître les besoins de ces programmes.

## 1.7 Compaction binaire et alignements

La compaction binaire est la concaténation de plusieurs champs de bits, chacun ayant une signification particulière, sans tenir compte des frontières de la mémoire centrale. Ainsi,

certaines champs ne débutent pas sur une frontière d'octet.

Assurément, l'alignement de toutes les instructions sur une frontière d'octet simplifie le décodage. Toutefois, cette façon de faire est un compromis majeur pour la compression du code. Sans la compaction, la compression perd une bonne part de son utilité.

La compaction provoque plusieurs difficultés techniques qui doivent être analysées attentivement. En fait, nous avons plusieurs choix sur la façon d'aligner les instructions. Dans cette section, nous analysons ces différents choix et les impacts sur l'exécution et le gain en espace. Nous concluons par une façon de compacter qui sera employée pour toute la suite du travail.

### 1.7.1 Changement du flot d'exécution versus compaction

Les instructions de changement de flot d'exécution posent une difficulté pour l'exécution du code compressé. Parmi ces instructions on retrouve les branchements conditionnels et inconditionnels, les appels de sous-routines et le retour de sous-routine. Nous appelons *instructions barrières* les instructions qui sont la cible d'instructions de branchement et les instructions qui suivent les instructions d'appel de sous-routine. En général, ce sont toutes les instructions pour lesquelles leur adresse doit être éventuellement spécifiée dans une instruction ou sauvegardée comme donnée.

Durant l'exécution d'un programme, il y a deux situations différentes pour les instructions barrières. Dans un cas, il est impossible d'exécuter l'instruction à moins de passer par une instruction de branchement (l'appel d'une sous-routine). Dans d'autres cas, l'instruction cible peut être rejointe par le flot normal d'exécution, ou par une instruction de branchement. Par exemple, pour les instructions de la forme « si-alors ». Dans le premier cas, l'alignement pose moins de difficulté car le décodage débute toujours de la même façon. Dans l'autre cas, l'alignement de l'instruction barrière requiert une instruction pour marquer cet alignement. Autrement, le décodeur pourrait décoder erronément les bits libres précédents l'instruction cible lors de son exécution sans branchement.

De plus, il faut distinguer deux problèmes différents causés par le flot d'exécution et la compaction des instructions: la sauvegarde de l'adresse de la prochaine instruction à exécuter et la spécification de l'adresse de la cible d'un branchement.

Analysons les techniques suivantes.

**Technique 1 :** Il n'y a aucun alignement pour toutes les instructions barrières.

**Technique 2 :** Il y a un alignement pour toutes les instructions barrières.

**Technique 3 :** Il y a un alignement seulement pour les instructions barrières cibles d'une instruction d'appel de sous-routine, et les instructions suivant l'appel des sous-routine.

Voici une analyse des avantages et désavantages de ces solutions.

L'avantage de la première technique est qu'il n'y a aucune perte d'espace mémoire. Une difficulté se présente pour l'instruction de branchement à une sous-routine : l'empilement de l'adresse de retour pose plusieurs difficultés. Cette adresse pourrait être un simple pointeur de bits, mais dans une implantation efficace elle peut se manifester sous la forme d'un pointeur de programme, du nombre de bits non décodés, etc. Toutes ces informations devraient être empilées ; ce qui modifie la hauteur de la pile selon la méthode d'implantation. Cela est difficilement acceptable car le compilateur devient dépendant de cette implantation<sup>4</sup>.

L'avantage de la deuxième technique est qu'il n'est plus nécessaire de spécifier les adresses au bit près. Son désavantage est la perte d'espace mémoire et l'introduction d'un mécanisme pour reconnaître et franchir les bits non utilisés. Par exemple, une instruction pourrait avertir le décodeur que la prochaine instruction à exécuter est sur la prochaine frontière d'octet. Nous avons fait quelques expérimentations de cette solution et conclu que non seulement cette instruction diminue le facteur de compression, mais elle vient ralentir l'interprète et compliquer la génération des décodeurs.

La troisième technique est une solution intermédiaire perdant légèrement de l'espace. L'avantage est que le traitement des appels de sous-routine se simplifie. L'adresse de retour sera l'adresse de l'octet suivant l'instruction d'appel. Ceci simplifie aussi le retour de fonction. Il y a toutefois quelques complications pour produire cette adresse de retour dans le cas de décodeurs chargeant plusieurs octets en même temps. Ce problème, et son traitement, seront élaborés à la section 3.7.

Nous considérons la troisième technique comme étant la meilleure. Elle simplifie les implantations, notamment la construction des décodeurs, tout en étant compacte. Elle est utilisée pour la suite du travail.

## ***1.8 Mesure d'entropie des programmes***

Avant toute tentative de compression d'un ensemble de programmes, une mesure de leur entropie permet d'orienter la méthode à utiliser. Cette section démontre quelques méthodes pour faire cette mesure, et suggère comment utiliser ces résultats pour améliorer la compression tout en permettant une exécution sans décompression préalable.

---

4. On peut aussi envisager l'utilisation d'une autre pile pour stocker les adresses de retour, mais clairement cela vient compliquer l'implantation.

### 1.8.1 Éléments de compression de données

Cette section présente quelques notions de base à propos de la compression des données, de la théorie de l'information, et de la codification à préfixes uniques. Nous nous référons principalement à [Sal98, Say96, BCW90] avec quelques adaptations. Ces notions seront utilisées, par la suite, pour la construction des algorithmes et pour mieux évaluer les résultats expérimentaux de compression.

#### Définition 1.8.1

Un **code** est une chaîne non-vide formée par les symboles '0' et '1'; c'est-à-dire des bits. Soit un ensemble fini de symboles  $S = \{s_i\}$ . Une **codification** de  $S$  est formée d'un ensemble de codes  $C$  et d'une bijection de  $S$  vers  $C$ . Une **codification à préfixe unique** de  $S$  est un ensemble de codes  $C = \{c_i\}$ , un pour chaque symbole, dont aucun code n'a comme préfixe un autre code.

Cette restriction sur les préfixes des codes permet leur reconnaissance dans une séquence de codes sans séparateurs. Une codification à préfixe unique permet la codification en binaire de chaînes de symboles, sans séparateurs, tout en permettant un décodage unique.

Voici la définition d'entropie d'une chaîne de symboles. Le concept d'entropie permet d'évaluer les performances d'une codification. Cette performance est exposée par la proposition 1.1.

#### Définition 1.8.2

Soient une chaîne non-vide  $M$  de symboles  $S = \{s_i\}$  et  $F = \{f_i\}$  leur fréquence d'occurrence dans  $M$ . La probabilité de  $s_i$  étant définie par  $p_i = f_i / \sum_{f_j \in F} f_j$ , l'**entropie** de la chaîne  $M$  est<sup>5</sup>

$$H(M) = - \sum_{s_i \in S} p_i \lg p_i \quad (1.1)$$

L'entropie peut servir à évaluer les possibilités de compression de la chaîne  $M$ . Si l'entropie est basse, une compression élevée de  $M$  peut être obtenue. La valeur maximum de  $H(M)$  est  $\lg |S|$ , dans le cas où les probabilités sont identiques, et de 0 dans le cas d'un seul symbole dans le message.

Dans le contexte de ce travail, une chaîne est une suite d'instructions d'un programme, plus particulièrement une suite de mnémoniques identifiant ces instructions.

---

5. Le symbole  $\lg$  est le logarithme en base 2.

**Définition 1.8.3**

Soient une chaîne non-vide  $M$  de symboles  $S = \{s_i\}$  et  $\{p_i\}$  leur probabilité d'occurrence dans  $M$ . La longueur moyenne d'une codification  $C = \{c_i\}$ ,  $c_i$  codant  $s_i$ , de longueurs  $l(c_i)$ , est la valeur

$$L(C) = \sum_{c_i \in C} p_i l(c_i) \quad (1.2)$$

Cette longueur moyenne permet d'évaluer la qualité de la codification. Plus celle-ci est basse, plus elle est adaptée aux probabilités provenant de la chaîne. Toutefois, ce facteur de compression est théoriquement limitée par son entropie. Une compression de  $M$  par une codification des symboles  $s_i$  ne peut donner une longueur moyenne de  $M$  plus basse que  $H(M)$ . En d'autres mots, chaque symbole de  $M$  utilisera, en moyenne, au moins  $H(M)$  bits.

**Proposition 1.1**

La longueur moyenne d'une codification  $C$  pour une chaîne  $M$  est bornée inférieurement par son entropie. Symboliquement,

$$H(M) \leq L(C) \quad (1.3)$$

Plus la différence  $L(C) - H(M)$  est faible, plus la qualité de la codification est bonne. Remarquez que la longueur moyenne permet d'évaluer le facteur de compression de  $M$  sans coder explicitement la chaîne  $M$ . C'est un moyen utile, en pratique, car l'encodage explicite de  $M$  pourrait nécessiter un temps de calcul important. La quantité  $L(C) - H(M)$  est la redondance du code  $C$  par rapport à la chaîne  $M$ .

## Chapitre 2

### CRÉATION DES INSTRUCTIONS VIRTUELLES

---

Nous présentons dans ce chapitre nos méthodes de codification compacte des instructions.

La première méthode consiste à compresser les codes opérationnels par codage de Huffman. L'attribution des codes doit tenir compte des fréquences de toutes les instructions de la machine virtuelle. C'est donc une fois la création des instructions effectuée que ces codes sont attribués. La section 2.2 traite de cette attribution.

Une part non négligeable de l'espace du code est utilisée par les arguments. Il est essentiel d'utiliser le moins d'espace possible pour coder chacun d'eux. Ce processus de diversification des instructions, selon la longueur des arguments, spécialise certains mnémoniques à des formats plus courts. De plus, le processus de spécialisation s'effectue sur la valeur des arguments des instructions. Ainsi, la spécialisation consiste en la création de nouveaux mnémoniques pour lesquels des valeurs de paramètres ont été fixées et/ou le nombre de bits des paramètres est inférieur au maximum possible.

À cause de la codification de Huffman, les codes opérationnels n'ont pas, en général, des longueurs multiples de huit. Ainsi, une codification compacte d'une suite d'instructions ne permet pas, en général, d'aligner les instructions sur une frontière d'octet. C'est donc dire qu'il n'est pas avantageux de tenter de codifier les paramètres sur des multiples de huit bits. Cela ne permettrait pas de les coder sur une frontière d'octet.

Ainsi, dans notre codification, les formats utilisent des longueurs au bit près. Pour effectuer un choix avantageux, les instructions candidates à ces spécialisations, ainsi que leurs formats, sont choisies par une analyse statistique du code généré par un compilateur.

La présence de séquences d'instructions répétitives est une autre source de compression. De telles séquences peuvent être remplacées par un seul code opérationnel. En d'autres mots, il y a création de macro-instructions.

Les sections suivantes traitent des détails de codification par code de Huffman, de recherche de formats compacts et de la création de macro-instructions à insérer dans la machine virtuelle finale. Ce chapitre culmine par l'algorithme de création des instructions virtuelles présenté à la figure 2.7.

Au préalable, il y a la création de deux bases : l'une pour les formats et l'autre pour les macro-instructions. Ces deux constructions sont traitées aux sections 2.4 et 2.3 respectivement.

Nous présentons les méthodes conventionnelles de codification dans la section suivante.

## 2.1 Méthodes conventionnelles de codification

L'analyse de la codification des instructions de plusieurs machines virtuelles ([LY99, Knu99, Dub96, Que94, Ler90, BJ86, Bra82, Ber78, Wor72]) révèle l'utilisation de quelques techniques permettant la compaction du code.

La technique la plus prédominante consiste à représenter chaque code opérationnel sur un octet, ce qui implique une limite de 256 codes. Cependant, ce n'est pas, dans un premier temps, une contrainte majeure. L'usage d'un octet facilite le décodage. Un nombre variable de bits, ou un nombre fixe de bits non multiple de huit, provoque des complications lors du décodage des instructions.

Puisque certains codes, parmi les 256, sont disponibles, on peut relever une deuxième technique fort répandue : la spécialisation de certaines instructions de base. Il y a plusieurs types de spécialisation : composition d'opérations élémentaires, différents formats de codification des arguments, et fixation d'un ou plusieurs arguments.

Par exemple, pour la JVM, il y a une instruction `iload` ayant un argument quelconque de 16 bits ; mais il y a aussi des instructions spécialisées `iload_0`, `iload_1`, etc., pour des valeurs particulières. Une autre forme de spécialisation est la réduction de l'espace des arguments. Dans la JVM, par défaut, la plupart des instructions ont un format court. Si une instruction nécessite un format plus long, elle est précédée de l'instruction `wide`.

Une technique plus complexe est la fusion de plusieurs opérations en une seule. En d'autres mots, la création d'une instruction complexe permettant de remplir une séquence de fonctions précises pour le langage à implanter. Par exemple, pour la JVM, l'instruction `monitorenter` permet d'entrer dans une section critique de code. En fait, l'introduction de ces instructions simplifient la tâche d'écriture des compilateurs. Elle a aussi l'effet de réduire la longueur du code.

En se fondant sur ces préliminaires, nous introduisons nos méthodes de codification compacte à la section suivante.

## 2.2 Attribution de codes opérationnels par codes de Huffman

Pour chaque mnémorique un code unique doit lui être attribué pour permettre son identification, par l'interprète, durant l'exécution. Pour attribuer un code opérationnel à chacun des mnémoriques de base et aux macro-instructions, nous utilisons la codification de Huffman. En fait, pour améliorer la vitesse et/ou l'espace mémoire lors du décodage, il est préférable d'utiliser l'ordre canonique [SK64, HL90] de ces codes.

Au préalable, les fréquences d'utilisation des mnémoriques ont été mesurées. Celles-ci servent à guider l'algorithme de Huffman dans l'attribution des codes opérationnels. La

propriété majeure du résultat de l'attribution est que plus la fréquence d'un mnémonique est élevée, plus le code opérationnel sera court.

Il existe plusieurs façons de procéder pour générer ces codes; les techniques conventionnelles sont présentées par [SK64, HL90]. Pour ce faire, il est nécessaire d'utiliser des programmes échantillons pour mesurer les fréquences d'occurrence. En résumé, la construction des codes canoniques de Huffman s'effectue en deux phases. La première est la méthode conventionnelle de Huffman [Huf52], et consiste en la construction d'un arbre, des feuilles à la racine. Les longueurs des codes sont ensuite utilisées pour générer des blocs de codes dont les valeurs numériques sont consécutives. Nous reviendrons sur ce sujet à la section 3.4. Le reste de ce chapitre ne dépend pas de cette codification particulière.

Pour montrer un exemple d'un large ensemble de codes opérationnels « raisonnables », et indépendamment d'une machine virtuelle, nous utiliserons les  $n$  probabilités d'un cas particulier de la loi de Zipf:  $p_i = 1/(iH_n)$ ,  $1 \leq i \leq n$ , où  $H_n$  est le nombre harmonique  $\sum_{j=1}^n (1/j)$ . On notera par Zipf- $n$  l'ensemble des  $n$  premiers nombres  $\{p_1, \dots, p_n\}$ <sup>1</sup>.

À titre d'exemple, la figure 2.1 montre l'arbre canonique pour les codes opérationnels générés par la distribution Zipf-20. Il y a vingt feuilles et dix-neuf noeuds intérieurs<sup>2</sup>.

Le code opérationnel de m1 est '00', de m6 est '1010' et l'un des plus longs, soit '111111', a été attribué à m20. La table 3.2 présente une partie des codes opérationnels des probabilités de Zipf-200. On peut mieux y voir la propriété des valeurs consécutives des codes de même longueur.

L'attribution des codes opérationnels doit tenir compte de la création de nouveaux mnémoniques, soit à cause de la création de macro-instructions ou de la spécialisation des instructions. Devrait-on concevoir les codes opérationnels et la spécialisation des instructions avant ou après la sélection des macro-instructions? L'application de macro-instructions requiert des codes opérationnels; il est donc essentiel d'effectuer une phase d'attribution de codes opérationnels après leur sélection. Toutefois, le choix d'une macro-instruction à sélectionner est influencé par le gain en espace. Sans connaître les longueurs de codification des instructions, il serait difficile d'évaluer adéquatement ce choix. Nous envisageons donc un processus cyclique d'attribution de codes opérationnels, de spécialisation d'instructions et de sélection de macro-instructions. Cela se traduit en un algorithme général de création d'instructions, présenté à la fin de ce chapitre à la section 2.7.

Les sections suivantes traitent des préliminaires de cet algorithme.

---

1. Ces probabilités présentent une similitude à la distribution des fréquences des mots dans les langues naturelles.

2. Cela est conforme à une propriété des arbres binaires complets:  $n$  feuilles implique  $n - 1$  noeuds intérieurs.

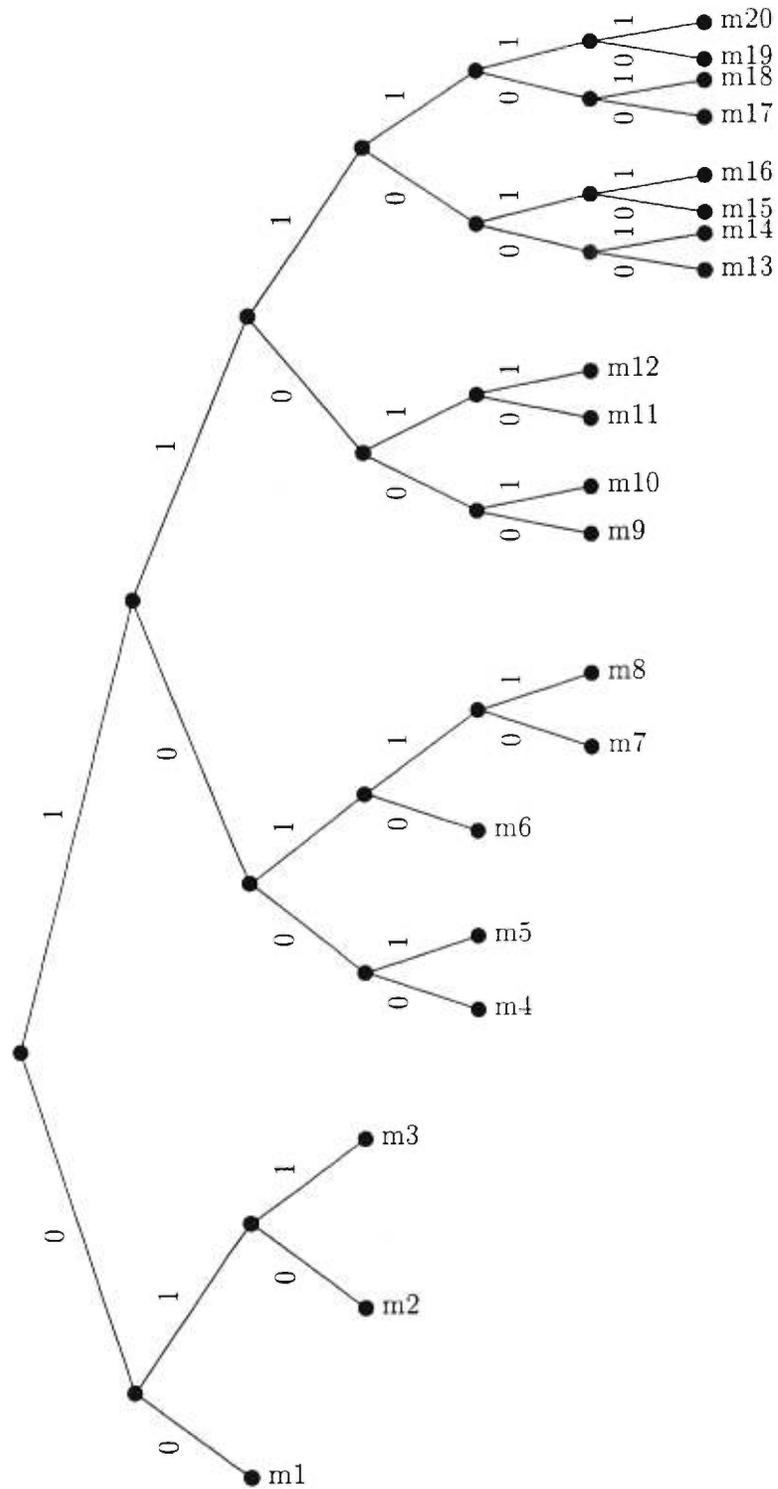


FIG. 2.1 -. L'arbre canonique de Huffman pour Zipf-20

### 2.3 Construction de la base des macro-instructions

Dans le processus de conception proposé, la construction d'une base de macro-instructions est une étape préliminaire à la création d'une machine virtuelle. Cette construction est indépendante de la construction de la base des formats ; elles peuvent être faites en parallèle.

Il faut distinguer cette étape de la sélection des macro-instructions. La base est un ensemble de séquences d'instructions paramétrés permettant une compression. Toutefois, seulement une partie de ces séquences pourrait être utilisée, car l'évaluation de leur utilité pour la compression ne peut être faite sans attribuer des codes opérationnels aux instructions de base.

Le travail essentiel de la création de la base consiste donc à conserver les séquences d'instructions paramétrés en tenant compte de leur fréquence d'occurrence. À cette étape, aucune évaluation de l'espace mémoire réel n'est effectuée pour décider du choix de ces séquences, bien qu'un minimum de fréquences permet d'élaguer les séquences jugées inutiles. L'évaluation précise sera effectuée par l'algorithme général d'attribution des codes opérationnels, de spécialisation des instructions et de l'application des macro-instructions présentés à la section 2.7.

L'un des principes de base guidant la conception des heuristiques utilisées lors de la sélection, nécessite de garder les emplacements des séquences. Ces emplacements permettent d'éliminer les chevauchements des séquences et de maintenir des fréquences d'occurrences adéquates. Sans cela, les évaluations seraient trop approximatives. D'autre part, les emplacements fournissent la possibilité de ne plus référencer le code lui-même lors de l'exécution de l'algorithme de création des instructions. C'est un autre principe guidant la conception des heuristiques d'évaluation de gain en espace : une fois les bases construites à partir des programmes échantillons, ceux-ci ne sont plus référencés par la suite.

L'algorithme général de la section 2.3.3 est précédé d'un pré-traitement (section 2.3.2) et d'un post-traitement (section 2.3.4). Le pré-traitement permet d'éviter d'analyser, a priori, des séquences jugées impossibles à implanter dans la machine virtuelle. Par exemple, l'appel d'une sous-routine ne peut se produire au milieu d'une macro-instruction. D'autre part, le post-traitement élimine les séquences finales ne remplissant pas d'autres critères plus facilement reconnaissables à la fin de la création de toutes les séquences. De plus, il y a complétion des formats et ajustement des fréquences d'occurrence.

Les sections suivantes présentent la méthode retenue pour la création d'une base de macro-instructions.

Java(JVM)		Scheme(Machina)	
Code Macro	Signification	Code Macro	Signification
(aload_0) (getfield *) (iload_1) (aload)	Référence au premier élément d'un vecteur du premier objet local.	(pushi 7) (not) (and) (dup) (pushi 1) (pusha) (jsr)	Appel d'une fonction dont l'adresse se trouve dans une fermeture.
(aload_0) (aload_1)	Chargement des deux premières références locales.	(pushi *) (alloc 8) (pushi 3) (or)	Allocation d'un objet dans le monceau et étiquetage.
(iconst_1) (ireturn)	Retourne la valeur un.	(storel *) (ret *)	Stocke le résultat d'une fonction, dépile les paramètres et retour.

TAB. 2.1 – Exemples de macros pour deux machines virtuelles

### 2.3.1 La forme générale des macro-instructions

Une macro-instruction est essentiellement une séquence d'au moins deux instructions de base. Une telle macro peut avoir des paramètres. Dans la macro-instruction, une référence à un paramètre se fait par la position de celui-ci.

La figure 2.1 démontre des macro-instructions pour les machines virtuelles JVM et Machina. L'étoile indique la présence d'un paramètre. Ces macros proviennent d'une conception automatique effectuée sur des échantillons de programmes.

Les macro-instructions sont éventuellement intégrées à la machine virtuelle. En quelque sorte, elles sont de nouvelles instructions découvertes par le processus de compression des programmes.

### 2.3.2 La division en blocs élémentaires

Avant de procéder à la recherche de séquences fréquentes, les programmes sont subdivisés en blocs élémentaires. La raison de cette subdivision est la suivante.

Certaines séquences d'instructions ne peuvent être utilisées comme macro-instructions. Par exemple, une séquence d'instructions de base, sans branchement, qui aurait une instruction de retour d'appel de procédure, pourrait très difficilement se trouver au milieu d'une macro ; car l'interprétation de la macro-instruction se terminerait sans exécution des instructions suivant le retour d'appel.

La création des séquences est donc contrôlée par deux aspects : gain en compression et facilité d'implantation.

JVM (Java)	Machina (Scheme)
<pre>jsr jsr_w ret ireturn lreturn freturn dreturn areturn return athrow breakpoint tableswitch lookupswitch ifeq ifne iflt ifge ifgt ifle if_icmpeq if_icmpne if_icmplt if_icmpge if_icmpgt if_icmple if_acmpeq if_acmpne goto goto_w ifnull ifnonnull</pre>	<pre>jsr jmp ret stop</pre>

TAB. 2.2 –. Instructions ne pouvant être au milieu d’une macro-instruction

Ainsi, pour aider la recherche de séquences utiles, les programmes échantillons sont découpés en plus petites unités: en *blocs élémentaires*. Cette découpe peut être contrôlée par un prédicat, nommé *inst-fin?*. Celui-ci indique si une instruction de la machine virtuelle termine un bloc élémentaire. La table 2.2 présente les instructions de deux machines virtuelles pour lesquelles *inst-fin?* est vrai. De telles instructions ne peuvent se trouver qu’à la fin d’une macro-instruction<sup>3</sup>.

### 2.3.3 L’algorithme de recherche de séquences répétitives

La figure 2.2 présente l’essentiel de l’algorithme de construction de la base. Voici quelques explications de son fonctionnement.

De façon générale, l’emplacement d’une séquence d’instructions sera indiqué par un couple  $(b, j)$ , où  $b \geq 0$  est l’indice du bloc élémentaire et  $j \geq 0$  est l’indice dans ce bloc de la première instruction de la séquence.

Le paramètre  $lb$  est une liste des blocs élémentaires construit selon la méthode de la section 2.3.2. Le paramètre  $f_{\min}$  spécifie la fréquence minimum des séquences à garder, et  $l_{\max}$  est la longueur maximum des séquences. La fréquence minimum permet de contrôler la rapidité de construction de la base. En effet, si une fréquence élevée est spécifiée, peu de macro-instructions seront gardées dans les premières itérations, ce qui augmente substantiellement la rapidité de construction des séquences subséquentes. En pratique, le concep-

3. Cette table n’est qu’un exemple. Dans le cas de la JVM, des variations sont possibles selon l’implantation des instructions de base.

teur peut spécifier une fréquence minimum assez élevée, permettant ainsi une construction rapide de la base. Si par la suite, c'est-à-dire à la phase subséquente de la sélection des macro-instructions, il semble avantageux d'en augmenter le nombre, il suffit de reprendre le processus.

Le paramètre  $\equiv$  désigne le prédicat d'équivalence de deux séquences d'instructions<sup>4</sup>. Celui-ci dépend des instructions virtuelles de base. Indirectement, il spécifie la façon d'introduire ou d'exclure des paramètres dans les macro-instructions. Par exemple, ce prédicat peut déclarer deux séquences équivalentes ssi les mnémoniques sont égaux. Ainsi, les deux séquences  $((\text{pushi } 2)(\text{add}))$  et  $((\text{pushi } 3)(\text{add}))$  seraient équivalentes. Cela signifie que la macro est la séquence  $((\text{pushi } *) (\text{add}))$ , ayant un paramètre. À cause de leur sémantique, certaines instructions ne seront équivalentes que si leurs arguments sont égaux. Par exemple, pour les instructions de branchement, il est préférable de les considérer équivalentes seulement si elles sont égales. Cela simplifie l'implantation des macro-instructions.

La fonction  $Seq_2(lb, f_{\min})$  construit toutes les séquences de deux instructions, apparaissant dans la liste  $lb$  et ayant une fréquence d'occurrence d'au moins  $f_{\min}$ . L'opération  $s \in (\equiv)S$  détermine si  $s$  est dans  $S$  selon le prédicat d'équivalence  $\equiv$ . Cette opération devrait être implantée efficacement, car elle est la plus complexe de toutes les opérations de base. Dans notre implantation en Scheme, elle est implantée en utilisant une méthode de fonction de dispersion (« hashing »).

La fonction  $Seq_n(S_i, lb, f_{\min})$  construit des séquences de longueur  $i + 1$  à partir de séquences de longueur  $i \geq 2$ . Cette construction peut se faire entièrement en réitérant sur les séquences de longueur  $i$ ; ce qui permet une construction rapide. La figure 2.3 montre l'essentiel de la méthode. La séquence  $s_1$  fait partie de l'ensemble  $S_i$ , précédemment construit. La séquence  $s$  est générée par la concaténation de  $s_1$  et de l'instruction à l'emplacement  $(b, j + i)$ ; cette instruction est l'une des instructions suivant un exemplaire de  $s_1$ . La séquence  $s_2$  de longueur  $i$  est un suffixe de  $s$ . Si cette séquence existe dans  $S_i$ , selon  $\equiv$ , ses emplacements  $P_{s_2}$ , et ceux de  $s_1$  permettent de construire rapidement les emplacements de  $s$ . La séquence  $s$  est gardée ssi la cardinalité de l'intersection des emplacements des deux séquences est supérieure ou égale à  $f_{\min}$ . L'expression  $P_1 \# P_2$  désigne l'intersection  $\{(b_1, i_1) | (b_1, i_1) \in P_1, (b_2, i_2) \in P_2, b_1 = b_2, i_1 + 1 = i_2\}$ ; c'est-à-dire, l'intersection des emplacements des deux séquences, en tenant compte de la translation d'une instruction de  $s_2$ .

Naturellement, si  $S_i$  est vide, il n'existe aucune séquence de longueur supérieure à  $i$  ayant une fréquence au minimum  $f_{\min}$ . La condition de fin d'itération se termine donc sur

---

4. En pratique, un prédicat d'équivalence d'instructions est plus simple. Dans la majorité des cas, le prédicat d'équivalence de séquences se définit comme étant l'équivalence deux à deux de leurs instructions.

```

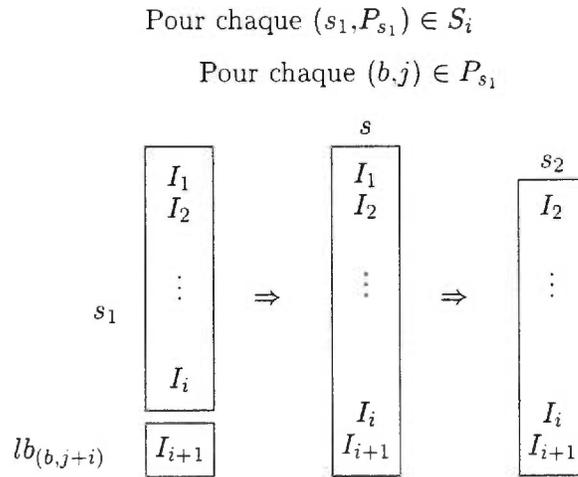
CreeBase( $lb, f_{\min}, l_{\max}, \equiv$ )
1    $i \leftarrow 2$ 
2    $S_i \leftarrow Seq_2(lb, f_{\min}, \equiv)$ 
3    $D \leftarrow S_i$ 
4   Tantque  $i \leq l_{\max}$  et  $S_i \neq \emptyset$ 
5        $S_{i+1} \leftarrow Seq_n(S_i, lb, f_{\min}, \equiv)$ 
6        $D \leftarrow D \oplus S_{i+1}$ 
7        $i \leftarrow i + 1$ 
8   Retourner  $D$ 

Seq2( $lb, f_{\min}, \equiv$ )
9    $S \leftarrow \emptyset$ 
10  Pour chaque bloc  $b \in lb$ 
11      Pour chaque séquence  $s = (I_1, I_2)$  dans  $b$ 
12          Si  $s \in (\equiv)S$  Alors Soit  $r = (s_r, P)$  le représentant de  $s$ 
13               $P \leftarrow P \oplus (b, i)$ 
14              Sinon  $S \leftarrow (s, \{(b, i)\})$ 
15   $S \leftarrow$  séquences de  $S$  où  $|P| \geq f_{\min}$ 
16  Retourner  $S$ 

Seqn( $S_i, lb, f_{\min}, \equiv$ )
17   $S_{i+1} \leftarrow \emptyset$ 
18  Pour chaque séquence  $s_1$  de  $S_i$ 
19      Soit  $P_{s_1}$  les emplacements de  $s_1$ 
20      Pour chaque  $(b, j) \in P_{s_1}$  où  $lb_{(b, j+i)}$  existe
21          Soit  $s = s_1 \oplus lb_{(b, j+i)}$ 
22          Soit  $s_2 = s[2..i]$ 
23          Si  $s \notin (\equiv)S_{i+1}$  et  $s_2 \in (\equiv)S_i$  Alors
24              Soit  $r = (s_r, P_{s_2})$  le représentant de  $s_2$ 
25               $P_s \leftarrow P_{s_1} \uplus P_{s_2}$ 
26              Si  $|P_s| \geq f_{\min}$  Alors  $S_{i+1} \leftarrow S_{i+1} \oplus \{(s, P_s)\}$ 
27  Retourner  $S_{i+1}$ 

```

FIG. 2.2 -. Algorithme de création de la base de séquences



$P_s = P_{s_1} \uplus P_{s_2}$ , Si  $|P_s| \geq f_{\min}$  Ajouter  $(s, P_s)$  à  $S_{i+1}$

FIG. 2.3 – Génération des séquences  $s$  de longueur  $i + 1$

cet événement ou quand la longueur maximum des macros est dépassée<sup>5</sup>.

La base construite contient des séquences pouvant se chevaucher. C'est certainement le cas pour des séquences de longueurs différentes, mais c'est aussi le cas pour les séquences de même longueur. De plus, des séquences présentes dans la base ont probablement des occurrences invalides, car des branchements pourraient s'effectuer au milieu de la séquence. Finalement, chaque liste d'emplacements d'une séquence doit être partitionnée en sous-listes selon les formats de base. Ces ajustements sont effectués en post-traitement.

#### 2.3.4 Post-traitement de la base des séquences

Après l'application de l'algorithme de la section précédente, les séquences sont analysées pour éliminer les cas jugés impossibles à implanter dans la machine virtuelle. Cette évaluation peut varier d'une machine à l'autre, mais elle contient essentiellement les trois

5. La longueur maximum a deux utilités pratiques. Bien sûr, elle limite la longueur des macros, mais elle évite aussi de poursuivre la construction de macros qui s'avèrent être une répétition adjacente d'une séquence courte. Par exemple, dans la seule séquence  $aaa \dots a$ , il est possible de poursuivre la construction de très longues séquences sans avantage majeur.

vérifications suivantes.

Si deux emplacements se chevauchent, il faut comptabiliser un seul emplacement pour la fréquence d'occurrence. L'évaluation du chevauchement est effectuée en balayant séquentiellement la liste d'emplacements. Ce n'est pas un calcul optimum, mais une approximation. Cela n'a pas, à notre avis, d'impact majeur car les séquences se chevauchant elles-mêmes fréquemment ont une structure cyclique correspondant à un code très particulier et peu fréquent.

Une instruction de branchement ne peut apparaître, dans une séquence, que si sa cible de branchement se trouve dans celle-ci ou à sa sortie. Cette vérification est rapide à effectuer. Toutes les séquences ne remplissant pas cette condition sont éliminées de la base.

Pour chaque emplacement, si une instruction branche de l'extérieur vers l'intérieur de la séquence, l'emplacement doit être enlevé. La séquence n'est pas éliminée car c'est seulement une instance de la séquence. Si tous les emplacements sont enlevés, la séquence est effectivement éliminée.

De plus, dans la base finale, chaque séquence doit avoir un format. Ainsi, la liste des emplacements est partitionnée en sous-listes. Il y a une sous-liste d'emplacements par format. La structure de la base est donc  $[(seq [(format freq [(ib . j)])])]$  où  $(ib . j)$  est un emplacement<sup>6</sup>.

Une approximation peut être appliquée, ici, par le concepteur. Dans certaines situations, le nombre de formats de base est très élevée par rapport au nombre total d'emplacements. Pour diminuer ce nombre, il est possible de transférer les emplacements de formats peu fréquents à leur supremum. Ce transfert est effectué récursivement jusqu'à l'obtention soit d'un seul format, ou quand tous les formats ont une fréquence minimum. Ce processus peut diminuer considérablement le temps de calcul de la macro-instruction ayant un gain maximum. Il est sous le contrôle du concepteur de la machine virtuelle par le choix de la fréquence minimum.

Finalement, l'ensemble des formats de chaque séquence est complété par l'algorithme de la section 2.4.2, et de la section 2.4.4. Un format ajouté par l'algorithme de complétion a une liste vide d'occurrence. Ce cas particulier permet de réduire le temps de traitement de la sélection des macro-instructions. En fait, la liste d'emplacements d'un tel format est l'union des formats couverts. L'évaluation du gain en espace, pour une telle séquence, fera donc le travail préliminaire de calculer la somme des fréquences d'occurrences des formats couverts. Cette façon de procéder réduit substantiellement la liste des emplacements à mettre à jour après chaque sélection d'une macro, car les formats provenant de la complétion couvrent, et cela à répétition, la majeure partie des emplacements des séquences.

---

6. La notation  $[...]$  représente une liste d'éléments.

## 2.4 Construction de la base des formats

Certaines instructions ont des paramètres. Ceux-ci occupent une part non négligeable de l'espace du code. À titre d'exemple, pour la librairie de JDK 1.1, les paramètres occupent près de 48% de l'espace du code-octet. Il est donc important de choisir un nombre adéquat de bits pour représenter les paramètres d'une instruction. Puisque les instructions compressées ne sont pas alignées sur une frontière d'octet, nous pouvons attribuer un nombre de bits non multiple de huit aux longueurs des paramètres.

Nous considérons deux méthodes de spécialisation : la première fixe la valeur de paramètres, et la seconde fixe leur longueur, c'est-à-dire le nombre de bits pour représenter leur valeur.

### Définition 2.4.1

*Soit le mnémonique  $m$  ayant  $p$  paramètres. Une version spécialisée de ce mnémonique peut être de deux types.*

**Type Valeur :** *Spécialise sur une ou plusieurs valeurs des  $p$  paramètres de  $m$ . L'instruction spécialisée contient, dans sa définition, une ou plusieurs valeurs précises apparaissant dans plusieurs instructions de  $m$ . Les paramètres correspondants aux valeurs disparaissent de l'instruction spécialisée.*

**Type Format :** *Spécialise sur une ou plusieurs longueurs des  $p$  paramètres de  $m$ . Cette version codifie certains des paramètres en moins de bits que le maximum possible.*

Par exemple, une instruction `pushi_0` (type valeur) pourrait être créée si la valeur 0 apparaît fréquemment comme argument.

Ou encore, une instruction `pushi` pourrait être spécialisée (type format) pour coder son argument sur 4 bits, 8 bits, 12 bits ou 32 bits. Dans ce cas, il y aurait quatre instructions `pushi`, que nous pourrions appeler `pushi4`, `pushi8`, `pushi12` et `pushi32`, chacune ayant une codification de longueur différente pour accommoder exactement le paramètre.

D'autres types de spécialisation sont possibles. Par exemple, si une instruction accepte des entiers négatifs et positifs, il serait possible, pour améliorer la vitesse d'extraction des paramètres, de la partager, selon le signe de l'argument, en deux instructions, ou encore, d'utiliser deux types de branchement : l'un pour avancer et l'autre pour reculer<sup>7</sup>. Toutefois, ce genre de spécialisation sort du cadre de cette section, car nous nous intéressons plus spécifiquement à la représentation compact des paramètres.

---

7. Cette possibilité permettrait d'être plus efficace dans le cas des programmes compressés. Car, pour les programmes compressés, un branchement vers l'arrière doit se faire d'une façon différente d'un branchement vers l'avant.

La phase de spécialisation introduit de nouvelles instructions ayant la même sémantique mais dont le format diffère. De nouveaux codes opérationnels doivent être attribués pour celles-ci. On peut envisager deux méthodes pour faire cette attribution. L'une permet éventuellement de faire une extension des longueurs des paramètres, et l'autre fixe un ensemble de longueurs.

**Méthode évolutive :** Cela peut être fait après l'attribution des codes opérationnels de base. Il suffit d'utiliser le code opérationnel déjà choisi et de lui ajouter des suffixes. Cela produit toujours des codes à préfixe unique.

**Méthode fermée :** Tous les codes opérationnels sont attribués en même temps. Ainsi, deux codes opérationnels, sans aucun préfixe commun, peuvent être attribués à deux instructions ayant la même sémantique, mais dont le format de codage des paramètres diffère.

De façon générale, la méthode fermée génère un code plus compact que la méthode évolutive. Toutefois, elle permet difficilement l'implantation d'une machine virtuelle dont les longueurs des paramètres peuvent augmenter avec les besoins. En fait, la méthode évolutive permet d'étendre un ensemble d'instructions déjà existant sans affecter des programmes objets ; la nouvelle machine peut exécuter des anciens ou des nouveaux programmes.

Nous utilisons la méthode fermée, car celle-ci est la plus prometteuse pour la compression des programmes.

Deux sources d'information devraient être utilisées pour décider de l'étendue de la spécialisation : des statistiques provenant de programmes échantillons, et le concepteur de la machine virtuelle. L'usage de programmes échantillons maintient le principe d'automatisation du processus de création de la codification des instructions, et d'une recherche d'un code compact. À priori, le concepteur peut prévoir des longueurs possibles plus élevées que celles apparaissant dans les programmes typiques retenus. Par exemple, pour être certain qu'une instruction `pushi`, avec un paramètre de 32 bits, existe dans la machine virtuelle, le concepteur intervient pour forcer sa génération.

La spécialisation des instructions a des impacts positifs et négatifs sur la compaction du code. Un algorithme de création d'instructions spécialisées devrait tenir compte de ces impacts.

1. L'introduction d'une instruction ayant des paramètres plus courts diminue l'espace mémoire pour la codification des instructions correspondantes. Pour la diminution de l'espace mémoire, c'est la raison d'être de la création de version spécialisée. Ce gain est évalué par la fonction  $g_{\text{format}}$  définie plus bas.
2. L'introduction d'une nouvelle instruction requiert un nouveau code opérationnel. Celui-ci peut augmenter la longueur de plusieurs autres codes opérationnels.

3. Une nouvelle instruction augmente l'espace mémoire pour le décodeur, car il faut au minimum une nouvelle adresse de branchement. Ce n'est pas le seul impact négatif pour le décodeur, puisque cela peut provoquer une nouvelle branche de décodage.
4. Naturellement, une nouvelle instruction augmente l'espace de la partie de l'interprétation des instructions virtuelles.

Dans les deux premiers cas, et pour une seule instruction, l'augmentation ou la diminution de l'espace mémoire peut être évaluée précisément. Les troisième et quatrième cas sont plus difficiles à évaluer puisqu'ils dépendent du processeur recevant l'implantation de l'interprète. Puisqu'il est nécessaire d'en tenir compte, une évaluation utilisant une borne inférieure est employée. Cette borne devrait indiquer l'espace mémoire minimum à utiliser pour interpréter une nouvelle instruction spécialisée.

L'évaluation de l'augmentation de l'espace mémoire dû à l'ajout d'une instruction est techniquement simple à faire : il suffit d'appliquer l'algorithme de Huffman avec cette nouvelle instruction et de calculer l'espace mémoire en tenant compte des fréquences d'occurrence et des longueurs des codes opérationnels obtenus.

La fonction d'évaluation du gain en espace mémoire pour la création d'une version spécialisée d'une instruction tient compte des quatre facteurs importants énumérés précédemment. Regardons plus précisément une méthode exacte pour évaluer la spécialisation de type format. Nous aborderons, plus loin, le type valeur et la construction d'une base de formats.

#### 2.4.1 Spécialisation de type format

L'idée centrale de ce type de spécialisation est de réduire le nombre de bits occupés par les paramètres des instructions. Il s'agit donc d'introduire de nouveaux formats pour un même mnémonique. Techniquement, cela introduit de nouveaux mnémoniques, un pour chaque format ; conséquemment, un nouveau code opérationnel pour chaque nouveau format.

Cette spécialisation se fonde sur des statistiques des formats de codification d'instructions provenant de programmes échantillons.

#### Définition 2.4.2

Les **formats** d'un mnémonique  $m$  à  $p$  paramètres sont décrits par des tuples de nombres entiers  $(l_1, \dots, l_p)$  où  $l_i$  spécifie le nombre de bits occupés par le  $i$ ème paramètre de  $m$ .

La définition suivante permet de préciser le concept d'une instruction codée par un format plus grand que nécessaire.

#### Définition 2.4.3

Un format  $F_a = (a_1, \dots, a_p)$  couvre, ou est supérieur ou égal à, un autre format  $F_b = (b_1, \dots, b_p)$  ssi on a  $a_i \geq b_i, \forall i$ . Si  $F_1$  couvre  $F_2$  on écrit simplement  $F_1 \geq F_2$  ou  $F_2 \leq F_1$ .

L'ensemble des formats d'un mnémonique devrait permettre de codifier toutes les instructions, pour ce mnémonique, envisagées par le concepteur de la machine virtuelle. Cet ensemble peut avoir un seul élément.

L'échantillonnage des formats se fait en colligeant tous les plus petits formats, c'est-à-dire ceux utilisant le moins de bits et pouvant codifier les instructions d'un ensemble de programmes. En fait, cet échantillonnage rapporte les formats et la fréquence d'utilisation de chacun d'eux. Le calcul du nombre de bits nécessaires pour la codification des paramètres dépend des types et des valeurs de ces paramètres<sup>8</sup>. Nous appelons les formats obtenus par échantillonnage, les *formats de base*. La section 2.4.6 présente le processus de construction d'une base de formats. Nous abordons les préliminaires pour effectuer cette construction.

#### 2.4.2 La complétion d'un ensemble de formats

Les spécialisations possibles pour le type format devraient être restreints aux cas où il y a un gain en espace. Pour un mnémonique  $m$ , cet ensemble ne contient pas toutes les combinaisons possibles des longueurs des paramètres. Par exemple, supposons qu'un mnémonique à trois paramètres a les formats de base  $(2,3,6)$ ,  $(4,2,7)$ ,  $(1,4,2)$  et  $(5,2,3)$ . Ce sont les formats de base possibles, c'est-à-dire que quatre instructions spécialisées de type format pourraient être créées pour ces longueurs. Il faudrait toutefois considérer d'autres cas. Par exemple, le format  $(2,4,6)$  peut coder les instructions dont les formats sont  $(2,3,6)$  et  $(1,4,2)$ . Il serait toutefois superflu, selon les formats de base, de considérer le format  $(3,4,7)$ , car celui-ci ne couvre pas plus d'instructions et utilise plus de bits.

Voici quelques définitions avant d'aborder un algorithme pour construire cet ensemble.

##### Définition 2.4.4

La *longueur* d'un format  $F = (l_1, \dots, l_p)$ , notée  $l_F$ , est la somme des longueurs de  $F$  ( $\sum l_i$ ). Le *supremum* d'un ensemble de formats  $P = \{F_1, \dots, F_n\}$ , où  $F_i = (l_{i1}, \dots, l_{ip})$  est le format  $(\max(l_{11}, \dots, l_{n1}), \dots, \max(l_{1p}, \dots, l_{np}))$ . Ce supremum de  $P$  est noté  $\sup(P)$ .

Voici une définition précise de l'ensemble des formats considérés pour la spécialisation de type format.

##### Définition 2.4.5

Soit un mnémonique  $m$  à  $p$  paramètres et  $P = \{F_k\}, F_k = (l_{k1}, \dots, l_{kp})$  l'ensemble des formats de base de  $m$ . La *complétion* de  $P$ , dénoté  $P^c$ , est l'ensemble  $\{F | \exists S \subseteq P, F = \sup(S)\}$ .

La figure 2.4 décrit un ensemble de formats et sa complétion. Notez que si  $p = 1$ , c'est-à-dire un mnémonique à un paramètre, l'ensemble  $P$  est sa propre complétion ( $P = P^c$ ).

8. Dans les cas pratiques considérés (JVM et Machina), ce calcul est simplement le nombre de bits occupés par un entier signé ou non-signé. Voir la section 2.4.3

$P$	$P^c$
(1, 4, 2) (2, 3, 6) (4, 2, 7) (5, 2, 3)	(1, 4, 2) (2, 3, 6) (4, 2, 7) (5, 2, 3)
	(2, 4, 6) (4, 3, 7) (4, 4, 7) (5, 2, 7)
	(5, 3, 6) (5, 3, 7) (5, 4, 3) (5, 4, 6)
	(5, 4, 7)

FIG. 2.4 –. Un ensemble de formats  $P$  et sa complétion  $P^c$ 

Cette définition de la complétion de  $P$  nous offre, d'emblée, un algorithme simple : pour calculer  $P^c$ , il suffit de générer tous les sous-ensembles  $S$  de  $P$  et de calculer  $\sup(S)$ . Clairement, c'est un algorithme dont la complexité en temps est dans  $\Omega(2^{|P|})$  ; ce qui laisse à désirer. Toutefois, l'ensemble  $P^c$  peut-être calculé autrement en ne considérant que des paires de formats.

### Proposition 2.1

L'ensemble  $P^c$  défini en 2.4.5 est l'ensemble minimal tel que  $P \subseteq P^c$ , et quelques soient les formats  $F_a, F_b \in P^c$   $\sup(\{F_a, F_b\}) \in P^c$ .

**Démonstration** 1) Montrons que  $P \subseteq P^c$ . Si  $F \in P$  alors  $\{F\} \subseteq P$ , ainsi selon 2.4.5  $F = \sup(\{F\}) \in P^c$ . 2) Soient deux formats  $F_a$  et  $F_b$  de  $P^c$  ; alors il y a deux sous-ensembles  $S_a$  et  $S_b$  de  $P$  tels que  $F_a = \sup(S_a)$  et  $F_b = \sup(S_b)$ . On a alors  $\sup(\{F_a, F_b\}) = \sup(S_a \cup S_b) \subseteq P^c$ . 3) Montrons que  $P^c$  est minimal. Supposons le contraire, c'est-à-dire supposons un ensemble  $C$  tel que  $C \subset P^c$  et ayant les deux propriétés. Il y a, alors, au moins un format  $F$  de  $P^c$  n'appartenant pas à  $C$ . Ainsi, selon la définition 2.4.5, il existe un  $S \subseteq P$  tel que  $F = \sup(S)$ , si  $S$  n'a qu'un seul élément,  $F$  appartiendrait à  $C$ , ainsi il y a au moins deux éléments dans  $S$  ; soit  $\{f_1, f_n\}$  ses éléments, mais puisque  $f_i \in P$ , on a  $f_i \in C$ , ainsi  $\sup(f_1, \sup(f_2, \dots), f_n) \dots) \in C$ , ce qui est contradictoire. ■

Cette proposition nous offre une autre façon de calculer l'ensemble  $P^c$  : à partir de  $P$ , calculer les supremum des paires de formats jusqu'à saturation, c'est-à-dire jusqu'au point où aucune paire de formats ne peut générer, par son supremum, un nouveau format. Il est possible aussi, pour augmenter la rapidité, de séparer les nouveaux formats créés des anciens formats. Un nouveau format ne peut être créé qu'en utilisant deux formats dont au moins un provient du nouvel ensemble.

Une autre amélioration de l'algorithme de construction de  $P^c$  provient de la proposition suivante.

**Proposition 2.2**

Soit un ensemble de formats  $P$  ayant  $p$  paramètres. Tout format de  $P^c$  peut être généré par le supremum d'un sous-ensemble d'au plus  $p$  éléments de  $P$ .

**Démonstration**

Soit un supremum  $F$  d'un sous-ensemble  $S$  de  $P$ .  $F$  a exactement  $p$  valeurs, chacune provenant d'un format de  $S$ . Si plus de  $p$  formats de  $S$  contribuent à fournir une valeur à  $F$ , celui-ci aurait plus de  $p$  valeurs. ■

Cela nous permet de définir une borne supérieure sur la cardinalité de  $P^c$ .

**Proposition 2.3**

Soient  $P$  un ensemble de formats à  $p$  paramètres et  $n = |P|$ . La cardinalité de  $P^c$  est au plus

$$\sum_{i=1}^p \binom{n}{i} \quad (2.1)$$

Cette limite peut être atteinte dans le cas  $p = 2$  quelque soit  $n > 0$  (c'est trivial pour  $n = 1$ , considérons  $n > 1$ ). En effet, il suffit de construire des formats de la forme  $(n + i, n - i + 1)$  pour  $i$  de 1 à  $n$ . Pour deux formats différents de cet ensemble, disons pour  $i_1 < i_2$ , leur supremum  $(n + i_2, n - i_1 + 1)$  est unique (de même pour le cas  $i_1 > i_2$ ). Ainsi il y a  $C(n, 2)$  nouveaux formats dans  $P^c$ .<sup>9</sup>

Une telle borne permet d'envisager un algorithme dont la performance serait dans  $O(p \sum_{i=1}^p C(n, i))$ . À titre d'exemple, pour  $n = 15$  et  $p = 3$ , un cas envisageable en pratique, il y aurait le calcul de 575 supremum. Pour des cas où  $n = 35$  et  $p = 4$ , cas encore probable en pratique, il y aurait 59535 supremum à évaluer. C'est toujours pratiquement réalisable, mais ce qui pose problème, avec une telle méthode, est l'évaluation d'autant de supremum sans tenir compte de d'autres aspects de la structure de  $P$ . En particulier, le fait que pour un sous-ensemble  $S$ , il n'y a souvent qu'une partie des formats contribuant à la formation du supremum.

Finalement, il faut considérer le fait, qu'en pratique, le nombre des longueurs possibles des paramètres des instructions est assez limité. On peut probablement envisager des cas où le nombre de ces longueurs est environ une dizaine, mais c'est rarement plus. Ainsi, il faudrait considérer la conception d'un algorithme qui tient compte de ce fait.

La figure 2.5 présente un algorithme de création de  $P^c$  étant donné  $P$ . La complexité de celui-ci est, au pire cas, dans  $O(|P|^p \prod_{i=1}^p l_i)$  où  $p$  est la longueur des tuples de  $P$  et  $l_i$  le nombre de valeurs différentes en position  $i$  des tuples de  $P$ . Le terme  $\prod_{i=1}^p l_i$  provient du nombre d'itérations pouvant se produire par la boucle 'Pour'. C'est le nombre maximum

9. La borne de 2.3 ne peut probablement pas être atteinte pour  $p > 2$  et  $n > p$ .

d'éléments du sur-ensemble des tuples considérés. Le terme  $|P|p$  est la complexité, en pire cas, du calcul de la couverture et du supremum. (Il est possible de faire l'union en un temps dans l'ordre de  $|C|$ . C'est possible puisqu'il suffit de vérifier si  $p.c \in C$  pour savoir si  $p.c \in R$ ; car  $C$  est un sous-ensemble de  $P$ , lui-même un sous-ensemble de  $R$ . Une fois ce test effectué, l'union se résume à un ajout sur une liste.)

L'algorithme passe parmi une partie des éléments d'un sur-ensemble des tuples pouvant devenir membre de  $P^c$ . Ce sur-ensemble est essentiellement les combinaisons des ensembles  $L$ . Aucun tuple n'appartenant pas à ce sur-ensemble ne peut appartenir à  $P^c$ . Les seuls tuples non considérés de ce sur-ensemble, sont ceux dont un préfixe ne couvre pas plus d'un élément de  $P$  (voir test  $|C| > 1$ ). La couverture  $C$  est l'ensemble des tuples de  $P$  dont le préfixe de longueur  $i$  est couvert par  $p.c$ ; c'est donc une couverture partielle des tuples de  $P$ , car  $p.c$  n'a pas en général la longueur  $n$ . Cela permet de diminuer l'ensemble des tuples à considérer lors des appels récursifs. Le test  $|C| > 1$  empêche de demeurer avec un préfixe  $p.c$  qui ne couvre qu'au plus un élément de  $C$ . Autrement ce préfixe ne peut former un tuple à ajouter à  $R$  car il ne couvre qu'au plus un élément de  $R$ , qui est donc déjà dans  $R$  (remarquez qu'au départ  $C = R$  et  $R$  ne perd jamais d'élément.). Le test ' $p.c = \text{supremum de } C$ ' vérifie si le tuple  $p.c$ , de longueur  $n$ , est vraiment le supremum de la couverture construite jusqu'à maintenant. Si c'est le cas, il s'agit nécessairement d'un tuple appartenant à  $P^c$  car tout supremum d'un sous-ensemble de  $P$ , et  $C$  est un sous-ensemble de  $P$ , est un élément de  $P^c$ .

L'algorithme de complétion est utilisé lors de la construction de la base des formats (voir section 2.4.6), et par la construction de la base des macro-instructions (voir section 2.3).

### 2.4.3 Les formats généraux

La section précédente a considéré les formats sans tenir compte de leur type. Cela était suffisant pour définir la complétion et décrire son algorithme. Toutefois, la génération des implantations des instructions, à partir de formats, nécessite la spécification du type des opérandes. C'est pourquoi nous introduisons la spécification du type de chaque argument d'un format.

De plus, la spécialisation par valeur se décrit plus facilement en introduisant des constantes dans les formats. Par exemple, une instruction (`pushi (c 2)`), signifie que `pushi` est un mnémonique à un argument, mais que l'argument est fixé à la valeur 2. Le dictionnaire final de formats, ainsi que de macro-instructions, peut contenir des formats avec constantes.

Voici la définition des formats généraux utilisés pour la suite.

**Algorithme 2.1** (Complétion d'un ensemble de formats)**Entrée** : un ensemble de formats  $P$  de même longueur.**Sortie** : la complétion de  $P$ , noté  $P^c$ . $P^c \leftarrow \text{CompletionR}(\lambda, 1, P, P)$  $\text{CompletionR}(p, i, P, R)$ Soit  $L$  l'ensemble des valeurs en position  $i$  des formats  $P$ Soit  $n$  la longueur des tuples de  $P$ Pour  $c \in L$  Faire    Soit  $p.c$  la concaténation de la valeur  $c$  au tuple  $p$     Soit  $C$  la couverture de  $p.c$  sur  $P$     Si  $|C| > 1$  Alors        Si  $i = n$  Alors Si  $p.c = \text{sup}(C)$  Alors  $R \leftarrow \{p.c\} \cup R$         Sinon  $R \leftarrow \text{completion}(p.c, i + 1, C, R)$ Retourner  $R$ 

FIG. 2.5 -. Algorithme de complétion des formats

**Définition 2.4.6**

Un format général est un tuple de la forme

$$(t_1 l_1 \ t_2 l_2 \ \dots \ t_p l_p)$$

où  $t_i = u, s$  ou  $c$  et les  $l_i$  sont des entiers positifs.

Le type  $u$  représente le type entier non-signé et le type  $s$  le type signé. Le nombre de bits d'un argument  $a$  dépend de son type. Dans le cas du type  $c$ , il s'agit d'une constante, ce n'est pas la longueur du paramètre. Nous utilisons la fonction suivante pour l'évaluation de la longueur d'un paramètre selon son type.

$$\|a\| = \begin{cases} \lceil \lg(a + 1) \rceil & \text{si } a \text{ est de type non-signé} \\ \lceil \lg(|a + 1/2| + 1/2) \rceil + 1 & \text{si } a \text{ est de type signé} \end{cases} \quad (2.2)$$

Note: dans le cas où  $a = 0$ , non-signé, le nombre de bits est zéro. Ce cas est équivalent à l'introduction d'une instruction ayant l'argument fixé à zéro. Il y a donc ici un léger chevauchement entre les spécialisations de type format et valeur.

Toutefois, notons qu'en général le format d'une instruction dépend du langage de base de la machine virtuelle. Ainsi la fonction de génération du format d'une instruction  $I$  doit être conçue pour chaque machine de base. La fonction  $\|\cdot\|$  est une approximation raisonnable pour la plupart des machines.

La longueur d'un format général est la somme des longueurs sans comptabiliser les constantes. Ainsi, la longueur d'un format d'un mnémonique représente bien le nombre de bits à utiliser pour coder ses arguments.

Les notions de supremum et de couverture de formats s'étendent à la forme générale de la façon suivante. La relation  $F_1 \geq F_2$  est vraie, ssi les longueurs de  $F_1$  peuvent coder les constantes de  $F_2$ ; et si  $F_1$  a des constantes,  $F_2$  doit avoir les mêmes constantes aux positions correspondantes. Ainsi,  $(c\ 2\ u\ 3) \geq (c\ 2\ u\ 1)$  et  $(u\ 3\ s\ 5) \geq (c\ 2\ c\ 3)$ ; mais  $(c\ 2\ u\ 3) \not\geq (u\ 2\ u\ 1)$  et  $(u\ 3\ u\ 3) \not\geq (c\ 10\ u\ 3)$ .

#### 2.4.4 Spécialisation de type valeur

La définition de format général permet de traiter, comme cas particulier, la spécialisation de type valeur. En effet, la longueur d'un format ayant des constantes ne comptabilise pas leur longueur. Ainsi, l'évaluation du gain d'un format, présentée à la section 2.4.1, va permettre d'évaluer l'introduction d'une valeur fixe pour un mnémonique.

La construction de la base des fréquences des formats doit introduire les formats avec constantes. De même, la construction de la base des macros-instructions (section 2.3), introduit des constantes dans les formats des macros-instructions. Ces préparatifs permettent de considérer tous les formats potentiellement utiles à la compression.

La section suivante présente les détails de la construction de la base des formats et, ainsi, de la présence des constantes dans les formats.

#### 2.4.5 Gain en espace d'un format

Nous envisageons la construction d'un algorithme vorace pour la sélection des formats à utiliser dans la machine virtuelle. Le travail principal d'un tel algorithme est de calculer le meilleur gain possible d'un ensemble de formats étant donné les formats déjà choisis. Le gain pour un format est la différence du nombre de bits entre ce format et le format actuellement utilisé, multiplié par sa fréquence d'utilisation. Toutefois, non seulement faut-il considérer la fréquence de ce format, mais aussi les fréquences de tous les autres formats plus petits, et non encore codifiables avec moins de bits pour ce même mnémonique. Puisque, toutes les instructions du même mnémonique, utilisant des formats plus petits, peuvent se codifier dans le nouveau format choisi. C'est pourquoi nous introduisons la notion de formats « effectivement couverts » par un format. Préalablement, nous définissons la notion

de supremum d'un format par rapport à un ensemble de formats.

#### Définition 2.4.7

Le **supremum du format  $F$  par rapport** à un ensemble de formats  $P$ , noté  $F^P$ , est le format appartenant à  $P$  étant plus grand ou égal à  $F$  et ayant la plus petite longueur<sup>10</sup> ; si plusieurs formats ont ces caractéristiques, c'est lexicographiquement le plus petit.

Le concept de supremum permet d'identifier, parmi un ensemble de formats  $P$ , le « bon » format qui couvre un format donné. En effet, ce supremum utilise le moins d'espace tout en permettant de codifier toutes les instructions du format  $F$ . Il peut être utilisé pour spécifier le format à utiliser lors de la codification d'une instruction.

Notez que plusieurs formats peuvent être plus grand qu'un certain format  $F$ , ayant tous la même longueur, tout en étant incomparables entre eux. Par exemple, parmi les formats  $(u\ 2\ u\ 4\ u\ 4)$ ,  $(u\ 2\ u\ 3\ u\ 5)$  et  $(u\ 4\ u\ 4\ u\ 3)$ , les formats  $(u\ 2\ u\ 4\ u\ 4)$  et  $(u\ 2\ u\ 3\ u\ 5)$  couvrent le format  $(u\ 1\ u\ 2\ u\ 2)$ , et ont la plus petite longueur (10) ; mais ils sont incomparables entre eux. Néanmoins, c'est le format  $(u\ 2\ u\ 3\ u\ 5)$  qui est préféré. Ainsi, l'ordre lexicographique départage ces formats<sup>11</sup>. Ce choix est arbitraire pour l'objectif de gain en espace et pourrait être remplacé par une caractéristique différente pour satisfaire à un second objectif<sup>12</sup>.

Voici la définition des formats à considérer lors du calcul du gain en espace pour un nouveau format.

#### Définition 2.4.8

Soient deux ensembles de formats  $P_1$ ,  $P_2$  et un format  $F$ . L'ensemble  $EC(F, P_1, P_2) = \{F_i \in P_1 \mid F = F_i^{P_2 \cup \{F\}}\}$  des formats de  $P_1$  sont **effectivement couverts** par  $F$  selon  $P_2$ .

Cette notion « d'effectivement couverts » permet de considérer seulement les formats de  $P_1$  pouvant être codés avec moins d'espace. En effet, si on suppose que l'ensemble  $P_2$  joue le rôle des formats actuellement disponibles pour la codification des instructions, pour un mnémonique donné, le format  $F$  n'est utile que si l'ensemble  $EC(F, P_1, P_2)$  n'est pas vide. C'est l'ensemble complet des formats devant être comptabilisés pour un gain en espace, si  $F$  est introduit comme nouveau format.

10. C'est la longueur selon la définition 2.4.4.

11. Il ne faut pas confondre « ordre lexicographique », qui est un ordre total, et l'ordre partiel défini en 2.4.3.

12. Un tel second objectif pourrait être l'utilisation du format ayant le plus petit maximum parmi ses longueurs. Par exemple, entre le format  $(u\ 2\ u\ 4\ u\ 4)$  et  $(u\ 2\ u\ 3\ u\ 5)$ , ayant la même longueur et couvrant le format  $(u\ 1\ u\ 3\ u\ 4)$ , le format  $(u\ 2\ u\ 4\ u\ 4)$  serait préféré. Un tel choix réduirait la manipulation de longs paramètres.

Voici une définition précise du gain d'un format pour un mnémonique.

**Définition 2.4.9**

Soit le mnémonique  $m$  à  $p$  paramètres. Soit l'ensemble des paires  $E = \{(F_i, f_i)\}$  où l'ensemble  $P = \{F_i\}$  sont les formats de base de  $m$  et  $f_i$  leur fréquence d'occurrence. Soient un sous-ensemble  $C \subseteq P$  couvrant  $P$ ; et  $P^c$  la complétion de  $P$ . La fonction  $g_{\text{format}}$  de gain d'espace du format  $F \in P^c$ , pour  $m$ , par rapport à  $C$  est définie de la façon suivante.

$$g_{\text{format}}(F, E, C) = \sum_{F_i \in EC(F, P^c, C)} (l_{F_i} - l_F) f_i \quad (2.3)$$

Dans cette définition, le terme  $l_{F_i} - l_F$  exprime le gain, en bits, du passage du format  $F_i$  au format  $F$ ; multiplié par la fréquence du format  $F_i$ , soit  $f_i$ , on obtient le gain d'espace pour ce format effectivement couvert par  $F$ . Il suffit alors de sommer sur tous les formats effectivement couverts par  $F$  pour obtenir le gain total. Notez que  $C$  est l'ensemble des formats déjà disponibles pour  $m$ .

La rapidité de l'évaluation de cette fonction peut être grandement améliorée dans le cas où le mnémonique n'a qu'un seul paramètre. Dans ce cas, le supremum de  $F$  est unique et les formats effectivement couverts par celui-ci ont tous le même supremum. Ainsi, l'expression  $(l_{F_i} - l_F)$  est constante pour un  $F$ . De plus, le calcul de l'ensemble  $EC(F, P^c, C)$  est beaucoup plus simple. En effet, il suffit de prendre l'infimum de  $F$ , par rapport à  $C$ , et de colliger les formats de  $P$  (notez que  $P^c = P$  dans le cas de formats à un seul paramètre) qui sont strictement plus grand que l'infimum et plus petit ou égal à  $F$ .

Cette valeur de  $g_{\text{format}}(F, P, C)$  est le gain maximum pouvant être obtenu en créant un nouveau format  $F$ , considérant que les formats  $C$  sont déjà utilisés. C'est le gain maximum, car cette fonction ne tient pas compte, entre autres, de l'espace perdu à cause de l'introduction d'un nouveau code opérationnel.

Pour tenir compte de l'augmentation de l'espace des codes opérationnels, ainsi que du décodeur, il faudrait considérer tous les mnémoniques, avec formats déjà choisis, et leur fréquence d'occurrence. Pour chaque couple  $(m_i, F_{m_i})$ , formé d'un mnémonique et d'un format, un code opérationnel lui est attribué.

Soit l'ensemble  $M = \{((m_i, F_{m_i}), f_{m_i})\}$  de mnémoniques avec format de fréquence  $f_{m_i}$ . L'espace des codes opérationnels des couples  $(m_i, F_{m_i})$  est la longueur des codes multiplié par leur fréquence. Cette valeur est notée  $e_c(M)$ . Les longueurs des codes sont déterminées par l'algorithme de Huffman. L'espace du décodeur pour  $M$  est noté  $e_d(M)$ .

Voici une définition du gain réel de l'ajout d'un format.

**Définition 2.4.10**

Soient un mnémonique  $m$ ,  $E = \{(F_i, f_i)\}$  l'ensemble des formats pour  $m$  des instructions échantillons avec fréquence,  $P = \{F_i\}$ ,  $C$  l'ensemble des formats déjà définis pour  $m$ ,  $F \in P$ ,  $M = \{((m_i, F_{m_i}), f_{m_i})\}$  et  $M' = M \cup \{((m, F), f_F)\}$ . Le gain réel en espace est défini comme suit.

$$gr_{format}(F, E, C, M', M) = g_{format}(F, E, C) - (e_c(M') - e_c(M)) - (e_d(M') - e_d(M)) \quad (2.4)$$

Ainsi, la soustraction des termes  $(e_c(M') - e_c(M))$  et  $(e_d(M') - e_d(M))$  du gain, tient compte de l'augmentation de l'espace des codes opérationnels et du décodeur. L'évaluation de  $e_c(M')$  est coûteuse en temps, car elle demande une exécution de l'algorithme de Huffman. L'évaluation de  $e_d(M')$ , peut être très coûteuse si nous désirons une évaluation exacte<sup>13</sup>. Toutefois, nous utilisons une approximation ; soit le nombre de codes opérationnels multiplié par le nombre moyen d'octets implantant une instruction virtuelle.

La fonction d'évaluation de l'espace du décodeur, soit  $e_d$ , est fournie par le concepteur. C'est ici qu'il est possible de considérer les différents contextes de l'utilisation de la machine virtuelle, dont nous avons discutés à la section 1.6. Si la taille de cette machine n'a aucune importance, la fonction  $e_d$  peut alors toujours valoir zéro. Si, au contraire, la taille de la machine virtuelle a de l'importance, cette fonction devrait l'évaluer le plus précisément possible.

*2.4.6 Processus de construction de la base des formats.*

Une base de formats est un ensemble de couples  $(m, \mathcal{F})$  où  $m$  est un mnémonique et  $\mathcal{F}$  est un ensemble de couples  $(F, f)$  où  $F$  est un format général et  $f$  sa fréquence d'occurrences. Cette base est construite à partir d'un ensemble de programmes échantillons.

Le processus général de construction de la base est exposé à la figure 2.6. Les programmes sont les échantillons considérés. La première étape consiste à produire, pour chaque groupe  $i$  d'échantillons, deux bases temporaires : l'une  $B'_c$ , pour les constantes et une autre  $B'_{f_i}$  pour les formats sans constantes.

Dans ses grandes lignes, la construction d'une base temporaire  $D'_{f_i}$  procède comme suit. Chaque instruction  $I = (m \ a_1 \dots a_p)$  produit une occurrence d'un format  $F = (t_1 \ ||a_1|| \dots t_p \ ||a_p||)$  sans constante. Toutes les occurrences sont comptabilisées produisant une fréquence pour chaque format.

Pour un même mnémonique, la complétion de l'ensemble de ses formats est ajouté à la base. La fréquence d'un format ajouté par l'algorithme de complétion est zéro. Cela est

13. Les termes  $e_d(M)$  et  $e_c(M)$  proviennent de l'itération précédente ; il n'y a pas lieu de les recalculer.

correct, car c'est l'ensemble des formats couverts qui est considéré lors du calcul du gain en espace.

Les formats  $F$  générés dépendent du langage de base de la machine virtuelle, et certaines particularités du langage doivent être appliquées par cette génération ; en particulier, le problème des instructions de branchement utilisant un déplacement relatif. Les arguments de telles instructions ne peuvent être utilisés directement par la fonction  $\|\cdot\|$  (cf. équation 2.2). Par exemple, pour *Machina*, nous estimons le nombre de bits nécessaires par les arguments des instructions *br* et *bf* du code exécutable, en utilisant  $\|8a_i\|$ .

Les bases  $D'_{c_i}$  contiennent les occurrences des valeurs des arguments pour chaque mnémotique. À chaque paramètre en position  $j$  d'un mnémotique, il y a une liste de couples  $(c_j, f_j)$  des valeurs, avec fréquences, apparaissant comme argument dans les programmes échantillons. Le couple est conservé seulement si la fréquence est supérieure à un seuil minimum contrôlé par le concepteur. Cet élagage permet de garder seulement les cas prometteurs. En pratique, il suffit d'utiliser une fréquence minimum sous laquelle aucun gain de compaction n'est possible. Ce seuil peut être évalué en considérant l'espace nécessaire d'une nouvelle instruction virtuelle sur la machine hôte<sup>14</sup>. Dans d'autres situations, là où l'échantillon est volumineux, le concepteur peut élever le seuil pour diminuer le temps de construction et, surtout, le temps de l'exécution de l'algorithme de création d'instructions virtuelles.

Les deux bases  $D'_{c_i}$  et  $D'_{f_i}$  sont combinées, selon le processus suivant, pour ne former qu'une seule base  $D_{f_i}$  de formats généralisés. L'objectif est de produire tous les formats possibles ayant des constantes. C'est donc dire que nous générons toutes les combinaisons des constantes avec les formats n'ayant pas de constantes. À première vue, cela apparaît prohibitif, mais nous appliquons le principe de la fréquence du seuil minimum. Un format ayant au moins une constante doit avoir une fréquence minimum. Cela diminue considérablement le nombre de formats avec constante, si le seuil représente une valeur raisonnable par rapport à l'échantillon. La fréquence de chaque format avec constante est calculée à partir des données de  $D'_{c_i}$ .

Si un format  $F_c$  avec, au moins, une constante de fréquence  $f_c$ , est ajouté à la base, il faut diminuer la fréquence du format  $F$  couvrant  $F_c$  de  $f_c$ . De cette façon, le calcul du gain en espace de  $F$  sera correct, puisque nous utilisons la notion d'effectivement couvert pour calculer le gain en espace d'un format.

La dernière étape consiste à réunir tous les  $D_{f_i}$  pour ne former qu'une seule base  $D_f$ . Cette combinaison utilise des poids  $p_i$  pour chaque base. Ce poids reflète l'usage de chaque échantillon dans la machine virtuelle. La combinaison consiste en une union pondérée des

14. Cet élagage ne peut être facilement fait dans le cas du dictionnaire temporaire  $D'_{f_i}$  des formats ; car un format contribue, par la notion d'effectivement couvert, au gain en espace de d'autres formats plus grands.

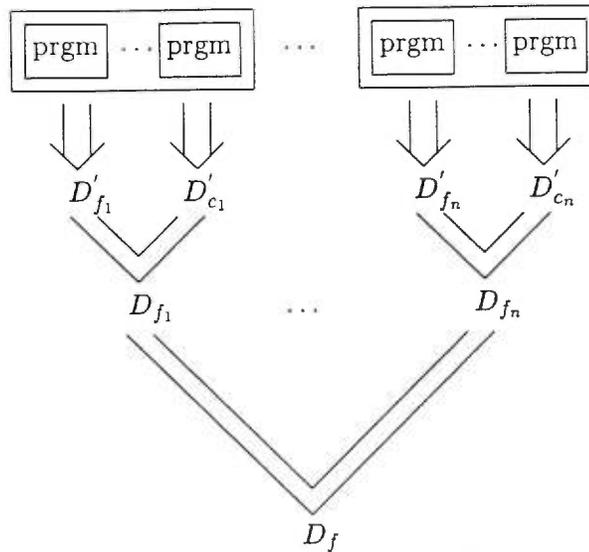


FIG. 2.6 –. Processus de création d'un dictionnaire de formats  $D_f$

listes des formats pour chaque mnémonique : la fréquence d'un format de  $D_f$  est la somme pondéré des fréquences des formats dans les  $D_{f_i}$ .

Plus précisément, soient les poids  $p_i$ . La combinaison des bases est définie par

$$D_f = \{(m, \mathcal{F}) \mid \mathcal{F} = \{(F, t) \mid t = \sum_{(m, \mathcal{F}_i) \in D_{f_i}} \sum_{(F, f) \in \mathcal{F}_i} f p_i; t \neq 0\}; \mathcal{F} \neq \emptyset\}$$

Ainsi la base finale  $D_f$  contient des formats typés ayant potentiellement des constantes. C'est cette base qui sera utilisée par l'algorithme général de création des instructions de la figure 2.7.

## 2.5 Sélection des formats

La sélection des formats fait partie de l'algorithme de création des instructions de la figure 2.7. Il s'agit principalement de choisir le format dont le gain est maximum. Cela est fait par la fonction *format-gain-maximum*. Celle-ci applique simplement la fonction  $gr_{\text{format}}$  à chaque mnémonique et format disponible dans la base des formats. Le gain maximum en espace est alors retourné accompagné du format et du mnémonique. Ce gain compétitionne avec le gain maximum pour une macro-instruction.

Si le format est sélectionné, il faut mettre à jour la base des formats. Cette mise à jour est effectuée par la fonction *ajuste-base-formats*. Celle-ci diminue la fréquence du supremum du format par rapport aux formats choisis. Bien entendu, la sélection d'un format peut modifier le gain d'une macro-instruction. Toutefois, la base des macro-instructions n'est pas modifiée, car le calcul du gain maximum d'une macro-instruction utilise l'information courante des formats sélectionnés.

## 2.6 Sélection des macro-instructions

Le processus de sélection choisit des macro-instructions à utiliser dans la machine virtuelle, parmi une base préalablement construite. Cette sélection se fait en même temps que la spécialisation des instructions.

Une macro-instruction n'est choisie que si celle-ci permet de réduire l'espace mémoire du programme. Son apport est mesuré par l'espace mémoire des instructions qu'elle remplace moins l'espace pour la représenter dans la machine virtuelle.

La recherche d'une solution optimum est, dans les deux cas, NP-complet<sup>15</sup>. Nous utilisons donc des heuristiques pour trouver une solution non optimum.

Pour évaluer le gain en espace d'une macro-instruction, il faut calculer la somme des bits des instructions remplacées moins le nombre de bits utilisés par la représentation de la macro. L'évaluation des longueurs des instructions remplacées devrait tenir compte des longueurs des codes opérationnels et des arguments. Cette évaluation doit donc faire référence aux codes opérationnels et aux formats actuels. C'est pourquoi, dans la formulation suivante, il y a une référence aux mnémoniques et formats codifiant les instructions de la séquence. Sans cela, on supposerait qu'il serait possible de codifier les instructions de la séquence de façon optimum ; ce qui serait loin de la réalité.

Voici plus précisément la fonction d'évaluation du gain en espace d'une macro-instruction.

### Définition 2.6.1

Soit une macro-instruction  $m$  dont la séquence d'instructions  $S$  est de format  $F_m$  et de fréquence d'occurrence  $f_m$ . Soient  $(m_i, F_i)$  les mnémoniques  $m_i$  et leur format  $F_i$  des instructions codifiant les instructions de  $S$  et  $l(c(m_i, F_i))$  les longueurs des codes opérationnels des  $(m_i, F_i)$ . Soient  $l(c(m, F_m))$  la longueur du code opérationnel pour la macro-instruction  $(m, F_m)$  et  $l(F_m)$  la longueur de son format  $F_m$ .

Le gain en espace de la sélection de  $m$  est

$$g_{macro}(m, S, F_m) = f_m(-l(F_m) - l(c(m, F_m))) + \sum_{(m_i, F_i)} l(c(m_i, F_i)) + l(F_i) \quad (2.5)$$

---

15. Pour une preuve voir [GJ79], page 231, le problème *External Macro Data Compression*.

Naturellement, si la valeur est négative, il s'agit d'une perte et non d'un gain d'espace.

Lors de l'utilisation de cette évaluation, des mnémoniques avec formats ont été choisis et des codes opérationnels leur ont été attribués. Mais, la longueur du code opérationnel de la macro-instruction, soit  $l(c(m, F_m))$ , est inconnue, car elle n'a pas encore été sélectionnée. L'évaluation exacte de cette longueur est coûteuse à effectuer, il faudrait appliquer l'algorithme de Huffman en utilisant toutes les fréquences des codes opérationnels des instructions choisies, plus la fréquence de la macro-instruction. Nous utilisons plutôt une approximation, soit la longueur du code opérationnel déjà existant dont la fréquence est la plus près de  $f_m$ .

Notez que cette définition est, d'emblée, une approximation. Il faudrait tenir compte de l'ensemble de tous les codes opérationnels pour évaluer exactement le gain.

Dans le cas où  $F_m$  n'est pas un format provenant de l'algorithme de complétion, les formats  $F_i$  peuvent être construits à partir de celui-ci. Un format dont la liste d'emplacements est vide, est un format ajouté par l'algorithme de complétion. Dans ce cas, il faut sommer les valeurs de  $g_{\text{macro}}$  pour chaque format couvert par  $F_m$ .

L'ajout d'une macro augmente l'espace du décodeur. Ainsi, comme dans le cas de l'ajout d'un format, il faudrait tenir compte de cette augmentation. La fonction d'évaluation réelle de l'ajout d'une macro est donnée par la définition suivante.

### Définition 2.6.2

Soient l'espace du décodeur  $e_d(M)$  tel que défini en 2.4.5, et  $M' = M \cup \{(m, F_m), f_{F_m}\}$ . Le gain réel en espace de la sélection de  $m$  est

$$gr_{\text{macro}}(m, S, F_m, M', M) = g_{\text{macro}}(m, S, F_m) - (e_d(M') - e_d(M)) \quad (2.6)$$

Les commentaires de la section 2.4.5 à propos de l'évaluation de  $e_d(M')$  s'appliquent aussi dans ce cas. En résumé, c'est le concepteur de la machine virtuelle qui fournit cette fonction.

Si une macro est choisie, elle fait partie de la machine virtuelle et ne sera pas retirée par la suite. Ce choix implique que d'autres macros perdent de l'importance, car des chevauchements existent dans la base de macro-instructions. La base est mise à jour pour éliminer les emplacements chevauchés et ajuster la fréquence d'occurrences. C'est l'opération la plus coûteuse dans l'algorithme de création des instructions virtuelles. Elle est effectuée par la fonction *ajuste-base-macros*.

## 2.7 Algorithme de création des instructions virtuelles

Nous présentons, dans cette section, l'algorithme général pour la création de nouvelles instructions virtuelles. Cet algorithme réunit les concepts et algorithmes de ce chapitre.

L'algorithme de la figure 2.7 spécialise des instructions, sélectionne des macro-instructions et fait une attribution finale des codes opérationnels. Il y a plusieurs attributions de codes opérationnels, c'est seulement la dernière qui est gardée; le résultat est donc un ensemble d'instructions spécialisées selon des formats, des macro-instructions à garder dans la machine virtuelle et des codes opérationnels pour toutes ces instructions. Ce résultat est exprimé sous la forme de deux dictionnaires: l'un contenant les macro-instructions avec leur format et l'autre les formats pour tous les mnémoniques, incluant les macro-instructions, et l'attribution des codes opérationnels.

Essentiellement, l'algorithme débute par une attribution de codes opérationnels sans créer de nouvelles instructions. Par la suite, certaines instructions ayant des paramètres sont spécialisées ou des macro-instructions sont appliquées. Le processus est répété jusqu'à l'obtention d'un ensemble d'instructions qui ne peut être amélioré.

Les fonctions *format-gain-maximum* et *macro-gain-maximum* ont été décrites aux sections 2.5 et 2.6 respectivement.

La fonction *HuffmanCanonique* attribue des codes opérationnels à un ensemble de couples  $(mne, format)$ , plus potentiellement un nouveau couple d'une fréquence donnée.

Lors des calculs des gains en espace, il est nécessaire de tenir compte de l'augmentation en espace de l'interprète; autrement, le gain en espace est transféré, sans en tenir compte, au code exécutable de son implantation. Cette augmentation a deux composantes: l'espace occupé par les instructions virtuelles et l'espace du décodeur. Cet espace dépend de la machine hôte et du décodeur employé. Cette évaluation est laissée à la fonction  $e_d$ . Elle utilise l'ensemble des mnémoniques avec formats ainsi que l'attribution des codes opérationnels.

La fonction *ajuste-base-macros* modifie la base de macros pour tenir compte des chevauchements. La fonction *ajuste-base-formats* modifie la base de formats en réduisant les fréquences d'occurrences étant donné la sélection d'une macro-instruction ou d'un nouveau format.

## 2.8 Compression des programmes

Comme présentée sommairement à la section 1.5 (voir le diagramme), la compression des programmes utilise les dictionnaires de macro-instructions et de formats. Nous élaborons ici quelques détails techniques pertinents à propos de cette compression.

La compression est effectuée en deux étapes. La première substitue les séquences d'instructions définies comme macro-instruction. Nous n'appliquons pas un algorithme optimum, mais un heuristique simple; les macros sont appliquées en débutant par les plus longues.

Par exemple, supposons que la macro-instruction `(pushl_pushi ((pushl *) (pushi *) (u 5 s 7)))` existe dans le dictionnaire. Une séquence `((pushl 2) (pushi 18))`, ap-

**Algorithme 2.2 (Création des instructions virtuelles)**

**Entrée :** une base  $B_m$  de macro-instructions, une base  $B_f$  de formats des mnémoniques de base, un dictionnaire  $D_c$  des choix initiaux des formats maximum des mnémoniques et une fonction  $e_d :: D_f \rightarrow \text{entier}$  où  $D_f$  est un dictionnaire d'attribution de codes opérationnels à des mnémoniques avec formats.

**Sortie :** un dictionnaire  $D_m = \{(mne, seq, format)\}$  de macro-instructions, un dictionnaire  $D_f = \{(mne, format, codeop)\}$  de formats pour tous les mnémoniques avec attribution de codes opérationnels.

$D_f \leftarrow \text{HuffmanCanonique}(D_c, (), 0)$   
 $D_m \leftarrow \{\}$

Répète

$(v_m, fq_m, (g_m, (mne_m, s, f_m))) \leftarrow \text{macro-gain-maximum}(B_m, D_f, e_d)$

$(v_f, fq_f, (g_f, (mne_f, f_f))) \leftarrow \text{format-gain-maximum}(B_f, D_f, e_d)$

Si  $v_f > 0$  ou  $v_m > 0$  Alors

Si  $v_f > v_m$

Alors  $D_f \leftarrow \text{HuffmanCanonique}(D_f, (mne_f, f_f), fq_f)$

$B_f \leftarrow \text{ajuste-base-formats}((mne_f, f_f), B_f)$

Sinon  $D_m \leftarrow D_m \cup \{(mne_m, s, f_m)\}$

$D_f \leftarrow \text{HuffmanCanonique}(D_f, (mne_m, f_m), fq_m)$

$B_m \leftarrow \text{ajuste-base-macros}((mne_m, s, f_m), B_m)$

$B_f \leftarrow \text{ajuste-base-formats}((mne_m, s, f_m), B_f)$

jusqu'à  $v_s \leq 0$  et  $v_m \leq 0$

**FIG. 2.7 – Algorithme de création de versions spécialisées d'instructions, la sélection de macro-instructions et d'attribution de codes opérationnels**

paraissant dans le programme à compresser, sera remplacée par (`pushl_pushi 2 18`). La substitution est effectuée seulement si aucune instruction de branchement ne se produit dans la macro-instruction. Pour cet exemple, il s'agirait d'un branchement sur (`pushi 18`). De plus, le format de la macro-instruction doit être capable de représenter les arguments. Notez que plusieurs macro-instructions peuvent avoir la même séquence de base, mais varier par leur format. Cependant, à cette étape, il n'est pas nécessaire de choisir le format optimum pour la séquence. Ce choix est effectué à la deuxième étape.

Pour chaque substitution, les arguments des instructions de branchement sont modifiés pour franchir correctement les macro-instructions.

À la deuxième étape, chaque instruction est codée sous la forme binaire. Cela produit une longue séquence de bits, en général sans alignement. Toutefois, certaines instructions demandent un alignement sur une frontière d'octet. L'ensemble de ces instructions dépend de la machine virtuelle. Ces instructions sont rares : pour la JVM, ce sont les mnémoniques `jsr` et `jsr_w` ; pour Machina, c'est le mnémonique `jsr`.

Pour chaque instruction, les arguments déterminent le format à utiliser. Pour trouver le format optimum, c'est le supremum du format formé par les constantes qui est utilisé. Plus précisément, soit une instruction ( $m \ a_1 \dots a_p$ ) à coder. Soit  $P$  l'ensemble des formats disponibles dans le dictionnaire des formats pour le mnémonique  $m$ . Le format  $F_c = (c \ a_1 \dots c \ a_p)$  est préalablement construit. Le format optimum est alors le format supremum de  $F_c$  par rapport à  $P$ . Notez que ce calcul traite convenablement les cas où  $P$  contient des formats avec constantes. Par exemple, si  $P = \{(c \ 2 \ u \ 3), (u \ 3 \ u \ 3)\}$  et  $F_c = (c \ 2 \ c \ 4)$ , le supremum de  $F_c$  par rapport à  $P$  sera  $(c \ 2 \ u \ 3)$ . Ce qui représente bien le format optimum pour les deux arguments 2 et 4. Si  $F_c$  est plutôt  $(c \ 7 \ c \ 4)$ , c'est le format  $(u \ 3 \ u \ 3)$  qui est le supremum par rapport à  $P$ .

Le mnémonique et le format déterminent un code opérationnel unique spécifié par le dictionnaire de formats. C'est une identification identique pour les instructions de base et les macro-instructions.

La codification des arguments s'effectue de façon normale, c'est-à-dire sous la forme binaire en complément à deux. Si le format indique la présence d'une constante, l'argument correspondant est bien sûr non codifié.

Une dernière complication qui doit être traitée avec minutie est l'ajustement des arguments des instructions de branchement. Sous leur forme préliminaire, les arguments spécifient le nombre d'instructions à franchir. Ils doivent plutôt être en bits. Cette modification est effectuée en supposant l'usage des formats les plus longs pour les instructions de branchement. Les instructions sont codées et le processus est réitéré tant qu'il est possible de réduire, au moins, un format pour les instructions de branchement.

## Chapitre 3

### LE DÉCODAGE DES INSTRUCTIONS VIRTUELLES

---

Le chapitre précédent s'est concentré sur la conception des instructions, plus particulièrement sur leur codification. Dans ce chapitre nous abordons le problème inverse, c'est-à-dire celui du décodage des instructions.

Le décodage doit reconnaître les instructions à exécuter et extraire leurs arguments. Naturellement, nous nous intéressons à des techniques rapides et peu coûteuses en espace mémoire.

#### **3.1 Introduction aux techniques de décodage Huffman**

Plusieurs travaux ont abordé la décompression de suites de codes de Huffman [Tan87, Sie88, Chu97, Kle97, TM98, Nek98, Has97, MT97]. Ces techniques peuvent être adaptées à la décompression de programme durant l'exécution. Nous retenons deux méthodes : celle de l'automate fini déterministe et la méthode dite « canonique ».

Ces deux méthodes seront évaluées du point de vue de la rapidité de décodage et de l'espace mémoire utilisé. Dans la plupart des cas, la méthode canonique utilise moins d'espace mémoire ; elle peut être toutefois plus lente que l'automate déterministe. Bien que les auteurs de [MT97] démontrent que la méthode canonique est plus rapide que la méthode de l'automate fini déterministe, ce résultat ne s'applique que dans un cas particulier où l'automate déterministe utilise peu d'espace mémoire. Il n'est donc pas, à priori, dénué d'intérêts d'évaluer ces deux méthodes. Elles offrent la possibilité de laisser le concepteur de la machine virtuelle troquer le temps de décodage pour l'espace mémoire, et vice versa. Cette caractéristique est importante pour permettre la conception d'une machine adaptée à des contextes différents.

Essentiellement, l'automate déterministe permet de décoder un nombre fixe de bits à chaque cycle de décodage, tandis que la méthode canonique utilise un nombre variable de bits. Cette dernière décode toujours une seule instruction par cycle, tandis que la première peut décoder plus d'une instruction par cycle.

Dans les sections suivantes nous aborderons plus en détails ces deux techniques.

#### **3.2 Stockage et lecture des instructions en mémoire**

Les implantations des décodeurs utilisent trois variables pour charger, décoder et lire les instructions de la mémoire centrale. La figure 3.1 présente l'essentiel de la technique de

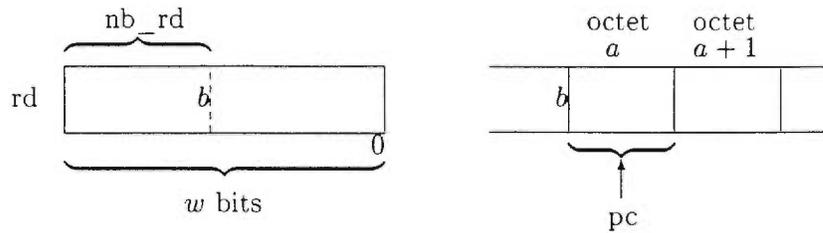


FIG. 3.1 – Technique de base d'accès au programme compressé

lecture des instructions du programme. La variable `rd`, d'une capacité de  $w$  bits, contient dans sa partie haute, un nombre variable de bits du programme. Le nombre de bits est contenu dans la variable `nb_rd`. La variable `pc` pointe le prochain octet à charger dans `rd`. Dans la figure, le bit  $b$  montre clairement la position du prochain octet à charger par rapport au contenu de `rd`.

Ainsi, `rd` peut contenir plus d'une instruction ; la variable `pc` ne pointant pas nécessairement sur la prochaine instruction. Il y a toujours un chargement d'un multiple de huit bits de la mémoire centrale dans la variable `rd` ; le pointeur de programme `pc` est donc un pointeur d'octet et non de bits. Le chargement d'octets se fait nécessairement quand au moins huit bits sont libres dans `rd`.

Il n'y a aucune « inversion » de bits lors du chargement des octets dans `rd`. Ainsi, en supposant la partie haute des octets à gauche sur la figure 3.1, les préfixes uniques (Huffman) des codes opérationnels apparaissent à gauche dans les octets.

Tous les arguments des instructions apparaissent en mémoire sous la forme « big-endian ».

### 3.3 Les décodeurs automates

Le décodeur automate est l'une des deux approches proposées, dans cette thèse, pour décoder rapidement les codes opérationnels de Huffman. Nous appelons ces décodeurs « automates », car ils sont similaires à un automate fini déterministe où l'état courant détermine un contexte sur lequel le prochain symbole lu est interprété. L'autre approche est le décodeur canonique, présenté à la section 3.4.

Pour un décodeur automate, le nombre de bits décodés à chaque cycle est fixé une fois pour toute. L'objectif général est d'éliminer au moins  $k$  bits à chaque cycle de décodage. Dans le cas du décodage canonique, le nombre de bits éliminés de l'entrée couvre exactement une instruction, incluant les paramètres. L'intention du décodeur automate fini déterministe est d'utiliser au maximum l'information fournie par la lecture de  $k$  bits. La complication générale d'une telle approche est qu'un cycle de décodage ne correspond plus exactement

à une instruction à exécuter. Plusieurs instructions peuvent être reconnues par un cycle de décodage. Cela provoque l'introduction d'instructions combinées, similaires à des macro-instructions.

La construction d'un tel décodeur offre peu de choix sur sa forme, bien que des variations sont possibles sur la façon d'implanter l'automate. Entre autre, cette construction peut être faite sans évaluation préalable de paramètres d'espace et de temps pour la machine hôte. Ce qui est différent de la méthode de construction du décodeur canonique.

Le désavantage majeur de la technique de l'automate fini déterministe, par rapport au décodeur canonique, est l'espace mémoire pour représenter les tables de décodage et le code de l'interprète. En particulier, la multiplicité des instructions combinées peut augmenter substantiellement la taille de l'interprète lui-même. Nous analysons attentivement ce problème et proposons plusieurs méthodes pour y remédier.

Les sections suivantes expliquent la méthode de construction de l'automate décodeur, la structure résultante et son implantation. Les sections 3.3.3 abordent des méthodes pour diminuer l'espace de l'automate. Les figures 3.6 et 3.13 présentent, entre autres, deux exemples de code C de décodeurs automates.

### 3.3.1 La construction de la structure d'un décodeur automate

Nous allons concevoir un algorithme de construction de décodeurs dont le principe général est d'utiliser, à chaque cycle, l'information contenue dans  $k$  bits du programme à décoder. Une telle construction se fera à partir d'un ensemble de tuples  $(c, m, F)$  où  $c$  est un code opérationnel,  $m$  est un mnémonique et  $F$  un format. Cette entrée est suffisamment générale pour permettre l'existence de plusieurs mnémoniques ayant différents formats. Il est aussi nécessaire de connaître les mnémoniques exigeant de remettre le décodeur à l'état initial après leur exécution. Cette information est fournie sous la forme d'une fonction booléenne à un paramètre  $Iz(m)$  où  $m$  est un mnémonique. Bien entendu, la valeur  $k$  est aussi choisie par le concepteur de la machine ; c'est un paramètre essentiel de l'algorithme déterminant son espace et sa performance.

Au début du cycle de décodage, le décodeur est dans un état symbolisé par une chaîne de bits. L'état initial est la chaîne vide (notée  $\lambda$ ). Il y a une lecture de  $k$  bits du programme. La chaîne formée par la concaténation de l'état et de ces  $k$  bits forme un *bloc de décodage*.

Pour mieux comprendre la suite, notez que le décodeur sera un ensemble de couples  $(e_i, L_{e_i})$  où  $e_i$  est une chaîne de bits, c'est-à-dire un état, et  $L_{e_i}$  une liste de  $2^k$  couples  $(s_j, e_j)$  où  $e_j$  sera le prochain état et  $s_j$ , dans l'état  $e_i$ , sera la séquence d'instructions à exécuter, si les  $k$  bits valent  $j$ .

Nous allons analyser la forme des séquences d'instructions  $s_j$  pouvant être générées par la

- Forme complète**  $((m_1 \ a_{11} \dots a_{1p_1}) \dots (m_n \ a_{n1} \dots a_{np_n}))$   
 où, pour  $m_i$ ,  $a_{ij}$  sont des arguments entiers  
 et  $p_i$  le nombre d'arguments
- Forme partielle**  $((m_1 \ a_{11} \dots a_{1p_1}) \dots (m_n \ a_{n1} \dots a_{nr} \ b))$   
 où  $r < p_n$  et  $b$  est une chaîne de bits

FIG. 3.2 -. Formes générales des séquences

construction de l'automate ainsi que les propriétés des états  $e_j$ . Cela dépend de la longueur du bloc de décodage, mais les cas possibles sont essentiellement les suivants.

- Cas 1 :** Aucune instruction n'est déterminée. Le décodeur passe par un état intermédiaire, représenté par tous les bits du bloc de décodage. Le prochain état est nécessairement non nul.
- Cas 2 :** Une ou plusieurs instructions sont déterminées avec tous les arguments de ces instructions. L'état suivant est formé par les bits demeurant après le dernier argument de la dernière instruction. Cet état peut être nul, c'est-à-dire être l'état initial.
- Cas 3 :** Une ou plusieurs instructions sont déterminées avec des arguments, à l'exception de la dernière instruction manquant ses derniers arguments. L'état suivant est nécessairement nul.

Le cas 1 produit une séquence vide d'instructions ; il n'y a aucune instruction à exécuter, car le décodeur passe par un état intermédiaire. Les formes des séquences proviennent des cas 2 et 3. Tous les états non nuls sont produits par les cas 1 et 2<sup>1</sup>.

On peut résumer ces cas par les formes générales que peuvent prendre la séquence d'instructions couvertes par un bloc de bits et l'état suivant. La figure 3.2 présente ces deux formes. La forme complète peut générer un état suivant non nul, mais la forme partielle génère toujours un état suivant nul, car la présence d'un argument partiel implique que le bloc de décodage a été complètement utilisé. La chaîne de bits  $b$  contient une partie du  $(r + 1)$ ème argument de  $m_n$ . Cette chaîne devra être combinée aux prochains bits du programme durant son exécution pour former la valeur de l'argument. Si elle est vide, cela signifie qu'il manque au moins un argument pour la dernière instruction de la séquence.

Ainsi, la construction de l'automate peut se résumer au processus suivant : produire toutes les séquences d'instructions pour les  $2^k$  chaînes de bits, pour chaque état possible. L'ensemble des états possibles est déterminé par ce même processus.

La figure 3.3 présente un tel algorithme de construction d'automates décodeurs.

---

1. Naturellement, il y a d'emblée l'état initial nul.

**Algorithme 3.1** (Construction des automates décodeurs)

**Entrée :** ensemble  $C = \{(c,m,f)\}$ ,  $|C| > 1$ , de codes préfixes  $c$  ayant mnémonique et format associé  $m,f$ , et un entier  $k > 0$ .

**Sortie :** un automate décodeur a racine unique de  $k$  bits.

```

1  Soit  $A$  l'arbre binaire des codes  $\{c\}$ 
2  Retourner  $AutomateR(A,C,k,Iz,\lambda,\{\lambda\})$ 

3   $AutomateR(A,C,k,Iz,e,E)$ 
4    Soit  $L^e = [(s_i,e_i) \mid 0 \leq i < 2^k, (s_i,e_i) = S(A,C,e : b(i,k),Iz)]$ 
5    Soit  $E^e = \{e_i \mid (s_i,e_i) \in L^e\}$ 
6     $E' \leftarrow E \cup E^e$ 
7     $Ls \leftarrow (e,L^e) : ()$ 
8    Pour chaque  $e_i \in E^e - E$  Faire
9      Soit  $Ls^{e_i} = AutomateR(A,C,k,Iz,e_i,E')$ 
10      $Ls \leftarrow Ls^{e_i} \cup Ls$ 
11      $E' \leftarrow E' \cup \{e_j \mid (e_j,L) \in Ls^{e_i}\}$ 
12  Retourner  $Ls$ 

13  $S(A,C,b,Iz)$ 
14   $b' \leftarrow b; \nu \leftarrow A$ 
15  Répéter
16    Si  $\nu$  est une feuille Alors
17      Soit  $(c,m,f) \in C; a = (a_1, \dots, a_p) = b'_f; b' \leftarrow b' - b'_f$ 
18      Si  $Iz(m)$  Alors Retourner  $(s : (m,f,a : b'), \lambda)$ 
19      Sinon  $\nu \leftarrow A; s \leftarrow s : (m,f,a); c \leftarrow \lambda$ 
20    Sinon Si  $b' \neq \lambda$  Alors Soit  $b' = x : b''; b' \leftarrow b''; c \leftarrow x : c$ 
21      Si  $x = 0$  Alors  $\nu \leftarrow \nu.gauche$  Sinon  $\nu \leftarrow \nu.droite$ 
22    Sinon Retourner  $(s,c)$ 

```

FIG. 3.3 -. Algorithme de construction des automates décodeurs

Le travail préliminaire est la création de l'arbre binaire  $A$  des codes à préfixe unique  $c$ . C'est pour une raison de vitesse d'exécution que cette construction est effectuée préalablement à l'appel de *AutomateR*. En effet, l'arbre binaire rend l'exécution de la fonction  $S$  plus rapide. L'argument  $\lambda$  spécifie un état vide, c'est-à-dire l'état initial du décodeur. Ainsi, l'argument  $\{\lambda\}$  représente l'ensemble initial des états construits ou en construction.

La tâche principale est effectuée par la fonction *AutomateR*. Les paramètres  $A$ ,  $C$ ,  $k$  et  $Iz$  ne changent pas pour tous les appels récursifs. Ce sont respectivement l'arbre binaire des codes opérationnels, une fonction associant à chaque code opérationnel un mnémonique unique et un format des paramètres, le nombre de bits du bloc de décodage et la fonction déterminant si une instruction exige un retour à l'état initial. Le paramètre  $e$  est une chaîne de bits : c'est l'état actuel du décodeur, c'est-à-dire les bits non encore utilisés mais déjà rencontrés par le décodeur. L'ensemble des états déjà construits est représenté par  $E$ . Analysons plus en détail le fonctionnement de l'algorithme.

$L^e$  est une liste de couples<sup>2</sup>. La notation  $b(i,k)$  représente la valeur  $i$  en binaire sur  $k$  bits ;  $e : b(i,k)$  est la concaténation de la chaîne binaire  $e$  à  $b(i,k)$ . La fonction  $S$  retourne un couple formé par une séquence d'instructions  $s_i$  et un état  $e_i$ . Nous analyserons cette fonction un peu plus loin. Cette liste est une « ligne » de l'automate ; elle contient les instructions à appliquer quand l'automate est dans l'état  $e$ . Durant le décodage, si l'état est  $e$  et que le bloc de  $k$  bits lu du programme vaut  $i$ , il faudra alors appliquer la séquence d'instructions  $s_i$  et placer le décodeur dans l'état  $e_i$  où  $(s_i, e_i) = L_i^e$ .

L'ensemble  $E^e$  contient simplement les états  $e_i$  de  $L^e$ . Ainsi, l'ensemble  $E'$  contient tous les états construits ainsi que les états qui seront construits par l'appel courant de *AutomateR*. L'ensemble des nouveaux états et leur ligne de l'automate est cumulé dans l'ensemble  $Ls$ . Les appels récursifs *AutomateR*( $A, C, k, Iz, e_i, E'$ ) s'appliquent sur tous les nouveaux états  $e_i$  n'ayant pas été déjà construits, soit sur l'ensemble  $E^e - E$ . Pour ces appels, l'argument  $E'$  permet d'indiquer les nouveaux états en cours de construction. Clairement, cela évite une récursion infinie dans le cas où le même état est généré par la fonction  $S$ .

L'objet retourné est un ensemble de couples  $(e_j, L_{e_j})$  où chaque liste ordonnée  $L_{e_j}$  représente les séquences d'instructions à exécuter et les nouveaux états à entreprendre pour chaque état courant  $e_j$ .

La fonction  $S$  identifie la séquence d'instructions reconnues par le bloc de décodage  $b$ . Elle produit un couple  $(s, e)$  où  $s$  est l'une des formes générales présentées à la figure 3.2, et  $e$  est une chaîne de bits représentant le nouvel état du décodeur après le décodage de  $s$ . Pour la suite, les numéros entre parenthèses réfèrent aux numéros de ligne. L'essentiel est une série (15) de descentes dans l'arbre  $A$  conduites par les bits du bloc  $b$ . Si une feuille est

---

2. La notation  $[ \dots ]$  est un constructeur de listes.

atteinte (16), un code opérationnel complet a été décodé, donc une instruction reconnue. Le code opérationnel est  $c$ , le mnémotique  $m$  et le format  $f$  sont déterminés par association à  $c$  selon l'ensemble  $C$  (17). Les arguments  $a_i$  sont les valeurs des séquences de bits se trouvant dans  $b'$  selon le découpage du format  $f$ . Cet ensemble d'arguments peut être vide, être complet selon  $f$ , ou avoir une partie seulement des paramètres de  $m$ , et contenir un argument partiel  $a_p$ . Une fois ces arguments extraits, les bits restants du bloc  $b'$  forment les prochains bits à fouiller. Si le bloc est terminé par  $Iz$  (18), la séquence contient une instruction ayant comme dernier argument le bloc de bits restant, et l'état suivant est forcé à nul. Dans le cas contraire (19), on poursuit la fouille du bloc restant  $b'$  à partir de la racine de l'arbre  $A$ . Par conséquent, le processus de descente dans l'arbre peut se répéter pour une autre instruction. Si ce n'est pas une feuille (20), il y a continuation de la descente dans l'arbre vers la gauche ou la droite selon le prochain bit du bloc  $b'$  (21). Si le bloc  $b'$  est vide sans atteindre une feuille (22), les instructions reconnues forment une séquence  $s$  et l'état suivant est la chaîne de bits  $c$  obtenue par le parcours de la racine au noeud intermédiaire actuel  $\nu$ .

Le prédicat  $Iz$  détermine si un mnémotique oblige le décodeur à se remettre à l'état initial. C'est une fonction dont le domaine est l'ensemble des mnémotiques et le co-domaine les valeurs booléennes. Si  $Iz(m) = \text{vrai}$ , le mnémotique  $m$  exige du décodeur de ne pas exécuter d'instructions virtuelles immédiatement après  $m$  sans revenir à l'état initial. Par conséquent, si  $m$  est dans une instruction virtuelle combinée, elle ne peut être que la dernière. Dans la plupart des cas, de tels mnémotiques changent le flot d'exécution. On ne peut alors connaître statiquement les prochains bits, car cela dépend de la cible de branchement qui dépend du programme. En fait, cette fonction peut déterminer un mnémotique quelconque à forcer le décodeur à se remettre à l'état initial. C'est nécessaire pour certains mnémotiques et on ne peut, en règle générale, s'en passer. Toutefois, il est possible de choisir un sur-ensemble quelconque de mnémotiques pour lesquels le prédicat est vrai. Cela peut permettre de réduire l'espace occupé par l'automate. Nous reviendrons sur ce sujet à la section 3.3.5. C'est au concepteur de la machine virtuelle à fournir un tel prédicat. À titre d'exemple, pour *Machina*, les mnémotiques de base `ret`, `jsr`, `jmp`, `bf` et `br` doivent remettre le décodeur à l'état initial<sup>3</sup>. Une macro se terminant par l'une d'elle, est aussi dans cette catégorie.

---

3. L'instruction `bf` pourrait ne pas forcer le retour à l'état initial. Cependant, sa présence à l'intérieur d'une instruction compliquerait le calcul de l'adresse de branchement. Notez que, de toute façon, l'apparition d'une telle instruction au milieu d'une instruction combinée serait exceptionnelle : l'instruction n'est pas très fréquente dans les programmes, son code opérationnel est donc relativement long, probablement plus de quatre bits, la présence d'un argument de plus de deux bits apparaît très probable, ainsi l'instruction aurait plus de huit bits. Il faudrait une racine décodeur de plus de neuf bits, un cas rarissime étant donné l'espace nécessaire pour générer au moins une instruction à la suite de `bf`.

Les formes générales des séquences produites par la construction du décodeur automate semblent, à priori, assez complexes. Il faut toutefois noter que leur « complexité » est bornée par la longueur maximum du nombre de bits du bloc de décodage. La proposition suivante place une borne supérieure sur la longueur du bloc de décodage, et ainsi sur la complexité des séquences d'instructions.

**Proposition 3.1**

Soient  $l_{\max}$  la longueur maximum des codes opérationnels et  $k$  le nombre de bits de la racine du décodeur. La chaîne représentant un état a au plus  $l_{\max} - 1$  bits. Ainsi, il y a au maximum  $l_{\max} - 1 + k$  bits dans le bloc de décodage.

**Démonstration:** Une chaîne de bits représentant un état est construite en utilisant les bits restants après une suite d'instructions reconnues par un bloc de décodage. De plus, cette chaîne ne devient restante qu'après une descente partielle dans l'arbre des codes. Une telle descente partielle est toujours causée par une chaîne restante trop courte pour atteindre une feuille de l'arbre, et non à cause d'un manque de branche; car autrement l'arbre ne serait pas complet. Ainsi, la chaîne restante ne peut être aussi longue que le plus long code opérationnel, c'est-à-dire  $l_{\max}$  bits. ■

À titre d'exemple, pour un ensemble de codes opérationnels dont le code le plus long a douze bits et dont l'automate décodeur construit a  $k = 7$  bits de décodage, les séquences d'instructions de l'automate ne peuvent être formées par plus de dix-huit bits. Si l'instruction la plus courte, incluant les paramètres, a cinq bits, les séquences ont au plus trois instructions complètes.

### 3.3.2 Espace mémoire de l'automate

Cette section présente quelques résultats sur l'espace occupé par un automate décodeur. Les sections 3.3.3 et 3.3.4 élaborent des méthodes pour réduire cet espace.

**Proposition 3.2**

Soit un code de Huffman de  $n$  symboles. Le nombre d'états de l'automate construit par l'algorithme de la figure 3.3 est au maximum  $n - 1$ . De plus, chaque état a  $2^k$  entrées non nulles.

**Démonstration:** Une codification de  $n$  codes est construite par un arbre binaire complet de  $n$  feuilles, il y a donc  $n - 1$  noeuds internes, c'est-à-dire qu'il y a exactement  $n - 1$  préfixes débutant à la racine. Lors de la construction de l'automate, un état est un préfixe. Ainsi, il ne peut y avoir plus de  $n - 1$  états. Chaque état a  $2^k$  entrées puisque la construction de l'automate suppose toutes les séquences de bit possible. ■

Longueurs 2, 4	00 01 10 1101 1110 1111
Longueurs 2, 4, 6	00 01 10 1100 1101 1110 111101 111110 111111

FIG. 3.4 – Codes opérationnels ayant des automates compacts

Ainsi pour un ensemble de 200 codes opérationnels, un décodeur de  $k = 5$  bits aurait potentiellement  $200 \times 2^5 = 6400$  entrées. Pour une machine hôte dont  $w = 32$ , ce serait, pour l'implantation compacte du décodeur,  $2 \times 6400 + 200 \times 4 = 13600$  octets. C'est un espace substantiel. Toutefois, c'est le pire cas et il existe des situations diminuant le nombre d'états.

Par exemple, pour les six codes opérationnels de longueurs deux et quatre de la figure 3.4, un décodeur dont  $k = 4$  n'a que deux états : sa représentation courte a  $2 \times 2^4 + 6 \times 4 = 56$  octets. Notez que le décodeur dont  $k = 3$  a cinq états, il n'y a donc pas de diminution d'espace car  $5 \times 2^3 + 6 \times 4 = 64$  octets. Les neuf codes de longueurs deux, quatre et six ont aussi des automates dont le nombre d'états est inférieure à  $n - 1$ . Ce sont les automates  $k = 2i$ ,  $i \geq 1$ , ayant tous trois états ; quant aux automates  $k = 2j + 1$ ,  $j \geq 0$ , ils ont tous huit états.

### 3.3.3 Réduction de la taille de l'automate

Pour réduire la taille du décodeur, la première approche consiste à réduire le nombre de bits de décodage à la racine. Cette approche a le désavantage d'augmenter la longueur moyenne de décodage.

Pour y remédier, nous proposons de combiner les avantages du code canonique et le décodage par automate. La technique de base utilisée est le calcul de l'indice du code lorsque sa longueur est connue.

Cette approche peut réduire substantiellement l'espace occupé par l'automate tout en gardant un temps de décodage très efficace. C'est une technique inédite de la construction d'un décodeur. Voici l'essentiel de la méthode de construction d'un tel automate.

Lors de la construction de l'automate, si tous les codes correspondant à l'état et au bloc de  $k$  bits ont la même longueur, le code à exécuter sera le calcul de l'indice selon 3.1 ou 3.2 ; l'état résiduel sera alors nul<sup>4</sup>. Cela permettra potentiellement de réduire le nombre d'états finaux.

Le nombre d'états éliminés dépend de la répartition des codes. En se référant à l'arbre

4. L'automate va vers un état nul car si des bits suivant fournissent une information additionnelle, celle-ci ne peut être que pour confirmer complètement un code qui serait alors déterminé.

binaire des codes canoniques, c'est le nombre de noeuds intérieurs qui sont racines d'un arbre binaire parfait<sup>5</sup>. Tous les noeuds intérieurs correspondent à des chaînes préfixes de codes opérationnels. Ainsi, tous ces préfixes n'engendreront pas d'états dans la table de l'automate. Dans le cas d'un code de Huffman, ce nombre est toujours au moins 1 ; mais ce minimum n'est atteint que dans la situation d'un arbre dégénéré.

À la figure 2.1, représentant les codes opérationnels de Zipf-20, on peut voir que pour ces codes il y a treize noeuds intérieurs racines d'un arbre parfait. Ainsi, pour Zipf-20, seulement six états demeurent dans l'automate. C'est moins du tiers des états de l'automate non réduit.

Cette façon de construire l'automate est une forme intermédiaire entre l'automate « standard » et le décodeur canonique de la section 3.4. Nous pouvons d'ailleurs percevoir qu'il est possible de générer différents types de décodeurs, allant de l'automate fini non réduit au décodeur canonique compact.

### 3.3.4 Contraintes sur les codes opérationnels pour réduire la taille des automates

Une autre méthode pour réduire l'espace mémoire de l'automate consiste en l'usage de codes opérationnels dont les longueurs sont des multiples d'une constante  $k > 1$ . En général, forcer un ensemble de codes à utiliser certaines longueurs résulte en une codification dont la longueur moyenne est plus élevée. Toutefois, cette perte peut être compensée par le gain de la compacité de l'automate. Cette section analyse ces gains et ces pertes, et la méthode pour générer de tels codes.

#### *Algorithme pour générer des codes de longueurs multiples de $k > 1$*

L'algorithme de Huffman peut être adapté pour construire des codes de longueurs multiples de  $k > 1$  [Dub99]. En effet, au départ l'ensemble des  $n$  mnémoniques est complété par un maximum de  $2^k - 1$  mnémoniques ayant une probabilité 0 pour que leur nombre total  $n'$  soit tel que  $n' \bmod (2^k - 1) = 1$ . Ce nombre de mnémoniques à ajouter est  $(2^k - n \bmod (2^k - 1)) \bmod (2^k - 1)$ .

Durant l'algorithme de construction de l'arbre, il suffit de combiner simultanément  $2^k$  symboles. Les symboles de probabilités 0 feront nécessairement partie du groupe des codes les plus longs. Ainsi, une fois l'arbre construit, ils peuvent être éliminés.

On peut déjà percevoir que la valeur  $k$  devrait être assez petite pour espérer obtenir une longueur moyenne peu élevée par rapport à l'entropie. À titre d'exemple, la figure 3.5 présente les longueurs moyennes de Zipf-200 pour les multiples de 1 à 8. Naturellement, le cas  $k = 1$  est sans contrainte et, pour  $k = 8$ , la longueur moyenne est 8, car il y a moins

---

5. Un arbre binaire parfait est complet et toutes ses feuilles sont à la même profondeur.

$k$	Longueurs moyennes
1	6.0267
2	6.0974
3	6.2121
4	6.4375
5	6.7213
6	7.2063
7	7.5391
8	8.0000

FIG. 3.5 –. Longueurs moyennes de Zipf-200 avec contraintes

de 256 codes opérationnels. À titre de comparaison, l'entropie de Zipf-200 est 5.9857. La contrainte d'une longueur multiple de deux semble raisonnable, mais déjà pour un multiple de trois un saut important se produit.

### 3.3.5 Réduction de la taille d'un automate par le prédicat $Iz$

Rappelons que le prédicat  $Iz$ , utilisé dans l'algorithme de construction d'un automate, détermine si un mnémonique  $m$  exige la remise à l'état initial après son émulation. Cela est nécessaire pour plusieurs mnémoniques dont les branchements inconditionnels. Une autre application de ce prédicat est la diminution de l'espace de l'interprète. En effet, l'un des avantages de la remise à l'état initial est la diminution du nombre de séquences d'instructions. Dans un cas extrême, où  $\forall m, Iz(m) = \text{vrai}$ , l'automate n'aurait que des séquences d'instructions formées d'une seule instruction virtuelle. Nous aurions une situation similaire au cas canonique avec la différence que, pour l'automate, plusieurs instructions virtuelles peuvent exister, selon les arguments. Toutefois, cela ralentit l'interprète car moins d'instructions multiples sont décodées.

En pratique c'est une méthode simple où le concepteur peut facilement intervenir. Nous utilisons cette méthode dans quelques benchmarks de la JVM et Machina où  $k > 7$ .

### 3.3.6 Génération du code C d'un décodeur automate

Nous traitons ici des particularités de la génération du code C pour un décodeur automate. D'autres détails importants, dont l'accès au programme en mémoire centrale, l'extraction des paramètres, l'implantation des branchements et des macro-instructions sont traités aux sections 3.5, 3.6, 3.7 et 3.8.

Plusieurs variations sont possibles pour l'implantation de l'automate. On peut rechercher une implantation compacte au détriment de la vitesse, ou une implantation moins compacte afin d'augmenter la vitesse de décodage.

#### *Représentation des tables de transition de l'automate*

Les premières variations peuvent se faire sur les vecteurs d'états. Les entrées des états sont composées de deux champs : l'un indique le prochain état, et l'autre l'instruction virtuelle à exécuter<sup>6</sup>. On peut envisager deux approches pour représenter ces champs : indice dans un vecteur intermédiaire ou adresse directe. Dans la majorité des situations, l'indice pourra être stocké sur un octet, ce qui donnerait deux octets par entrée dans la table de l'automate. Dans le cas d'une adresse, elle sera souvent quatre fois plus longue. La différence entre les deux représentations est suffisamment importante pour y porter une attention particulière. Ainsi nous définissons quatre formats compacts de représentation de l'automate :  $F_{ii}$ ,  $F_{ai}$ ,  $F_{ia}$  et  $F_{aa}$  où le premier indice s'applique au champ instruction, le deuxième au champ état ; la lettre  $a$  pour 'adresse', la lettre  $i$  pour 'indice'. Nous appelons « format compact » le cas  $F_{ii}$ , et le « format long » le cas  $F_{aa}$ .

#### *Implantation des instructions combinées*

Rappelons que l'automate peut reconnaître plusieurs instructions par cycle de décodage. Ainsi, il peut y avoir un branchement à une instruction formée de plusieurs instructions de base, c'est-à-dire à une instruction combinée. Nous utilisons deux méthodes différentes pour générer le code C de telles instructions.

La première méthode est simplement la concaténation des implantations des instructions de base. Comme le démontrent les essais expérimentaux, c'est une méthode offrant, en rapidité d'exécution, de bonnes performances. Nous l'appelons « implantation par concaténation ». Son désavantage apparaît quand chaque instruction de base utilise beaucoup d'espace mémoire sur la machine hôte. La figure 3.6 donne un exemple de deux instructions combinées implantées par concaténation. Un terme `CODE_Imj` est une macro-instruction C pour l'implantation de l'instruction de base  $m_{ij}$ .

#### *La sauvegarde du pointeur de programme*

La méthode d'implantation du décodeur automate oblige les instructions, voulant sauvegarder l'adresse de l'instruction suivante, à calculer cette adresse à partir des registres pc

---

6. Rappelons que cette instruction virtuelle peut être composée de plusieurs instructions virtuelles de base.

```

L_decode_c:
    if(nb_rd < k){
        tmp =((unsigned int)(prgm[pc]))<<16
            | ((unsigned int)(prgm[pc+1]))<<8
            | (unsigned int)(prgm[pc+2]);
        rd |= tmp << (w-24-nb_rd);
        pc += 3;
        nb_rd += 24;
    }
    crd = rd > w-k;
    rd <<= k;
    nb_rd -= k;
    adr_inst = ietat[crd].adr_inst;
    ietat = ietat[crd].adr_prch_état;
    goto *adr_inst;

Imi1: CODE_Imi1; goto L_decode_c;
Imi2: CODE_Imi2; goto L_decode_c;
Imi3: CODE_Imi3; goto L_decode_c;
...
Imi1mi2:
    CODE_Imi1; CODE_Imi2; goto L_decode_c;

Imi3mi4mi5:
    CODE_Imi3; CODE_Imi4; CODE_Imi5; goto L_decode_c;

```

FIG. 3.6 -. Instructions combinées par *concaténation* pour un décodeur automate

et `nb_rd`. En effet, le registre `rd` peut contenir plusieurs octets ; le `pc` pointant ainsi plus loin que l'octet suivant l'instruction d'appel.

Pour effectuer un ajustement au `pc`, l'instruction d'appel a besoin d'une information essentielle : le nombre de bits du bloc de décodage n'ayant pas servi à la reconnaissance de son propre code opérationnel. Cette information devrait être stockée dans le décodeur automate pour permettre la génération du code `C` effectuant ce calcul.

### 3.4 Les décodeurs canoniques

Le décodage canonique nécessite une codification canonique de Huffman [SK64]. Celle-ci a une propriété essentielle qu'on peut définir comme suit.

#### Définition 3.4.1

*Une codification canonique de Huffman  $C = \{c_i\}$  des symboles  $S = \{s_i\}$  est une codification de Huffman telle que les codes  $c_i$  de même longueur sont numériquement consécutifs.*

C'est une construction de Huffman standard, mais la phase d'attribution des codes s'effectue en parcourant l'arbre en largeur : les feuilles de profondeur  $p$  reçoivent bien un code de longueur  $p$  comme dans le codage de Huffman, mais ces codes sont attribués consécutivement, selon leur valeur numérique. La codification canonique génère un ensemble de codes ayant la même performance en gain d'espace que la codification standard de Huffman.

En fait, il est possible de générer deux codes canoniques différents, il suffit d'inverser logiquement les bits d'un code canonique pour en former un autre. Nous distinguerons ces deux codes, puisque la génération du décodeur en dépend.

Voici une définition préliminaire, utile pour la suite.

#### Définition 3.4.2

*Soient  $b$  un code binaire,  $l_b$  sa longueur et un entier quelconque  $w \geq l_b$ . On note par  $V^w(b)$  la valeur  $v(b)2^{w-l_b}$  où  $v(b)$  est la valeur positive de  $b$ .*

En d'autres mots,  $V^w(b)$  est la valeur de  $b$  décalé à l'extrême gauche sur  $w$  bits. À titre d'exemples :  $V^4(11) = 12$ ,  $V^8(010) = 64$ . Cette façon de traiter les chaînes de bits provient de l'algorithme de décodage : l'entrée des bits provenant du programme sera toujours « cadrée à gauche ».

#### Définition 3.4.3

*Soit  $C = \{c_i\}$  un code canonique de Huffman en ordre croissant des longueurs des codes et  $l_{max}$  leur longueur maximum. Il s'agit du code **canonique croissant** ssi  $l(c_i) < l(c_j)$  implique  $V^{l_{max}}(c_i) < V^{l_{max}}(c_j)$  ; dans le cas inverse, c'est le code **canonique décroissant**.*

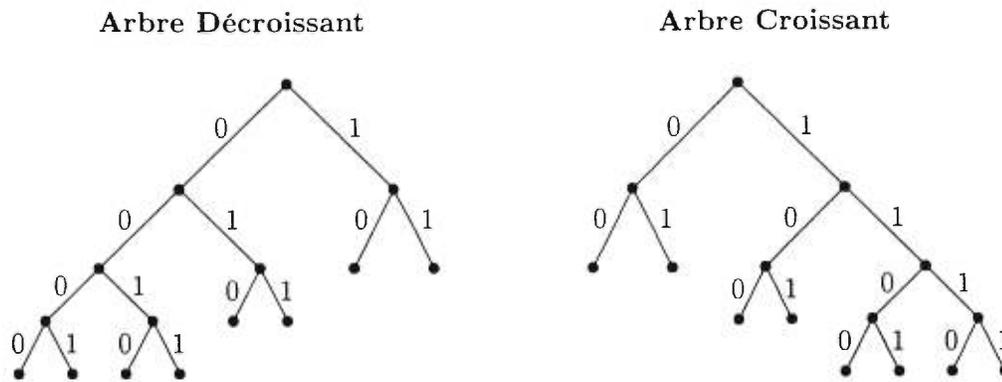


FIG. 3.7 – Arbres canoniques croissant et décroissant

mnémoniques	storeg	push	pushg	sub	add	pushl	storel	pushi
probabilités	0.06	0.07	0.08	0.08	0.12	0.16	0.19	0.24
code croissant	1111	1110	1101	1100	101	100	01	00
code décroissant	0000	0001	0010	0011	010	011	10	11

TAB. 3.1 – Exemple de mnémoniques et de leur code canonique

La figure 3.7 présente un arbre canonique croissant et un arbre canonique décroissant. L'arbre canonique a ses feuilles « tassées » le plus à droite ou le plus à gauche.

Une définition équivalente de canonique croissant serait la condition  $i < j \rightarrow base[i] < base[j]$  (voir la section 3.4.4 pour les définitions de *base* et *depl*). Remarquez que pour le code canonique croissant, l'un des codes les plus longs est formé de '1' seulement, et l'un des codes les plus courts est formé de '0' seulement ; c'est l'inverse dans le cas décroissant. Quand il ne sera pas nécessaire de faire une distinction entre ces deux cas, le terme « canonique » sera simplement utilisé.

À titre d'exemple, soient les mnémoniques et leur probabilité d'occurrence tels que présentés par la table 3.1. Le code canonique croissant de Huffman est découpé en trois séquences :  $\langle 1100, 1101, 1110, 1111 \rangle$ ,  $\langle 100, 101 \rangle$  et  $\langle 00, 01 \rangle$ . Chacune de ces séquences est formée par des nombres binaires ayant des valeurs numériques consécutives. De même pour le code décroissant. Les codes se lisent de la gauche vers la droite et pour interpréter leur valeur numérique, le bit de poids le plus fort est à l'extrême gauche.

L'intérêt majeur du code canonique est la possibilité de représenter, de façon très compacte, la relation bijective entre les codes et leur symbole. Ce que nous analysons à la section suivante.

### 3.4.1 Les décodeurs de codes canoniques

Nous nous intéressons à des techniques de décodage de codes canoniques de Huffman utilisant très peu de mémoire. Le type de décodeurs, traités dans cette section, exige que les codes opérationnels soient des codes canoniques de Huffman. Nous les appelons *décodeurs canoniques*.

Deux techniques de décodage de codes canoniques de Huffman sont proposées par [MT97]. Nous croyons que ce sont les meilleures méthodes publiées jusqu'à maintenant ; ce qui est confirmé par les expériences des auteurs. La première fait un traitement bit par bit des codes. La seconde utilise une première phase utilisant un bloc de  $k$  bits, pour passer ensuite à un traitement bit par bit.

Nous généralisons ces techniques pour permettre aux phases subséquentes d'utiliser plus d'un bit à la fois. En fait, nous avons mis au point un algorithme de construction d'un décodeur de codes canoniques, où le concepteur de la machine virtuelle peut spécifier une borne sur l'espace à utiliser. Ce paramètre permet de construire un décodeur rapide tout en limitant son utilisation d'espace mémoire. Essentiellement, le décodeur construit est un arbre de décodage où chaque noeud interne utilise un certain nombre de bits de décodage. C'est donc une généralisation de l'automate dans sa forme ; mais c'est une restriction dans la quantité d'instructions exécutées à chaque cycle de décodage.

Nous abordons les techniques de décodage de codes canoniques et l'algorithme de construction de leurs décodeurs dans les sections suivantes.

### 3.4.2 La génération de la structure d'un décodeur canonique

La propriété numérique des codes canoniques de Huffman permet un stockage compact de l'arbre de décodage. Les vecteurs *base* et *deplacement*, définis comme suit, permettent de représenter la relation bijective code/symbole.

#### Définition 3.4.4

Soient des codes opérationnels  $C = \{c_i\}$ ,  $l_{\max}$  la longueur maximum de ces codes et  $w$  une constante quelconque telle que  $w \geq l_{\max}$ . Le vecteur  $base^w(C)[1 \dots l_{\max}]$  est tel que  $base^w(C)[i]$  est la plus petite valeur  $V^w(c)$  des codes de longueur  $i$  ; similairement le vecteur  $base(C)[1 \dots l_{\max}]$  est la plus petite valeur  $v(c)$  des codes de longueur  $i$ . Le vecteur  $depl(C)[1 \dots l_{\max}]$  est tel que  $depl(C)[i]$  est le nombre de codes de longueur inférieure à  $i$ . L'indice du code  $c$  de longueur  $i$  est alors calculé par l'une ou l'autre des expressions suivantes.

$$\frac{V^w(c) - base^w(C)[i]}{2^{w-i}} + depl(C)[i] \quad (3.1)$$

$$v(c) - base(C)[i] + depl(C)[i] \quad (3.2)$$

Là où l'ensemble  $C$  est clairement sous-entendu, nous utiliserons simplement les notations  $base^w[i]$ ,  $base[i]$  et  $depl[i]$ . L'utilité de définir deux vecteurs similaires comme « base » provient de deux nécessités algorithmiques différentes : si on connaît la longueur du code,  $base$  est utile pour calculer son indice (voir noeud de type 2 dans le décodeur général) ; dans un autre cas, que nous traitons dans les prochains paragraphes, la longueur est inconnue et il faut cadrer à gauche la valeur des  $w$  prochains bits. Tous les codes apparaissent donc multipliés par une puissance de 2 ; c'est-à-dire par  $2^{w-i}$  où  $i$  est la longueur du code. Il faut donc, dans ce cas, employer les valeurs décalées des bases pour faire des comparaisons numériques.

En supposant les  $w$  prochains bits du programme à décoder dans une variable  $rd$ , il suffit de balayer séquentiellement le vecteur  $base^w$  pour connaître la longueur du prochain code : car  $base^w[i] \leq v(rd) < base^w[i+1]$  implique que le prochain code est de longueur  $i$ <sup>7</sup>. L'espace nécessaire se limiterait aux vecteurs  $base^w$  et  $depl$ <sup>8</sup>.

La figure 3.8 expose l'essentiel d'un décodeur très compact de codes opérationnels canoniques. C'est d'ailleurs la version la plus compacte pouvant être générée par l'algorithme de construction du décodeur présenté à la figure 3.9. La variable  $rd$ , de  $w$  bits, contient les prochains bits du programme à décoder ; ceux-ci y sont cadrés à gauche (dans la partie supérieure de  $rd$ ). Le code canonique croissant est utilisé, et l'itération de la ligne 2 s'exécute à partir de la fin du vecteur  $base$ . Après cette itération,  $i$  contient la longueur du code cadré à gauche dans  $rd$ . La ligne 3 implante simplement l'expression 3.1, la ligne 4 élimine le code opérationnel dans  $rd$ , et la ligne 5 branche à l'implantation de l'instruction virtuelle. Il va sans dire que le vecteur  $adr$  contient les adresses de ces implantations.

Il serait possible de faire une recherche à partir du début du vecteur  $base$ , ou encore d'employer le code canonique décroissant. Ces variations n'ont pas tous la même vitesse d'exécution, car cela dépend des fréquences dynamiques des instructions virtuelles ; en général, celles-ci ne sont pas proportionnelles à la longueur de leur code opérationnel. Il est donc possible de faire un choix judicieux si on connaît les fréquences dynamiques des codes.

On peut facilement voir que le temps d'exécution est, dans le pire cas, dépendant du nombre d'itérations de la ligne 2 ; c'est la longueur maximum du code canonique moins sa longueur minimum. Le tout est lent en comparaison à la méthode de l'automate fini. Il est, bien entendu, possible de faire mieux en utilisant une méthode similaire de l'automate fini : utiliser  $k$  bits afin d'indicer un vecteur donnant la prochaine étape de décodage.

---

7. On suppose que  $C$  est un code canonique croissant. On peut aussi faire une recherche dichotomique, mais puisque  $base^w$  est court (moins de 10 éléments dans la plupart des cas), ce ne sera pas nécessairement plus rapide.

8. Naturellement, il faut toujours ajouter l'espace du vecteur qui associe l'indice du code à l'adresse des instructions de la machine hôte implantant l'instruction virtuelle.

```

1  i = lmax;
2  while(rd < base_w[i]) i--;
3  crd = (rd-base_w[i]>> w-i)+depl[i];
4  rd <<= i;
5  goto *adr[crd];

```

FIG. 3.8 –. Décodeur très compact pour un code canonique croissant

L'idée générale est de reconnaître l'indice du code, ou sa longueur, en utilisant  $k$  bits de l'entrée. L'usage de  $k$  bits donne lieu à trois situations possibles.

- Cas 1 :** L'indice du code opérationnel est directement reconnu et le calcul 3.1 n'est pas nécessaire pour le déterminer.
- Cas 2 :** La longueur est directement reconnue, bien que l'indice du code opérationnel soit inconnu. Le calcul 3.1 doit être effectué pour le connaître.
- Cas 3 :** La longueur du code opérationnel n'est pas connue, il faut continuer à décoder d'autres bits de l'entrée pour la déterminer.

Le premier cas est le plus avantageux. Il est possible de brancher directement au code pour émuler l'instruction virtuelle. Le deuxième cas permet d'aller à l'instruction virtuelle après le calcul de 3.1 et un accès à un vecteur d'adresses. Le troisième cas est le moins avantageux. Il faut continuer le décodage. C'est donc une sous-étape de décodage, c'est-à-dire un sous-arbre du décodeur. Les sous-étapes de décodage peuvent utiliser différentes valeurs pour  $k$ . Ces séquences de décodage forment un arbre dont chaque noeud correspond à l'un de ces trois cas ; les noeuds de l'arbre sont donc de trois types différents. Un noeud interne est de type 3 et les feuilles sont de types 1 ou 2 (la racine est toujours de type 3).

#### Définition 3.4.5

Nous noterons  $k_\nu$  le nombre de bits de l'entrée utilisé par le noeud  $\nu$  ; l'arité du noeud est alors  $2^{k_\nu}$ . Le type d'un noeud  $\nu$  sera noté  $\bar{\nu}$ .

Nous nous intéressons à un algorithme de génération d'un décodeur rapide étant donné un ensemble de codes canoniques de Huffman. Comment devrait-on générer ce décodeur, c'est-à-dire, quels sont les paramètres à utiliser pour contrôler sa génération ? L'espace mémoire et le temps de décodage sont deux préoccupations majeures.

L'espace mémoire peut être évalué en cumulant l'espace utilisé par les vecteurs de chaque noeud. Il faut aussi tenir compte de l'espace occupé par le code du programme implantant le décodeur.

Pour faire cette évaluation nous utilisons les constantes suivantes. L'espace mentionné se réfère à la machine hôte.

$e_0$  est le nombre d'octets occupé par les instructions pour le code de la figure 3.8.

$e_1$  est le nombre d'octets d'une adresse (e.g. 4).

$e_2$  est le nombre d'octets occupé par les instructions pour un noeud de type 2.

$e_3$  est le nombre d'octets occupé par les instructions pour un noeud de type 3.

Dans la plupart des cas,  $e_1$  devrait se situer entre 2 et 8. Les valeurs de  $e_2$  et  $e_3$  devraient être approximativement égales. Par exemple, il est peu probable que  $e_3$  soit le double de  $e_2$ .

Voici la fonction d'évaluation de l'espace occupé par l'arbre  $A$  d'un décodeur.

$$E(A) = \sum_{\nu \in A} e(\nu) \text{ où } e(\nu) = \begin{cases} e_0 + 2e_1 l_{\max} & \text{si } \bar{\nu} = 0 \\ 0 & \text{si } \bar{\nu} = 1 \\ e_2 & \text{si } \bar{\nu} = 2 \\ 2^{k_\nu} e_1 + e_3 & \text{si } \bar{\nu} = 3 \end{cases} \quad (3.3)$$

Notez que cette fonction ne tient pas compte de l'espace mémoire nécessaire pour certaines parties du décodeur qui ne changent pas selon la structure retenue. Ces parties sont fixes et ne peuvent influencer le choix de la structure. Par exemple, le code nécessaire pour charger les octets de la mémoire au registre `rd` et le code de l'interprète.

Pour évaluer le temps de décodage, nous utilisons un vecteur de probabilités d'occurrences dynamiques  $D = \{d_c\}$  des codes opérationnels  $C = \{c\}$ . Ce n'est pas une évaluation du temps réel, mais bien une évaluation approximative tenant compte de facteurs pertinents. Ce vecteur attribue à chaque code opérationnel  $c$  la probabilité  $d_c$  d'être décodé lors d'une exécution typique d'un programme. C'est donc un facteur d'occurrence dynamique plutôt que statique utilisé lors de la construction des codes opérationnels. Le temps d'exécution d'un noeud de type  $i$  est dénoté par  $t_i$ . Ce temps doit être évalué pour la machine hôte choisie. En fait, nous devrions avoir  $t_1 = 0$ , et  $t_2$  devrait avoir sensiblement la même valeur que  $t_3$ . Toutefois, cela dépend de la machine hôte. Nous utiliserons  $t_0$  pour dénoter le temps d'exécution d'une itération de l'instruction 2 de la figure 3.8, et  $T_0(D, C, t_0)$  pour le temps total.

Le temps pondéré de décodage, noté  $T(A)$ , pour un décodeur  $A$  sur les codes  $C$  est donné par l'expression suivante.

$$T(A) = \sum_{c \in C} d_c \sum_{\nu \in P_c} t_{\bar{\nu}} \text{ où } P_c \text{ est l'unique chemin dans } A \text{ de la racine à } c \quad (3.4)$$

En d'autres mots, ce temps est la somme pondérée des temps d'exécution de chaque noeud du chemin de décodage pour chaque code opérationnel.

Est-ce qu'une recherche exhaustive parmi tous les décodeurs possibles est réalisable en un temps raisonnable? Cela est envisageable si une borne sur l'espace mémoire est fournie. Cette borne permettra de limiter la recherche.

Les définitions suivantes identifient des opérations à effectuer pour construire un décodeur. Elles sont utilisées par l'algorithme de construction du décodeur.

#### Définition 3.4.6

Soit  $C$  un ensemble de codes opérationnels et un entier  $k$ ,  $k \geq 1$ . L'ensemble des codes de  $C$  reconnaissables sans ambiguïté par leur  $k$  premiers bits est noté  $R^k$ . L'ensemble des codes de  $C - R^k$  dont la longueur est reconnue par leur  $k$  premiers bits est noté  $L^k$  et l'ensemble des longueurs distinctes de  $L^k$  est noté  $L^{k'}$ . L'ensemble  $C - R^k - L^k$  est noté  $P^k$  et l'ensemble des préfixes distinctes de  $k$  bits de  $P^k$  est noté  $P^{k'}$ .

La construction de ces ensembles peut se faire en temps linéaire par rapport au nombre de codes de  $C$ , si cet ensemble est trié selon l'ordre canonique.

Notez que  $R^k$  est l'ensemble des codes opérationnels de longueur au plus  $k$ . Pour un ensemble de codes à préfixe unique, non nécessairement Huffman,  $R^k$  n'a pas cette propriété, mais dans le cas des codes de Huffman, un code ayant une longueur de plus de  $k$  bits ne peut être le seul code ayant un certain préfixe de  $k$  bits. Sinon des bits suffixes d'un tel code peuvent être éliminés, sans perte d'ambiguïté, diminuant la longueur moyenne. Ainsi, un code ne peut être reconnu sans ambiguïté par l'un de ses préfixes.  $R^k$  peut être construit par simple parcours linéaire : un code ayant au plus  $k$  bits fait partie de  $R^k$ , et aucun autre code ne peut en faire partie. L'ensemble  $L^k$  peut aussi se construire par un parcours linéaire de  $C$  en supposant les codes en ordre canonique<sup>9</sup>.

L'algorithme 3.2 effectue une recherche exhaustive pour déterminer la structure d'un décodeur  $A$  le plus rapide possible, selon la fonction  $T(A)$ , en utilisant une borne d'espace mémoire sur  $E(A)$ . Le décodeur créé ne dépasse pas la borne et il n'existe pas d'autres décodeurs plus rapide selon  $T(A)$ .

Pour la fonction *DecodeurOptimum*, nous avons inclu seulement les paramètres modifiés lors des appels récurifs. Les autres paramètres importants sont  $D$  et  $l_{\max}$ . Le paramètre  $D$  est une fonction donnant la somme des probabilités dynamiques de codes opérationnels. Le paramètre  $l_{\max}$  est la longueur maximum des codes.

Le paramètre  $Cs$  contient la liste des noeuds de type 3 non encore traités. Chaque élément de cette liste est un couple  $(p,C)$  où  $p$  est un préfixe de tous les codes  $C$ . La notation  $Cs = (p,C) : Cr$  représente la tête  $(p,C)$  et le reste  $Cr$  de la liste  $Cs$ .

---

9. L'ordre canonique est tel que les codes sont en ordre croissant de leur longueur et, pour un ensemble de codes de même longueur, dans l'ordre de leur valeur numérique.

**Algorithme 3.2 (Construction de décodeurs canoniques)**

**Entrée :** ensemble non-singleton  $C = \{c\}$  de codes canoniques de Huffman;  
fonction  $D : \{c\} \mapsto \mathbb{R}$  définie par  $\sum d_c$  où les  $d_c$  sont des probabilités dynamiques;  
paramètres du processeur hôte  $e_i, t_i, 0 \leq i \leq 3$ ;  
et borne supérieure  $B$ .

**Sortie :** un arbre décodeur  $A$  pour  $C$ , où  $E(A) \leq B$  et  $T(A)$  est minimum.

```

1  Soit  $(\text{existe}, b, t, L, A) = \text{DecodeurOptimum}((\lambda, C) : (), B, 0, \infty, (), 1)$ 
2  Si  $\text{existe}$  Alors Le décodeur est  $A$ , de temps  $t + t_3$  et espace  $b$ 
3      Sinon L'espace  $B$  est insuffisant pour décoder  $C$ 

4   $\text{DecodeurOptimum}(Cs, b, t, t_{\text{mg}}, L, f_{\text{base}})$ 
5  Soient  $Cs = (p, C) : Cr; l_{\text{max}} = \max\{l(c_i) - l_p | c_i \in C\}; k_{\text{max}} = \min(l_{\text{max}}, \lceil \lg' b/e_1 \rceil)$ 
6   $R_{\text{min}} \leftarrow (\text{faux}, 0, 0, (), ())$ 
7   $t_{\text{min}} \leftarrow t_{\text{mg}}$ 
8  Pour  $k$  de  $k_{\text{max}}$  à 1 Faire
9      Début
10     Soient  $t' = t + D(P^k)t_3 + D(L^k)t_2; b' = b - 2^k e_1 + |L^{k'} - L|e_2 + |P^{kP}|e_3$ 
11     Si  $t' < t_{\text{min}}$  et  $b' \geq 0$  Alors
12         Soit  $L' = L \cup L^{k'}$ 
13         Si  $Cr = ()$  et  $P^k = \emptyset$ 
14             Alors  $t_{\text{min}} \leftarrow t'; R_{\text{min}} \leftarrow (\text{vrai}, b', t', L', ((k, p, R^k, P^k, L^k, P^{kP}, L^{k'}, ())))$ 
15             Sinon Soit  $Cs' = \text{ajoute}(P^k, P^{kP}, Cr)$ 
16                 Soit  $(\text{existe}, b'', t'', L', A) = \text{decodeurOptimum}(Cs', b', t', t_{\text{min}}, L', f_{\text{base}})$ 
17                 Si  $\text{existe}$  Alors  $t_{\text{min}} \leftarrow t'';$ 
18                      $R_{\text{min}} \leftarrow (\text{vrai}, b'', t'', L', (k, p, R^k, P^k, L^k, P^{kP}, L^{k'}, A_{1..|P^{kP}|}) : A_{|P^{kP}|..})$ 
19         Fin
20     Soient  $t' = t + T_0(D, C, t_0); b' = b - f_{\text{base}}(e_0 + l_{\text{max}}e_1)$ 
21     Si  $b' < 0$  ou  $t' \geq t_{\text{min}}$  Alors retourner  $R_{\text{min}}$ 
22     Si  $Cr = ()$  Alors retourner  $(\text{vrai}, b', t', L, (0, p, C) : ())$ 
23     Soit  $(\text{existe}, b'', t'', L', A) = \text{DecodeurOptimum}(Cr, b', t', t_{\text{min}}, L, 0)$ 
24     Si  $\text{existe}$  Alors retourner  $(\text{vrai}, b'', t'', L', (0, p, C) : A)$ 
25     Sinon retourner  $R_{\text{min}}$ 

```

FIG. 3.9 -. Algorithme de construction des décodeurs canoniques

Le paramètre  $b$  est l'espace disponible, en octets, pour la construction du décodeur. Le paramètre  $t$  est le cumul des temps de décodage des codes jusqu'ici décodés, et une partie des temps des codes à décoder. Cette partie correspond aux temps des noeuds de type 3 pour tous les codes se trouvant encore dans  $Cs$ .

Le paramètre  $t_{\text{mg}}$  est le temps minimum d'un arbre de décodage ayant déjà été entièrement conçu. Ce paramètre permet d'arrêter la conception d'un arbre quand le temps cumulé est supérieur à cette borne.

Le paramètre  $L$  est une liste de toutes les longueurs des noeuds de type 2 ayant déjà été conçus. Son usage permet d'évaluer correctement l'espace occupé par les noeuds de type 2, car il évite de cumuler leur espace plus d'une fois.

Le paramètre  $f_{\text{base}}$  vaut 1 ou 0. Implicitement, il spécifie l'existence (0) ou la non existence (1) du vecteur **base** (ou sa version **base<sub>w</sub>**). Durant la conception de l'arbre, si un noeud créé utilise une recherche séquentielle ( $k = 0$ ), le vecteur **base** devient existant. La continuation de la conception de ce même arbre peut alors supposer son existence. Ce paramètre permet ainsi d'évaluer correctement l'espace occupé par les prochains noeuds utilisant une recherche séquentielle.

Le travail principal de l'algorithme est de déterminer, pour chaque noeud  $\nu$ , le nombre de bits  $k_\nu$  d'entrée à utiliser. Ce nombre peut être différent pour chaque noeud de l'arbre de décodage. La valeur maximum pour  $k_\nu$  est bornée supérieurement par la borne globale  $b$  et la longueur du préfixe déjà décodé au noeud  $\nu$ . Naturellement, sa valeur minimum est 1. Toutefois, dans l'algorithme de construction du décodeur, le cas  $k_\nu = 0$  a un sens. Il s'agit d'un noeud à partir duquel est appliqué l'algorithme de recherche séquentiel de la longueur des codes opérationnels (voir figure 3.8).

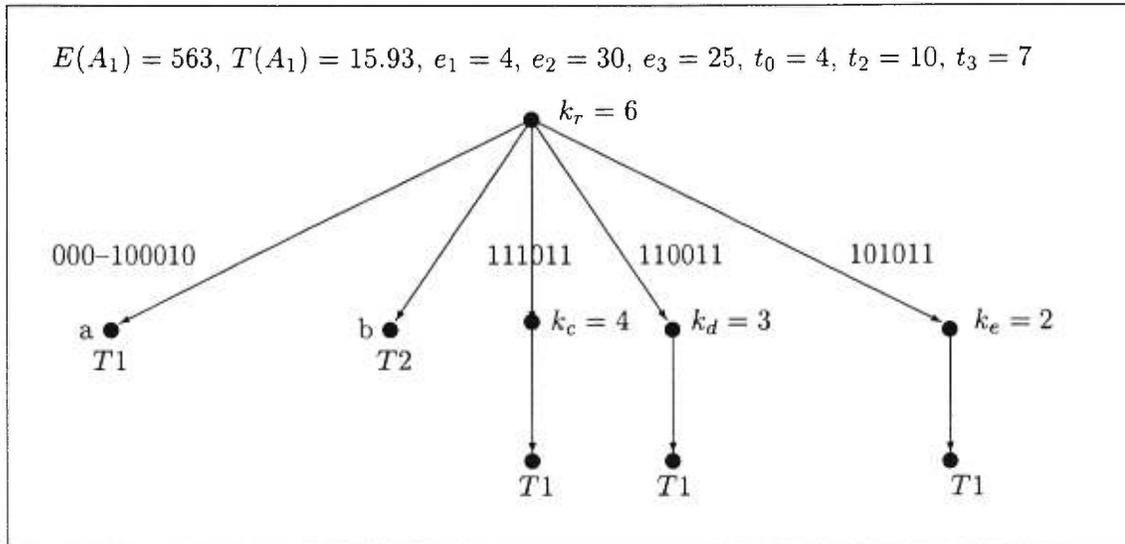
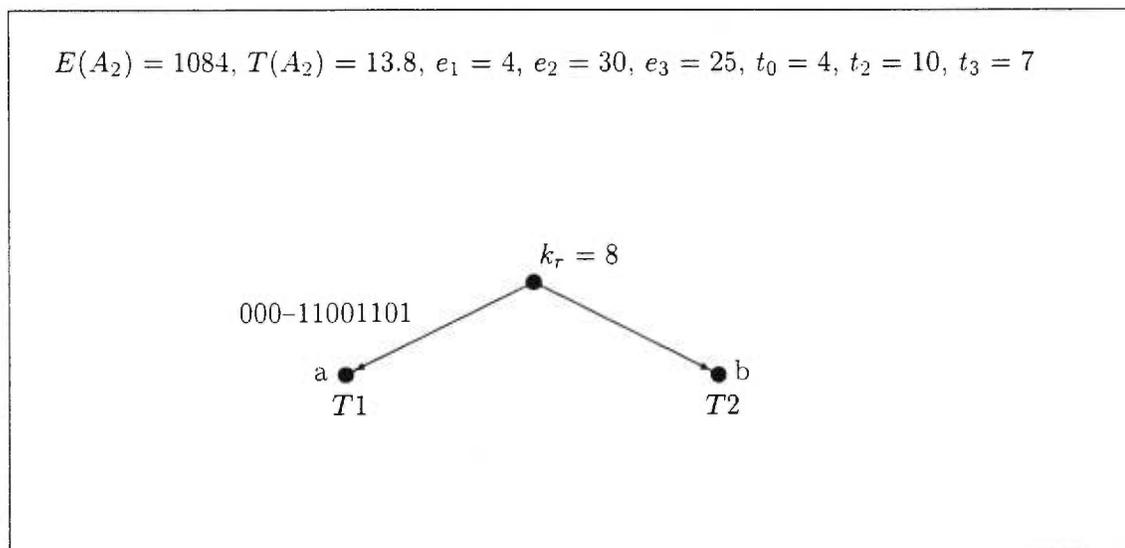
L'algorithme procède de façon récursive sur les noeuds de type 3. À chaque appel récursif, la valeur  $b$  est révisée pour n'inclure que l'espace restant disponible.

En pratique cet algorithme est très rapide. Il a été traduit en Scheme et son temps d'exécution, en mode interprété, est moins de quinze secondes pour des ensembles de plus de deux cent quarante codes opérationnels.

La table 3.2 présente les parties essentielles des codes opérationnels obtenus et leur décodage par deux arbres décodeurs  $A_1$  et  $A_2$ , présentés à la figure 3.10. Dans la table 3.2, les lettres réfèrent aux noeuds où le décodage final des codes opérationnels s'effectue. Les probabilités dynamiques sont égales. Le temps moyen pour  $A_1$  est de 15.93 et pour  $A_2$  de 13.8; mais  $A_2$  utilise presque deux fois plus d'espace que  $A_1$ .

<i>i</i> -Zipf	Code opérationnel	Arbre $A_1$		Arbre $A_2$	
		Noeud final	Temps	Noeud final	Temps
1	000	r	7	r	7
2	0010	r	7	r	7
3	0011	r	7	r	7
4	0100	r	7	r	7
5	01010	r	7	r	7
6	01011	r	7	r	7
7	01100	r	7	r	7
8	01101	r	7	r	7
9	011100	r	7	r	7
10	011101	r	7	r	7
11	011110	r	7	r	7
12	011111	r	7	r	7
13	100000	r	7	r	7
14	100001	r	7	r	7
15	100010	r	7	r	7
16	1000110	b	17	r	7
17	1000111	b	17	r	7
...	...				
31	1010101	b	17	r	7
32	1010110	e	14	r	7
33	10101110	e	14	r	7
34	10101111	e	14	r	7
35	10110000	b	17	r	7
...	...				
62	11001011	b	17	r	7
63	11001100	d	14	r	7
64	11001101	d	14	r	7
65	110011100	d	14	b	17
66	110011101	d	14	b	17
68	110011111	d	14	b	17
69	110100000	b	17	b	17
...	...				
124	111010111	b	17	b	17
125	111011000	c	14	b	17
126	111011001	c	14	b	17
...	...				
135	1110111110	c	14	b	17
136	1110111111	c	14	b	17
137	1111000000	b	17	b	17
138	1111000001	b	17	b	17
...	...				
199	1111111110	b	17	b	17
200	1111111111	b	17	b	17

TAB. 3.2 -. Codes Zipf-200 et les temps de décodage pour deux décodeurs

Arbre  $A_1$ Arbre  $A_2$ FIG. 3.10 -. Deux arbres décodeurs  $A_1$  et  $A_2$  pour Zipf-200

```

L_decode:
    { charge rd d'au moins  $c_{\max}$  bits }
b1      crd = rd » ( $w - k_r$ );
b2      goto *adr_[crd];
L_i :   /* codes opérationnels de longueur  $i$  (type 2) */
i1      crd = rd » ( $w - i$ );
i2      goto *adr_inst[crd -  $base(C^{t2})_i$  +  $depl(C^{t2})_i$ ];
Lp_prefixe: /* sous-décodeur (type 3) */
p1      crd = rd » ( $w - l_{\text{prefixe}} - k_{\text{prefixe}}$ );
p2      goto *adr_prefixe[crd -  $v(\text{prefixe})2^{k_{\text{prefixe}}}$ ];
Imne:   /* code C de l'instruction  $mne$  (type 1) */
        Si  $mne$  a des paramètres, les copier dans les variables  $p_i$ 
        { code pour l'exécution de l'instruction  $mne$  }
        rd <<=  $l_{\text{opcode}} + l_{\text{parm}}$ ; /* élimine codeop et paramètres */
        nb_rd -=  $l_{\text{opcode}} + l_{\text{parm}}$ ;
        goto L_decode;

```

FIG. 3.11 –. Structure générale, en C, d'un décodeur de codes canoniques

### 3.4.3 Génération du code C d'un décodeur canonique

Dans cette section, nous traitons des particularités de la génération du code C pour un décodeur canonique. D'autres détails importants dont l'accès au programme en mémoire centrale, l'extraction des paramètres, l'implantation des branchements et des macro-instructions sont traités aux sections 3.5, 3.6, 3.7 et 3.8 respectivement.

À partir de la structure créée par l'algorithme 3.2, il est possible de générer plusieurs implantations différentes, car celles-ci ne spécifient aucun détail sur la façon d'extraire les paramètres, de transférer les données de la mémoire aux variables, et en général sur le traitement des bits.

Nous allons donc analyser plusieurs façons d'implanter la structure d'un décodeur. Puisque la vitesse de décodage dépend de cette implantation, il est important d'y apporter un soin adéquat. Toutefois, tous les décodeurs seront implantés selon une forme générale. Nous étudierons cette forme et, par la suite, nous présenterons les différentes particularités d'implantation.

La figure 3.11 présente la structure générale du code C d'un décodeur d'instructions

virtuelles. En voici une description détaillée.

Une étiquette  $L_i$  existe s'il y a des codes opérationnels de longueur  $i$  qui ne sont pas directement reconnus par un noeud de type 3. L'ensemble de tous ces codes est noté  $C^{t2}$ . Une étiquette  $Lp\_prefixe$  existe s'il y a des codes opérationnels ayant ce préfixe et dont la longueur n'est pas reconnue directement par les noeuds de type 3. Il y a une étiquette  $Imne$  pour chaque instruction virtuelle. C'est le point d'entrée de leur implantation.

Le cycle de décodage débute à l'étiquette  $L\_decode$ . La première étape consiste à charger la variable  $rd$  d'au moins  $l_{max}$  bits. La variable  $rd$ , pour « registre décodeur », a une largeur de  $w$  bits et contient toujours, à l'exécution de  $b1$ , au moins  $l_{max}$  bits dans sa partie haute. La variable  $nb\_rd$  contient le nombre de bits de  $rd$  considérés valides<sup>10</sup>.

Les instructions  $b1$  et  $b2$  effectuent la première étape de décodage sur un bloc de  $k$  bits ; c'est la racine du décodeur. Le décalage logique  $b1$  produit la valeur numérique des  $k$  bits. Le branchement  $b2$  va soit à une instruction virtuelle à exécuter (cas 1), soit à une étiquette  $L_i$  (cas 2), ou soit à une sous-étape du décodeur (cas 3). L'expression  $w - k_r$  est une constante connue à la génération du code  $C$ . Le contenu du vecteur  $adr\_$  ayant  $2^k$  éléments, est constant et a été initialisé au chargement de la machine virtuelle. Tous les vecteurs  $adr$  sont initialisés de cette façon.

Les instructions  $i1$  et  $i2$  traitent les cas où la longueur du code opérationnel est connue sans en connaître directement l'indice. Dans ce cas, il suffit de brancher à l'adresse de l'instruction dont l'indice est donné par l'expression 3.1. Les expressions  $w - i$  et  $base(C^{t2})_i + depl(C^{t2})_i$  sont des constantes connues à la génération du code  $C$ . Les accès aux vecteurs  $base$  et  $depl$  sont donc éliminés. (Il serait ainsi possible de faire une variation du code où il y aurait un seul noeud  $L_x$  pour plusieurs longueurs. Ce serait un petit gain d'espace, mais aussi une perte de vitesse, car il faudrait accéder aux vecteurs  $base$  et  $depl$ . Le gain est, en effet, assez petit car bien qu'il y ait une diminution du nombre d'instructions de la machine hôte, il y aurait un espace additionnel pour représenter explicitement les vecteurs  $base$  et  $depl$ .)

Les instructions  $p1$  et  $p2$  effectuent un sous décodage dans le contexte d'un préfixe donné. Ces instructions effectuent un branchement à une adresse déterminée par la valeur de nouveaux bits de l'entrée. La valeur de la constante  $k_{prefixe}$  est déterminée par l'algorithme de création du décodeur ; il s'agit du nombre de bits de décodage utilisé par ce noeud. La valeur de la constante  $l_{prefixe}$  est la longueur du préfixe. En d'autres mots,  $l_{prefixe} + k_{prefixe}$  est le nombre bits décodés incluant ce noeud. Le terme  $v(prefixe)2^{l_{prefixe}}$  est la valeur numérique du préfixe décalée de  $l_{prefixe}$  bits à gauche. Ce terme est soustrait de  $crd$  car ces

10. Il peut y avoir plus de bits valides dans  $rd$ , mais l'objectif d'en indiquer moins est utile pour maintenir le pointeur de programme ( $pc$ ) aligné sur une frontière d'octet.

bits n'ont pas été éliminés par les noeuds supérieurs de l'arbre de décodage. En fait, elle peut être réduite à la valeur de `crd` en faisant un adressage au vecteur `adr_prefixe` avec un déplacement négatif de la valeur  $v(\text{prefixe})2^{\text{prefixe}}$  ; celle-ci étant une constante.

Pour simplifier l'implantation, nous supposons que la longueur maximale d'un code opérationnel est de  $w$  bits. En fait, cette restriction s'applique au cas où l'un des noeuds de l'arbre de décodage utilise la recherche séquentielle (cas 0 bit). Par implication, il n'y a aucun noeud décodant sur plus de  $w$  bits. C'est une très faible restriction. En effet, en supposant  $w = 32$ , le cas le plus fréquent, il est très peu probable qu'un code opérationnel ait plus de 32 bits<sup>11</sup>. Si nécessaire, cette restriction peut être satisfaite en utilisant des modifications à l'algorithme de construction des codes de Huffman [LH90, MTK95]<sup>12</sup>. Nous n'avons pas implanté cet algorithme, étant donné la rareté de son emploi.

### 3.5 L'accès efficace au programme en mémoire centrale

Une perte d'efficacité importante est l'accès à la mémoire centrale. Si aucune attention n'est apportée à cette difficulté, le temps d'exécution de tout programme compressé peut s'avérer être très lent. Il est important d'y apporter un soin adéquat.

Nous utilisons une codification des programmes en mémoire centrale sous la forme « big-endian ». À priori, cela pose un problème pour les processeurs « little-endian », mais en fait, on ne peut supposer, pour les processeurs « big-endian », la possibilité de pouvoir lire un mot de  $w$  bits à une adresse quelconque. Ainsi, tous les accès mémoires doivent être faits octet par octet pour les deux types de processeurs<sup>13</sup>.

Au début du cycle de décodage, il y a chargement d'un minimum de bits dans `rd` pour permettre un fonctionnement correct du décodeur. Ce nombre varie d'un décodeur à l'autre et, surtout, du type de décodeur.

Une première méthode consiste à charger  $w$  bits de la mémoire centrale et d'ajouter une partie de ces bits au registre `rd`. Pour un mot de 32 bits, ce chargement peut se faire, en langage C, de la façon suivante (`prgm` est un vecteur d'octets).

```
tmp = ((unsigned int)(prgm[pc]))<<24 | ((unsigned int)(prgm[pc+1]))<<16
      | ((unsigned int)(prgm[pc+2]))<<8 | (unsigned int)(prgm[pc+3]);
```

11. Il est possible d'obtenir un code opérationnel de plus de 32 bits avec seulement 34 mnémoniques, mais ayant des fréquences très particulières. Par exemple, avec les fréquences  $2^i, 0 \leq i \leq 33$ , les codes de Huffman sont de la forme  $0_1 \dots 0_i 1$  pour  $0 \leq i \leq 32$  et de la forme  $0_1 \dots 0_{33}$  pour  $i = 33$ .

12. Il va sans dire qu'il y a alors une perte de compression par rapport au codage de Huffman sans limite de longueur.

13. Certains processeurs « big-endian » peuvent permettre l'accès à des mots non alignés. Toutefois, ces accès sont souvent plus lents. De toute façon, nous désirons une solution plus portable.

```

1.  movzbl (%edi,%ecx),%edx      ; Octet prgm[pc]
2.  sall   $24,%edx             ;
3.  movzbl 1(%edi,%ecx),%eax    ; Octet prgm[pc+1]
4.  sall   $16,%eax            ;
5.  orl    %eax,%edx           ;
6.  movzbl 2(%edi,%ecx),%eax    ; Octet prgm[pc+2]
7.  sall   $8,%eax             ;
8.  orl    %eax,%edx           ;
9.  movzbl 3(%edi,%ecx),%eax    ; Octet prgm[pc+3]
10. orl    %eax,%edx           ; Quatre octets dans edx

```

FIG. 3.12 –. Code assembleur du Pentium pour le chargement de quatre octets

La figure 3.12 contient une traduction en assembleur de ce code, produit par le compilateur gcc avec option -O3 pour le Pentium.

Le terme `(%edi,%ecx)` est une traduction de `prgm[pc]`, le registre `edx` joue le rôle de la variable `tmp`, le registre `eax` sert aux calculs intermédiaires des décalages, l'instruction `movzbl` charge un octet avec mise à zéro de la partie haute, et l'instruction `sall` est le décalage à gauche de 32 bits<sup>14</sup>.

On peut rapidement percevoir que ce code, exécuté à chaque début de cycle imposera une perte de vitesse importante. Effectivement, des benchmarks simples montrent que c'est une solution trop lente.

À la section suivante nous présentons trois autres techniques et nous montrerons, à la section 3.9 sur des benchmarks, que deux d'entre elles ont des performances adéquates.

### 3.5.1 Trois formes d'accès au programme

Pour améliorer l'accès au programme à interpréter, c'est-à-dire au code compressé dans la mémoire centrale, nous analysons trois techniques portables. Ces trois techniques tentent de minimiser le nombre d'accès à la mémoire. Nous les distinguons par leur forme du code C. Des tests sur benchmarks, faits à la section 3.9, démontrent la pertinence de ces techniques.

#### *Forme simple de base : forme-a.*

C'est une amélioration en tenant compte du nombre de bits déjà dans le registre `rd`. Le nombre d'octets accédés en mémoire est exactement le nombre d'octets ajoutés dans `rd`.

14. Pour le Pentium, il existe une façon beaucoup plus rapide d'effectuer ce chargement. En effet, l'instruction `bswap` permet de le faire de la façon suivante: `movl prgm(%ebx),%edx; bswap %edx` où `bswap` effectue une transformation de « big-endian » à « little-endian ». Mais c'est clairement une solution non portable.

Cette méthode a été implantée par une instruction `switch`. Elle s'avèrera non performante.

*Forme longue d'accès au programme : forme-b*

Pour cette forme, nous avons l'invariance qu'à l'entrée du décodeur, il y a entre  $w - 7$  et  $w$  bits dans `rd`. En supposant  $w = 32$ , il y a quatre racines,  $0 \leq x < 4$ , pour le décodeur : une racine charge  $x$  ou  $x + 1$  octets dans `rd`, selon la valeur de `nb_rd`.

Puisque chaque instruction virtuelle connaît le nombre de bits extraits du registre `rd`, il est possible de connaître approximativement le nombre d'octets à charger. En effet, supposons qu'une instruction virtuelle utilise  $b \leq w - 7$  bits, puisqu'à l'entrée de son implantation il y a entre  $w$  et  $w - 7$  bits dans `rd`, il y a, après son exécution, entre  $w - b$  et  $w - b - 7$  bits restant dans `rd`. Ainsi, il y a entre  $\lceil (b - 1)/8 \rceil$  et  $1 + \lceil (b - 1)/8 \rceil$  octets à charger dans `rd`. Si  $b$  est une constante, ce qui est le cas le plus fréquent en pratique, il est possible de brancher à l'une de ces racines sans aucun test. Sur des tests expérimentaux, cette solution s'avère être nettement plus efficace que la forme-a.

La figure 3.13 présente un exemple de code C pour l'accès forme-b avec un décodeur automate dont  $k = 3$ . Cet exemple a deux racines. L'étiquette `L_decode_1_inter` est utilisée par le décodeur lors d'un décodage où aucune instruction virtuelle n'est reconnue. L'étiquette `L_decode_sans_chargement` est utilisée par les instructions ayant rechargé `rd` et ne nécessitant pas le test de la ligne 29. Ces instructions changent le flot d'exécution et rechargent `rd` en conséquence.

La forme-b est souvent moins performante que la forme-c, présentée plus bas. Cependant, nous gardons cette forme disponible car elle utilise moins d'espace, en terme de code C, que la forme-c.

*Forme conditionnelle d'accès au programme : forme-c*

Pour cette forme, que nous appelons « forme-c », il y a toujours, à la racine du décodeur, une vérification du nombre de bits dans `rd`. Si ce nombre est en-deça d'un certain seuil, il y a alors chargement d'au moins un octet. Ce seuil devrait être, au minimum, le nombre de bits utilisés par la racine. Toutefois, il est plus simple d'utiliser la longueur maximum des codes opérationnels. De cette façon, aucune étape du décodeur n'a à vérifier le nombre de bits dans `rd`. D'ailleurs, des tests expérimentaux nous ont convaincus d'utiliser ce seuil pour ne pas diminuer la vitesse de décodage par rapport à une dispersion des vérifications dans les sous-décodeurs.

L'avantage de cette méthode, pour le temps d'exécution, est l'élimination du chargement de peu d'octets. Par exemple, dans le cas où la longueur maximum des codes opérationnels est 12 et  $w = 32$ , s'il y a accès au programme par la racine du décodeur, il y aura au moins

```
1. L_decode_2:
2.   if(nb_rd > w-16){
3.     tmp      = (unsigned int)PRGM(pc);
4.     rd       |= tmp << (w-8 -nb_rd);
5.     pc       += 1;
6.     nb_rd    += 8;
7.     crd      = rd > (w-3);
8.     adr_goto = ietat[crd].adr_inst;
9.     ietat    = ietat[crd].adr_prch_etat;
10.    goto     *adr_goto;
11.  }
12.  tmp      = ((unsigned int)PRGM(pc)<<8) |
13.            (unsigned int)PRGM(pc+1);
14.  rd       |= tmp << (w-16 -nb_rd);
15.  pc       += 2;
16.  nb_rd    += 16;
17.  crd      = rd > (w-3);
18.  adr_goto = ietat[crd].adr_inst;
19.  ietat    = ietat[crd].adr_prch_etat;
20.  goto     *adr_goto;
21.
22. L_decode_1_inter:
23.   rd <= 3;
24.   nb_rd -= 3;
25.
26. L_decode_1:
27.
28.   if(nb_rd > w-8){
29.     crd      = rd > (w-3);
30.     adr_goto = ietat[crd].adr_inst;
31.     ietat    = ietat[crd].adr_prch_etat;
32.     goto     *adr_goto;
33.   }
34.   tmp      = (unsigned int)PRGM(pc);
35.   rd       |= tmp << (w-8 -nb_rd);
36.   pc       += 1;
37.   nb_rd    += 8;
38.
39. L_decode_sans_chargement:
40.   crd      = rd > (w-3);
41.   adr_goto = ietat[crd].adr_inst;
42.   ietat    = ietat[crd].adr_prch_etat;
43.   goto     *adr_goto;
```

FIG. 3.13 –. Exemple d'accès mémoire forme-b accompagné d'un décodeur automate

deux octets de chargés. Cela diminue le nombre de transferts, et de manipulation de bits, de la mémoire au registre `rd`.

Le désavantage est que nous ne pouvons plus supposer, au début des implantations des instructions virtuelles, qu'il y a entre  $w - 7$  et  $w$  bits dans `rd`. Pour les instructions virtuelles ayant des paramètres, il est alors potentiellement nécessaire de vérifier le nombre de bits dans `rd` avant de procéder. Ce n'est pas nécessaire pour toutes les instructions ayant au total moins de  $l_{\max}$  bits. Ce désavantage se traduit par une augmentation du code de l'interprète. Pour une machine de plus de deux cents instructions, comme pour la JVM, ce code peut être substantiel pour les décodeurs automatés. C'est pourquoi la forme-b est utile pour générer de plus petits interprètes.

La figure 3.14 présente un exemple de code C pour l'accès mémoire forme-c. Cet exemple est tiré du décodeur canonique pour la JVM dont la racine a 10 bits. Il s'agit de la racine du décodeur sans aucune instruction virtuelle, c'est-à-dire sans l'interprète. L'exécution d'une méthode JVM débute essentiellement à l'étiquette `L_decode_vide_rd`. Les registres de décodage `rd` et `nb_rd` sont mis à zéro et il y a chargement de quatre octets (ici  $w = 32$ ).

La plupart des implantations des instructions virtuelles branchent, après l'émulation, à `L_decode_c`. S'il y a moins de douze bits dans `rd`, deux octets lui sont ajoutés. Le décodage se fait alors à partir de la ligne 29.

### 3.6 Extraction des arguments

Les arguments d'une instruction sont codés à la suite du code opérationnel. Puisque ces arguments ont des longueurs non multiples de huit, ils doivent être extraits par des opérations sur les bits.

Le processus doit être rapide, il faut donc simplifier les traitements au minimum. La méthode employée est simple. Une fois le code opérationnel retiré de `rd`, les arguments sont retirés les uns après les autres pour les placer dans des variables  $p_i$ . Cela peut se faire par un décalage à droite de  $w - l$  bits où  $l$  est la longueur de l'argument. Ce décalage peut être arithmétique ou logique selon le type de l'argument.

Il peut être nécessaire de charger de nouveaux octets de la mémoire. C'est ici que se produit une différence notable de la taille de l'interprète entre la forme-c et la forme-b. Dans le cas de la forme-c, le nombre de bits assurés présents dans `rd` est dans la plupart des cas assez faible. C'est au maximum  $l_{\max} - l_c$  où  $l_c$  est la longueur du code opérationnel et  $l_{\max}$  la longueur du code opérationnel le plus long. Si la longueur du premier argument est supérieure à  $l_{\max} - l_c$ , il faut recharger des octets dans `rd` pour les transférer dans  $p_1$ . Mais dans le cas de forme-b, le nombre de bits dans `rd`, après l'élimination du code opérationnel, est au moins  $w - 7 - l_c$ . Ce nombre est en pratique supérieur à  $l_{\max} - l_c$ . Ainsi, dans la

```

1.  L_decode_vide_rd :
2.
3.      nb_rd = 0; rd = 0;
4.
5.  L_decode:
6.      tmp = ((unsigned int)(PRGM(pc))) <24 |
7.            ((unsigned int)(PRGM(pc+1)))<16 |
8.            ((unsigned int)(PRGM(pc+2)))<8 |
9.            (unsigned int)(PRGM(pc+3));
10.     rd = tmp;
11.     pc += w/8;
12.     nb_rd = w;
13.
14.  L_decode_sans_chargement:
15.
16.     crd = rd > (w-10);
17.     adr_goto = adr_[crd];
18.     goto *adr_goto;
19.
20.  L_decode_c: /* Forme c */
21.
22.     if(nb_rd < 12){
23.         tmp = ((unsigned int)PRGM(pc)<8) |
24.              ((unsigned int)PRGM(pc+1));
25.         rd |= tmp < (w-16 - nb_rd);
26.         pc += 2;
27.         nb_rd += 16;
28.     }
29.     crd = rd > (w-10);
30.     adr_goto = adr_[crd];
31.     goto *adr_goto;
32.
33.  L_11:
34.     crd = rd > (w-11);
35.     goto *adr_inst[crd-1956+0];
36.
37.  L_12:
38.     crd = rd > (w-12);
39.     goto *adr_inst[crd-3992+38];
40.
41.  Lp_1111100101:
42.     crd = rd > (w-12);
43.     goto *adr_1111100101[crd-4*997];

```

FIG. 3.14 –. Exemple de l'accès mémoire forme-c accompagné d'un décodeur canonique

forme-b, il y a moins d'instructions virtuelles chargeant des octets ; ce qui aide à produire des interprètes de plus petites tailles.

### 3.7 Génération des instructions de branchement

Les instructions de branchement nécessitent un traitement particulier lors de la génération de leur implantation en C, principalement pour l'automate. Cela tient au fait que, dans le cas de l'automate, les bits décodés font peut-être partie de l'instruction suivante. Ce qui demande un ajustement au calcul de l'adresse de branchement.

L'instruction de branchement spécifie le nombre de bits à franchir à partir de zéro.

En règle générale, la génération du code d'une instruction virtuelle se fonde sur une séquence d'instructions, mais nous supposons que seulement la dernière peut être une instruction de branchement. Cette supposition peut être satisfaite, lors de la construction de l'automate, en utilisant un prédicat adéquat pour  $Iz$ .

En plus des arguments normaux, un mnémonique  $m$  tel que  $Iz(m) = \text{vrai}$  peut former une instruction à implanter avec un dernier argument spécifiant l'état suivant. C'est la longueur de cet état qui est important, bien que celui-ci est spécifié sous la forme d'une chaîne de caractères. Cette chaîne ne peut être confondue avec un argument partiel, car si l'état suivant est non-nul, il n'y a pas d'argument partiel et tous les arguments, si le mnémonique a des paramètres, sont complets.

Supposons qu'il faut générer l'implantation d'une instruction de branchement ayant un paramètre. Pour proprement calculer l'adresse de branchement, il faut tenir compte des bits encore présents dans `rd`. Essentiellement, il faut calculer le nombre de bits à franchir en tenant compte de la position exacte, au bit près, de la fin de l'instruction de branchement dans `rd`. Dans le cas d'un décodeur canonique, une fois l'argument chargé, `nb_rd` contient le nombre exact de bits à franchir encore restant dans `rd`. Dans le cas d'un décodeur automate, il est possible que l'instruction ait un état suivant non-nul. Dans ce cas, il ne faut pas soustraire de `nb_rd` la longueur de l'état, car ces bits font partie de l'instruction suivante et doivent être franchis par le branchement.

Le nombre de bits de l'état est une constante, car pour une même instruction de branchement, nous générons une instruction virtuelle pour chaque longueur d'état.

Dans le cas du calcul de l'adresse de l'instruction suivante, par exemple pour empiler l'adresse de retour pour une instruction d'appel de sous-routines, il faut faire ce même ajustement en tenant compte de l'état non-nul.

### 3.8 Implantation des macro-instructions

Les macro-instructions sont implantées directement dans la machine virtuelle. Leur implantation est similaire aux instructions combinées des automates, mais quelques particularités s'appliquent.

Si la macro a des paramètres, tous ces paramètres sont transférés dans des variables  $p_i$ . Ce transfert se charge d'accéder la mémoire centrale et de maintenir le pointeur de programme. Le code de chaque instruction de base est alors généré séquentiellement, en référant chaque paramètre si nécessaire. Techniquement, chaque macro  $C$  décrivant l'implantation d'une instruction virtuelle de base, a autant de paramètres que l'instruction virtuelle. Ainsi, il est possible de symboliquement référencer chacun des paramètres de la macro-instruction.

Ce processus fonctionne si aucune instruction virtuelle de branchement ne se trouve dans la macro-instruction. Pour *Machina* nous acceptons des instructions de branchement à l'intérieur de la macro. Dans ce cas, il suffit de générer des instructions de branchement conditionnel  $C$  pour implanter correctement ces macro-instructions. Les instructions de branchement conditionnel effectuent directement le flot de contrôle de la macro.

### 3.9 Benchmark des décodeurs

Dans cette section, nous appliquons des benchmarks synthétiques aux décodeurs pour démontrer leur efficacité et déterminer la forme d'accès à la mémoire la plus rapide. Notre conclusion générale est que la forme-c est, très souvent, la plus performante et que les décodeurs créés sont suffisamment performants pour être applicables en pratique.

Les chapitres suivants appliquent les décodeurs à des environnements plus complexes.

#### 3.9.1 Les machines simples utilisées pour les benchmarks

Pour analyser le temps de décodage par rapport aux tâches à accomplir par le reste de la machine virtuelle, nous utilisons trois machines virtuelles ayant des instructions de granularités croissantes. Dans la première, c'est-à-dire la machine 1, chaque instruction virtuelle additionne un à sa propre variable entière. Dans la deuxième, il y a deux opérations supplémentaires sur deux variables entières. Pour la troisième, il y a deux accès additionnels à un vecteur (une pile). Les instructions n'ont pas de paramètres. La figure 3.15 présente l'essentiel des instructions virtuelles des trois machines. Seules les instructions 1 et 20 (`CODE_Im1` et `CODE_Im20`) diffèrent des autres instructions. L'instruction 1 vérifie la condition d'arrêt du programme, donc de la machine virtuelle. L'instruction 20 effectue un branchement à l'instruction 1. Un branchement requiert un traitement particulier pour chaque décodeur, qui est essentiellement la remise à l'état nul du décodeur ; la façon précise d'effectuer cette

Pentium			Sparc		
M <sub>1</sub>	M <sub>2</sub>	M <sub>3</sub>	M <sub>1</sub>	M <sub>2</sub>	M <sub>3</sub>
0.38	0.45	0.81	2.13	2.56	5.08

TAB. 3.3 – Temps absolu du décodeur code-octet de Zipf-20, pour trois machines virtuelles sans paramètres, sur deux processeurs

Pentium			Sparc		
MP <sub>1</sub>	MP <sub>2</sub>	MP <sub>3</sub>	MP <sub>1</sub>	MP <sub>2</sub>	MP <sub>3</sub>
0.40	0.49	0.85	2.32	2.83	3.76

TAB. 3.4 – Temps absolu du décodeur code-octet de Zipf-20, pour trois machines virtuelles avec paramètres, sur deux processeurs

tâche dépend du décodeur et nous la représentons par la macro `ETAT_ZERO`.

Les performances des décodeurs sont comparées à un décodeur de code-octet non compressé. C'est un décodeur rapide indexant un vecteur d'adresses pour choisir l'instruction virtuelle à exécuter. Pour la suite de ce chapitre, nous l'appelons simplement « décodeur code-octet ». La figure 3.16 démontre l'essentiel du code C de ce décodeur.

Pour tous les benchmarks de ce chapitre, nous exécutons le même programme. Il s'agit de l'exécution cyclique des vingt instructions de base. La constante `MAXCOMPTEUR` a la valeur  $4 \times 10^5$ . Ainsi, chaque instruction de base est exécutée  $4 \times 10^5$  fois<sup>15</sup>.

### 3.9.2 Performance des décodeurs canoniques et automates

La table 3.6 détaille les vitesses d'exécution des trois machines virtuelles à instructions sans paramètres, utilisant des décodeurs canoniques, sous les trois formes d'accès mémoire pour deux processeurs. Les temps sont comparés aux temps du décodeur code-octet, dont les temps absolus d'exécutions apparaissent dans la table 3.3 (voir la figure 3.16 pour la forme de ce décodeur).

Le décodeur canonique  $k_r = 4, L_5, L_6$  a été généré avec les paramètres  $e_0 = 50, e_1 = 4, e_2 = 30, e_3 = 25, t_0 = 4, t_2 = 10, t_3 = 7$ . L'espace occupé par le décodeur est de 124 octets et il a trois noeuds; la racine utilise quatre bits, et les deux noeuds de type 2 traitent les codes opérationnels de longueurs cinq et six.

Si on augmente quelque peu l'espace disponible pour représenter le décodeur, il est

15. Une fois additionnelle pour l'instruction `m1`!

---

**Machine virtuelle 1**

```
#define CODE_Im1  compteur1++; if(compteur1 > MAXCOMPTEUR) goto FIN;
#define CODE_Imi  compteuri++;
#define CODE_Im20 compteur20++; ETAT_ZERO; goto L_decode;
```

---

**Machine virtuelle 2**

```
#define CODE_Im1  compteur1++; aux += compteur1; aux <= compteur1;
                 if(compteur1 > MAXCOMPTEUR) goto FIN;
#define CODE_Imi  compteuri++; aux += compteuri; aux <= compteuri;
#define CODE_Im20 compteur20++; aux += compteur20;
                 aux <= compteur20; ETAT_ZERO; goto L_decode;
```

---

**Machine virtuelle 3**

```
#define CODE_Im1  compteur1++; aux += compteur1; aux <= compteur1;
                 pile[aux & 0xFF] = aux; aux = pile[compteur1 & 0xFF];
                 if(compteur1 > MAXCOMPTEUR) goto FIN;
#define CODE_Imi  compteuri++; aux += compteuri; aux <= compteuri;
                 pile[aux & 0xFF] = aux; aux = pile[compteuri & 0xFF];
#define CODE_Im20 compteur20++; aux += compteur20;
                 aux <= compteur20; pile[aux & 0xFF] = aux;
                 aux = pile[compteur20 & 0xFF]; ETAT_ZERO;
                 goto L_decode;
```

---

**FIG. 3.15 – Instructions virtuelles des machines benchmarks**

```
1. unsigned char prgm[100] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
2.                               13, 14, 15, 16, 17, 18, 19};
3.
4. int main(){
5.     register unsigned int code, pc=0, tmp;
6.
7.     Inst adr[] = {&&Im1,  &&Im2,  &&Im3,  &&Im4,  &&Im5,  &&Im6,  &&Im7,
8.                  &&Im8,  &&Im9,  &&Im10, &&Im11, &&Im12, &&Im13, &&Im14,
9.                  &&Im15, &&Im16, &&Im17, &&Im18, &&Im19, &&Im20};
10.
11.     int compteur1 = 0, compteur2 = 0, compteur3 = 0, compteur4 = 0,
12.         ...
13.         compteur17 = 0, compteur18 = 0, compteur19 = 0, compteur20 = 0;
14.
15. L_decode:
16.
17.     code = prgm[pc];
18.     pc++;
19.     goto *adr[code];
20.
21. Im1:
22.     compteur1++;
23.     if(compteur1 >= MAXCOMPTEUR) goto FIN;
24.     goto L_decode;
25.
26. Im2: compteur2++; goto L_decode;
27.     ...
28. Im19: compteur19++; goto L_decode;
29. Im20: compteur20++; pc = 0; goto L_decode;
30.
31. FIN:
32.     printf("Le compteur1 = %d\n", compteur1);
33.     ...
34.     printf("Le compteur20 = %d\n", compteur20);
35. }
```

FIG. 3.16 -. Un décodeur simple de code-octet non compressé

possible d'augmenter le nombre de bits de la racine à 5 bits. Un tel décodeur canonique n'a qu'un seul noeud de type 2 pour les codes opérationnels de longueur 6. On peut voir que les temps d'exécution ont légèrement diminué par rapport au décodeur précédent. Un résultat cohérent car, pour les six codes opérationnels de longueur cinq, il n'y a plus qu'une étape de décodage.

Les temps de décodage, où tous les codes opérationnels sont reconnus par la racine, nécessitent un décodeur de 256 octets. Il n'y a donc pas d'étapes intermédiaires de décodage. Ainsi, dans la plupart des cas, les temps de décodage diminuent.

La table 3.5 présente les temps d'exécution de cinq automates pour deux processeurs selon les formes a, b et c. L'implantation des instructions combinées s'effectue par concaténation des instructions de base. On peut voir que l'accès mémoire forme-c sort gagnante pour la plupart des automates. Notez les cas des automates de cinq et six bits à la racine pour la machine 3 sur Pentium. Il y a une augmentation de la vitesse d'exécution. Cela se produit sur Sparc pour les automates de trois à six bits. Il faut toutefois noter que le code exécutable de ces automates sur Sparc varie de 28 à 73 Ko. En comparaison, le code exécutable du décodeur code-octet a 25 Ko. Bien entendu, la compression du programme virtuel n'a, ici, aucune influence sur l'espace du code exécutable résultant, car ce programme virtuel a une petite taille. La différence entre la taille du programme compressé (13 octets), et non compressé (20 octets), n'est que de sept octets. De plus, la machine générée n'est pas spécialisée pour le programme mais demeure une machine virtuelle capable d'exécuter un programme quelconque.

Les tables 3.8 et 3.7 présentent les temps d'exécution pour les mêmes décodeurs, processeurs et machines mais dont une partie des instructions virtuelles ont des paramètres. Les granularités des instructions sont les mêmes, mais six instructions virtuelles ont un paramètre. Ces benchmarks démontrent que l'extraction des arguments par des opérations sur les bits n'implique pas une perte majeure de performance. Les temps sont comparables aux machines sans paramètres.

En conclusion, la forme-a d'accès mémoire est à abandonner. Les deux autres formes sont utiles car leur performance est bonne, bien que la forme-c est à préférer pour la vitesse. Là où l'espace de l'interprète est une contrainte majeure, c'est la forme-b qu'il faudrait utiliser.

D'autre part, les décodeurs automates ont un potentiel très intéressant. On peut voir qu'ils peuvent augmenter la vitesse d'interprétation. Toutefois, la présence de paramètres diminue leur capacité à reconnaître plus d'une instruction par cycle, et leur taille est imposante.

Décodeur	Machine <sub>1</sub>			Machine <sub>2</sub>			Machine <sub>3</sub>		
Pentium									
	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>	<b>c</b>
$k_r = 2$	3.74	2.21	2.21	3.48	2.04	2.07	2.09	1.54	1.64
$k_r = 3$	2.71	1.92	1.97	2.66	1.82	1.84	1.98	1.18	1.21
$k_r = 4$	2.00	1.55	1.47	2.08	1.49	1.42	1.51	1.01	1.05
$k_r = 5$	1.68	1.37	1.32	1.66	1.36	1.33	1.43	0.99	0.91
$k_r = 6$	1.42	1.21	1.18	1.48	1.22	1.13	1.33	0.88	0.81
Sparc									
	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>	<b>c</b>
$k_r = 2$	4.32	2.24	2.27	3.60	2.04	2.08	1.97	1.23	1.23
$k_r = 3$	2.92	1.70	1.67	2.55	1.59	1.57	1.52	0.99	0.99
$k_r = 4$	2.33	1.46	1.42	1.99	1.41	1.36	1.17	0.86	0.86
$k_r = 5$	1.88	1.22	1.19	1.69	1.19	1.15	0.99	0.78	0.78
$k_r = 6$	1.73	1.15	1.08	1.50	1.12	1.05	0.89	0.72	0.64

TABLE 3.5 – Temps relatif de décodage de Zipf-20 par des décodeurs automates, selon trois formes d'accès mémoire, pour trois machines virtuelles sans paramètres, sur deux processeurs

Décodeur	Machine <sub>1</sub>			Machine <sub>2</sub>			Machine <sub>3</sub>		
Pentium									
	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>	<b>c</b>
$k_r = 4, L_5, L_6$	1.81	1.76	1.52	2.19	1.64	1.48	1.38	1.07	0.96
$k_r = 5, L_6$	1.60	1.71	1.47	2.13	1.64	1.55	1.32	0.99	0.96
$k_r = 6$	1.60	1.58	1.34	2.08	1.49	1.42	1.31	0.95	0.90
Sparc									
	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>a</b>	<b>b</b>	<b>c</b>
$k_r = 4, L_5, L_6$	2.77	1.61	1.52	2.04	1.60	1.39	1.09	0.99	0.97
$k_r = 5, L_6$	2.77	1.51	1.43	2.50	1.42	1.35	1.02	0.91	0.88
$k_r = 6$	2.39	1.63	1.21	2.12	1.23	1.18	0.93	0.81	0.78

TABLE 3.6 – Temps relatif de décodage de Zipf-20 par trois décodeurs canoniques, pour trois machines virtuelles sans paramètres, sur deux processeurs

Décodeur	MachineP <sub>1</sub>			MachineP <sub>2</sub>			MachineP <sub>3</sub>		
Pentium									
	a	b	c	a	b	c	a	b	c
$k_r = 2$	3.62	2.85	2.57	3.14	2.84	2.32	2.24	1.68	1.72
$k_r = 3$	2.60	1.92	2.09	2.26	1.77	1.93	1.80	1.89	1.18
$k_r = 4$	2.07	1.80	1.72	1.87	1.53	1.57	1.52	1.05	1.20
$k_r = 5$	1.67	1.37	1.37	1.77	1.34	1.26	1.27	0.91	0.98
$k_r = 6$	1.40	1.25	1.15	1.39	1.16	1.06	1.10	0.86	0.84
Sparc									
	a	b	c	a	b	c	a	b	c
$k_r = 2$	4.09	2.19	2.28	3.53	2.01	2.04	2.92	1.83	1.88
$k_r = 3$	2.84	1.72	1.76	2.61	1.66	1.62	2.20	1.46	1.46
$k_r = 4$	1.87	1.46	1.46	2.22	1.41	1.44	1.89	1.38	1.27

TAB. 3.7 -. Temps relatif de décodage de Zipf-20 par des décodeurs automates, selon trois formes d'accès mémoire, pour trois machines virtuelles avec paramètres, sur deux processeurs

Décodeur	MachineP <sub>1</sub>			MachineP <sub>2</sub>			MachineP <sub>3</sub>		
Pentium									
	a	b	c	a	b	c	a	b	c
$k_r = 4, L_5, L_6$	1.80	1.62	1.47	1.57	1.55	1.38	1.37	1.22	1.18
$k_r = 5, L_6$	1.67	1.62	1.44	1.57	1.53	1.40	1.34	1.20	1.14
$k_r = 6$	1.70	1.45	1.30	1.46	1.41	1.26	1.25	1.10	1.14
Sparc									
	a	b	c	a	b	c	a	b	c
$k_r = 4, L_5, L_6$	2.06	1.59	1.55	1.91	1.51	1.44	1.70	1.41	1.35
$k_r = 5, L_6$	2.20	1.42	1.37	1.76	1.37	1.34	1.54	1.30	1.25
$k_r = 6$	1.90	1.29	1.16	1.60	1.27	1.16	1.46	1.19	1.14

TAB. 3.8 -. Temps relatif de décodage de Zipf-20 par trois décodeurs canoniques, selon trois formes d'accès mémoire, pour trois machines virtuelles avec paramètres, sur deux processeurs

## Chapitre 4

### APPLICATION À LA MACHINE JVM

---

*«I have no data yet. It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories instead of theories to suit facts.»*

*Sherlock Holmes au Dr. Watson dans A Scandal in Bohemia  
par Sir Arthur Conan Doyle*

Comment appliquer les techniques précédentes à des machines déjà existantes? Peut-on en tirer profit sur des machines possédant leur propre ensemble d'instructions? C'est ce que nous analysons pour une machine virtuelle conçue pour le langage Java, la JVM (Java Virtual Machine). Notre point de départ est un ensemble d'instructions virtuelles. Nous supposons que le compilateur ne peut être modifié; en d'autres mots, nous avons un code objet sous la forme d'instructions virtuelles. Ce code peut être modifié et tout le format du code objet peut être remodelé pour obtenir une nouvelle implantation de la machine virtuelle.

La première approche spécialise certaines instructions sans ajout de macro-instructions; la seconde ajoute des macro-instructions. Dans les deux cas, il y a compression des codes opérationnels par codage de Huffman.

Nous analysons ces deux approches sur la JVM pour mesurer leur impact sur la vitesse, et le facteur de compression. La seconde approche produit les meilleurs facteurs de compression ainsi que les meilleures vitesses.

#### 4.1 La machine virtuelle JVM de Java

La machine virtuelle Java est une extension de la machine conçue par James Gosling, en 1992, pour exécuter des programmes écrits dans le langage Oak. La spécification complète de cette machine est décrite par Lindholm et Yellin [LY99], et les raisons qui ont motivé certains choix techniques de sa conception sont exposées dans [LY97] par les mêmes auteurs. Le livre de Gosling, Joy et Steele [GJS96] décrit le langage Java.

Pour les mesures du temps d'exécution, nous utilisons l'implantation Harissa[MMBC97]. Dans la section 4.4.1 nous discutons des modifications apportées à cette implantation pour exécuter des programmes compressés.

Les mesures d'économie d'espace sont effectuées indépendamment de la machine virtuelle.

## 4.2 Compression sans macro-instructions

La table 4.1 présente l'espace du code-octet ainsi que la proportion de cet espace occupé par les codes opérationnels et les opérandes pour plusieurs groupes de programmes objets Java. La colonne 'Taille Code-octet' présente les tailles du code exécutable des fichiers '.class' des groupes. Ces tailles ne contiennent pas l'espace occupé par la table des constantes ou autres informations contenues dans les fichiers '.class'. Toutes les tailles sont en octet. La colonne 'Taille Codeops' indique l'espace occupé par les codes opérationnels; on peut donc déduire le taux d'occupation de ces codes et des opérandes. Ces informations permettent d'évaluer les facteurs de compression des codes opérationnels et des opérandes pour les tables suivantes. Elles permettent aussi de réaliser l'importance de l'espace occupé par ces deux parties du code-octet. On peut conclure que, pour le code Java sans macro-instructions, l'espace des codes opérationnels est approximativement égal à l'espace des opérandes.

La table 4.3 présente la fréquence des codes opérationnels pour plusieurs groupes de la librairie JDK 1.1. Les mnémoniques ayant une fréquence inférieure à 100 ne sont pas présentés.

La table 4.2 présente les facteurs de compression de ces groupes en appliquant seulement la codification de Huffman sur les codes opérationnels. Deux facteurs de compression sont présentés, l'un par rapport à la taille du code-octet (incluant les opérandes), et l'autre par rapport aux codes opérationnels seulement. Toutes les tailles sont en octet. Ces valeurs ont été obtenues en utilisant un ensemble de codes opérationnels dont les fréquences proviennent des classes de la librairie JDK 1.1. Ces fréquences proviennent de la table 4.3. Les tailles des codes opérationnels incluent l'espace perdu de quelques bits à chaque méthode. En effet, pour chaque méthode, le dernier octet contient moins de huit bits utiles. En d'autres termes, les valeurs présentées tiennent compte du fait que chaque méthode débute à une frontière d'octet. L'espace total réel occupé par tous les codes opérationnels compressés de tous les groupes est de 93339 octets. Ainsi, il y a  $8 \times (94376 - 93339) = 8296$  bits perdus pour l'alignement des méthodes.

Le groupe LANG présente un bon facteur de compression de 72%. Ce facteur est dû, en majeure partie, au seul fichier `Character.class`, dont la taille de son code-octet est de 63338 octets. Ce fichier contient majoritairement les instructions `sipush`, `bipush`, `dup` et `bastore`. Son facteur de compression est de 67%. En fait, la compression d'un tel fichier sera plus élevée en utilisant un dictionnaire (macro-instructions), car la plupart des instructions forment la même séquence répétitive, soit (`dup`, `sipush`, `bipush`, `bastore`). Le même phénomène se produit avec le groupe TEXT. Il contient un fichier, `DecompositionIterator.class`, dont la taille de 40150 octets est de 49% la taille totale du groupe. Le code-octet de cette classe est, en grande partie, formé par cette même séquence d'instructions.

Groupe	Taille Code-octet	Taille Codeops	Facteur Codeops	Taille Opérandes
AWT	44413	22622	51%	21791
IO	27176	14968	55%	12208
LANG	82928	47570	57%	35358
NET	8818	4448	50%	4370
SECURITY	4220	2154	51%	2066
SQL	3577	1776	49%	1801
TEXT	81036	43115	53%	37921
UTIL	18764	10109	53%	8655
Total	270932	146762	54%	124170

TAB. 4.1 – Tailles des codes opérationnels et des opérandes pour le code-octet JVM des bibliothèques de JDK 1.1

Il y a 201 codes opérationnels possibles pour la JVM (sans compter les codeops « quick » et l'instruction `breakpoint`. Le code opérationnel 186 n'est pas utilisé; les codes sont donc de 0 à 201). L'entropie des probabilités obtenues est approximativement 5.044 bits par code opérationnel. Puisque chaque code opérationnel originel est codé sur un octet, le facteur de compression moyen est de  $5.044/8 \approx 63.05\%$ . Le facteur final reporté dans la table 4.2 est de 64.3%; mais comme mentionné précédemment, le facteur exact est de  $93339/146762 \approx 63.56\%$ , c'est-à-dire une longueur moyenne de  $8 \times 0.6356 \approx 5.085$  bits par code opérationnel. Nous avons donc obtenu un facteur pratique très près du facteur théorique de l'entropie des probabilités. Pour cet exemple, la codification de Huffman s'avère très efficace. Une codification arithmétique ne pourrait d'aucune façon améliorer substantiellement la compression.

La spécialisation de formats a été appliquée à l'ensemble des groupes AWT, IO, LANG, NET, SECURITY, SQL, TEXT et UTIL.

Les résultats sont présentés par la table 4.4. Elle inclut les anciens formats et les fréquences d'utilisation de chaque format pour l'ensemble des groupes. Les mnémoniques à paramètres n'ayant pas obtenus de nouveaux formats ne sont pas listés, ce sont : `dload`, `dstore`, `fload`, `fstore`, `lload`, `lstore`, `ret`, `ldc_w`, `if_acmpeq`, `goto_w`, `jsr_w`, `multianewarray` et `wide`. De plus, ne sont pas présentés les mnémoniques ayant des spécialisations dont la somme des fréquences est inférieure à 100. Rappelons qu'un format est exprimé sous la forme d'un tuple, où le type de chaque paramètre est soit un entier non-signé ('u'), ou un entier signé ('s'). Le nombre de bits occupé par le paramètre suit le type. La fréquence

Groupe	Taille Codeops	Taille Codeops Compressé	Taille Code-octet Compressé	Facteur de Compression Code-octet	Facteur de Compression Codeops
AWT	22622	17991	39782	89%	80%
IO	14968	11933	24141	88%	80%
LANG	47570	24947	60305	72%	52%
NET	4448	3507	7877	89%	79%
SECURITY	2154	1682	3748	89%	78%
SQL	1776	1385	3186	89%	78%
TEXT	43115	24810	62731	77%	58%
UTIL	10109	8121	16776	89%	80%
Total	146762	94376	218546	80.6%	64.3%

TAB. 4.2 –. Facteur de compression Huffman, sur les codes opérationnels, pour les bibliothèques de JDK 1.1

d'un format est le nombre d'instruction l'utilisant; en d'autres mots, c'est la somme des fréquences de base des formats effectivement couverts. Une simplification a été appliquée pour les instructions de longueur variable. Pour les instructions `lookupswitch` et `tableswitch`, seulement les trois premiers paramètres ont été considérés pour la spécialisation. Il n'y a donc aucune diminution du nombre de bits pour le quatrième paramètre. Pour en tenir compte, il faudrait considérer les longueurs des instructions en plus de leur fréquence. De toute façon, leur fréquence d'utilisation est inférieure à 50.

Ces nouveaux formats permettent une codification compacte des opérandes. Ils ont été appliqués aux mêmes groupes. La table 4.6 présente les résultats des tailles et des facteurs obtenus. Il faut noter que les tailles des codes opérationnels ne changent pas. La colonne 'Taille Code-octet Compressé' présente l'espace occupé par les codes opérationnels sans compression, plus l'espace des opérandes compactés. Toutes les tailles sont en octet.

### 4.3 Compression avec macro-instructions

Pour effectuer cette expérience, nous utilisons les benchmarks BYTEmark et une partie de la bibliothèque de JDK 1.0.2, soient les groupes UTIL, IO, AWT, LANG. Ces benchmarks sont présentés à la section 4.4.2. Nous utiliserons ces benchmarks pour des mesures de vitesse d'exécution. La section 4.4.4 présente les résultats de temps d'exécution, avec macros, pour ces benchmarks.

La phase préliminaire de découpage a généré 8599 blocs élémentaires. Un bloc élémentaire

Mnémonique	Fréq	Mnémonique	Fréq	Mnémonique	Fréq
sipush	18906	dup	18031	bastore	13288
bipush	13056	aload_0	10104	getfield	6390
invokevirtual	5702	aload_1	3318	iconst_0	3253
iload	3143	invokespecial	3014	return	2143
sastore	2075	putfield	2057	ldc	1864
new	1727	aload	1714	aload_2	1629
goto	1624	iconst_1	1616	istore	1503
iload_2	1381	ireturn	1375	iload_1	1370
invokestatic	1236	areturn	1086	iload_3	1047
iadd	1016	ifeq	870	astore	828
athrow	763	aload_3	743	getstatic	677
iinc	615	if_icmplt	588	ifnull	575
iastore	566	checkcast	547	astore_2	521
isub	493	ifne	481	pop	466
astore_1	437	if_icmpne	436	monitorexit	406
istore_2	395	arraylength	350	ifnonnull	337
istore_3	325	astore_3	315	iand	314
invokeinterface	310	iconst_2	305	aconst_null	302
iconst_3	297	aaload	278	if_icmpeq	278
ldc2_w	262	iconst_m1	262	aastore	260
putstatic	260	iaload	246	if_icmple	238
newarray	233	ifge	219	istore_1	217
monitorenter	191	instanceof	190	if_icmpge	185
iload_0	184	lload	173	iconst_4	162
baload	157	iconst_5	152	lload_1	147
lconst_0	147	lcmp	144	iflt	140
jsr	137	castore	124	i2l	122
ishl	120	ifle	119	lstore	118
imul	111	ifgt	109	dload	107
if_acmpne	106	i2b	105	if_icmpgt	105

TAB. 4.3 -. Fréquences des mnémoniques pour l'ensemble des groupes awt, io, lang, net, security, sql, util et text

Mnémonique	Format standard	Fréquence	Nouveaux formats	Fréquence
bipush	(s 8)	2332	(s 7)	10724
sipush	(s 16)	807	(s 11)	9201
”			(s 14)	8898
getfield	(u 16)	1	(u 7)	5995
”			(u 9)	394
invokevirtual	(u 16)	1	(u 7)	5346
”			(u 9)	355
iload	(u 8)	4	(u 4)	2939
”			(u 5)	200
invokespecial	(u 16)	40	(u 7)	2974
goto	(s 16)	35	(s 8)	1426
”			(s 6)	1008
”			(s 10)	163
putfield	(u 16)	69	(u 7)	1988
new	(u 16)	56	(u 6)	1671
aload	(u 8)	56	(u 4)	1658
istore	(u 8)	114	(u 4)	1389
ldc	(u 8)	517	(u 4)	1347
invokestatic	(u 16)	37	(u 7)	1199
ifeq	(s 16)	3	(s 7)	773
”			(s 10)	94
astore	(u 8)	20	(u 4)	808
getstatic	(u 16)	1	(u 7)	606
”			(u 8)	70
iinc	(u 8 s 8)	44	(u 4 s 2)	571
if_icmplt	(s 16)	43	(s 8)	545
checkcast	(u 16)	5	(u 6)	542
ifnull	(s 16)	44	(s 7)	531
ifne	(s 16)	66	(s 6)	415
if_icmpne	(s 16)	66	(s 7)	370
invokeinterface	(u 16 u 8)	16	(u 8 u 2)	294
ifnonnull	(s 16)	47	(s 6)	290
if_icmpeq	(s 16)	24	(s 7)	254
putstatic	(u 16)	8	(u 7)	252
ldc2_w	(u 16)	18	(u 8)	244
newarray	(u 8)	1	(u 4)	232
if_icmple	(s 16)	12	(s 7)	226
instanceof	(u 16)	3	(u 6)	187
ifge	(s 16)	40	(s 6)	179
if_icmpge	(s 16)	32	(s 6)	153
iflt	(s 16)	20	(s 7)	131
jsr	(s 16)	10	(s 8)	127

TAB. 4.4 -. Nouveaux formats générés par l'algorithme de spécialisation pour les groupes AWT, IO, LANG, NET, SECURITY, SQL, TEXT et UTIL

Mnémonique	Format	Codeop	Mnémonique	Format	Codeop
aload_0	()	000	invokevirtual	(u 7)	0010
dup	()	0011	getfield	(u 5)	01000
iconst_0	()	01001	getfield	(u 7)	01010
bipush	(s 5)	01011	iload	(u 4)	01100
aload_1	()	01101	invokespecial	(u 6)	011100
return	()	011101	ireturn	()	011110
iconst_1	()	011111	iload_1	()	100000
iastore	()	100001	iload_2	()	100010
bipush	(s 8)	100011	aload	(u 4)	100100
aload_2	()	100101	istore	(u 4)	100110
putfield	(u 5)	100111	iadd	()	101000
new	(u 5)	1010010	iload_3	()	1010011
putfield	(u 7)	1010100	ldc2_w	(u 9)	1010101
ldc	(u 3)	1010110	sastore	()	1010111
areturn	()	1011000	lastore	()	1011001
goto	(s 6)	1011010	iload_0	()	1011011
iconst_5	()	1011100	isub	()	1011101
invokestatic	(u 6)	1011110	iconst_2	()	1011111
inc	(u 4 s 2)	1100000	getstatic	(u 7)	1100001
sipush	(s 11)	1100010	ifeq	(s 7)	1100011
aload_3	()	1100100	ifnull	(s 6)	11001010
if_icmplt	(s 7)	11001011	sipush	(s 16)	11001100
newarray	(u 4)	11001101	ifne	(s 6)	11001110
iconst_4	()	11001111	astore_2	()	11010000
ldc	(u 8)	11010001	iconst_3	()	11010010
astore	(u 3)	11010011	goto	(s 8)	11010100
aaload	()	11010101	aastore	()	11010110
astore_1	()	11010111	istore_2	()	11011000
iand	()	11011001	invoke <sup>a</sup>	(u 7 u 3 u 0)	11011010
istore_3	()	11011011	if_icmpne	(s 7)	11011100
athrow	()	11011101	arraylength	()	11011110
checkcast	(u 5)	11011111	istore_1	()	111000000
if_icmple	(s 7)	111000001	iconst_m1	()	111000010
iaload	()	111000011	astore_3	()	111000100
lconst_0	()	111000101	ifge	(s 7)	111000110
invokespecial	(u 7)	111000111	if_icmpge	(s 7)	111001000
ifle	(s 16)	111001001	putstatic	(u 6)	111001010
pop	()	111001011	if_icmpeq	(s 7)	111001100
if_icmpgt	(s 8)	111001101	iflt	(s 16)	111001110
aconst_null	()	111001111	ifnonnull	(s 7)	111010000
dload	(u 8)	111010001	iload	(u 8)	111010010

TAB. 4.5 -. Nouveaux formats et codes opérationnels de 82 instructions JVM

<sup>a</sup> invokeinterface

Groupe	Taille Opérandes	Taille Opérandes Compacts	Taille Code-octet Compressé	Facteur Code-octet	Facteur Opérandes
AWT	21791	11029	33651	76%	51%
IO	12208	6010	20978	77%	49%
LANG	35358	26968	74538	89%	76%
NET	4370	2121	6569	74%	49%
SECURITY	2066	945	3099	73%	46%
SQL	1801	826	2602	73%	46%
TEXT	37921	24741	67856	83%	65%
UTIL	8655	4679	14788	79%	54%

TAB. 4.6 -. Facteur de compression des opérandes en utilisant les formats de la table 4.4

Groupe	Taille Code-octet	Taille Code-octet Compressé	Facteur de Compression Code-octet
AWT	44413	29520	66%
IO	27176	18283	67%
LANG	82928	53638	64%
NET	8818	5733	65%
SECURITY	4220	2669	63%
SQL	3577	2260	63%
TEXT	81036	51989	64%
UTIL	18764	13021	69%
<b>Total</b>	<b>270932</b>	<b>177113</b>	<b>65%</b>

TAB. 4.7 -. Facteur de compression en utilisant les formats de la table 4.4 et les codes de Huffman

Benchmark	Facteur de Compression	Taille Code-octet
NumericSort	63.5%	773
StringSort	66.3%	1541
BitfieldOps	70.4%	831
FPemulation	71.5%	3903
Fourier	71.1%	640
Assignment	68.3%	1634
Encryption	69.8%	1800
Huffman	68.1%	1395
NeuralNet	89.4%	7467
LUdecompose	69.3%	1602

TAB. 4.8 –. Facteur de compression, sans macros, des benchmarks BYTEmark en utilisant les formats de la table 4.5 et les codes de Huffman

ne pouvant contenir, pour cette expérience, aucune instruction de branchement. C'est une simplification pour permettre la génération automatique du code C des macro-instructions dans l'interpréteur. Nous procédons selon le processus général établi au chapitre 2.

Nous limitons à 15 la longueur des séquences d'instructions des macros. La table 4.9 présente le nombre de séquences pour chaque longueur. La somme des fréquences d'occurrences de toutes ces séquences est 73089, répartie sur un total de 484 séquences. Notez que ces séquences peuvent se chevaucher. Ainsi, le nombre de substitutions effectives est moindre. En effet, plus bas nous montrons que le nombre de séquences effectivement couvertes est de 10255. L'ensemble de ces séquences est transformé en une base de macro-instructions en attribuant un nom et un format à chaque séquence. Ce nom est simplement la concaténation des mnémoniques formant la séquence. Naturellement, ce choix est arbitraire et n'a aucun impact sur la performance de la compression ou de la vitesse d'interprétation. Les formats initiaux de la séquence, c'est-à-dire de la macro-instruction, est la complétion des formats apparaissant dans le code (la complétion est effectuée par l'algorithme de la figure 2.5).

Naturellement, nous ne pouvons présenter la totalité de cette base, mais la table 4.10 présente quelques exemples de macro-instructions avec leur fréquence d'occurrence. Nous avons choisi des séquences avec des fréquences élevées, dont certaines se chevauchent. La plus fréquente est la macro-instruction ((`aload_0`) (`getfield *`)). Les séquences 4 et 5 se chevauchent, ainsi, lors de la sélection des macro-instructions, si l'une est choisie, la fréquence de l'autre sera diminuée. En général, le choix d'une séquence chevauchant une autre séquence

<b>Longueur</b>	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>Nombre</b>	112	60	23	19	20	22	23	26	29	31	34	31	28	26

TAB. 4.9 –. Nombre de séquences de la base de macro-instructions, selon leur longueur; au total, il y a 484 séquences

	Séquence	Format	Fréquence
1	(aload_0) (getfield *)	(u 7)	2333
2	(dup) (bipush *)	(s 8)	1299
3	(dastore) (dup)	()	1070
4	(dconst_0) (dastore)	()	654
5	(dconst_0) (dastore) (dup)	()	618
6	(putfield *) (aload_0)	(u 7)	527
7	(dastore) (dup) (bipush *)...		
	(dconst_0) (dastore) (dup)	(s 7)	457
8	(new *) (dup)	(u 7)	456
9	(ldc2_w *) (lastore) (dup)...		
	(bipush *) (ldc2_w *)...		
	(lastore) (dup) (bipush *)...		
	(ldc2_w *)	(u 9 s 7 u 9 s 7 u 9)	208

TAB. 4.10 –. Une partie du dictionnaire préliminaire de macro-instructions

n'empêche pas le choix de cette deuxième, mais par exemple, si la séquence 4 est choisie, la fréquence 5 deviendra zéro. Cette séquence ne sera alors pas choisie. Cependant, si la séquence 5 est choisie en premier, la séquence 4 pourrait l'être par la suite.

La table 4.11 présente les macro-instructions du dictionnaire final les plus fréquentes. Nous avons supposé un espace de 30 octets pour l'implantation, sur la machine hôte, d'une macro-instruction quelconque. Les fréquences mentionnées sont les nombres d'application des séquences sans chevauchement. Il y a un total de 70 macro-instructions dont la somme des fréquences est 10255.

Un cas intéressant à noter, la séquence (dup) (bipush \*) où sa fréquence de la base est passée de 1299 à la fréquence réelle de 406. Cette diminution fut causée par la sélection de d'autres séquences chevauchant une partie de ses occurrences. Pour sa part, la séquence (dconst\_0) (dastore) n'a pas été choisie. Elle fut entièrement couverte par le choix de

deux macro-instructions plus longues.

La macro-instruction 1 est nettement la plus fréquente. Cela se comprend par la fréquence élevée de `getfield` et sa sémantique, car celle-ci nécessite toujours une référence à un objet sur la pile. Ce qui se traduit souvent par l'instruction `aload` pour la charger. Les macros 2, 9, 11, 27, 28 et 34 sont similaires. De même, les macros faisant un appel de méthode sont fréquemment précédées d'une instruction empilant une référence à un objet<sup>1</sup>.

La macro-instruction 23, la plus longue, provient d'une longue séquence cyclique d'instructions de l'une des classes du JDK 1.0.2. Elle ne devrait probablement pas être insérée dans la machine virtuelle. C'est un cas particulier où l'échantillonnage devrait être ajusté pour éliminer des cas pathologiques. Mais elle représente peut-être une tendance réelle du compilateur Java ayant généré cette classe.

Au total, il y a 362 instructions dans la machine virtuelle avec macro-instructions. Cela inclut les nouveaux formats. Une partie de ces instructions est présentée à la table 4.12. Les mnémoniques `mi` réfèrent aux macro-instructions de la table 4.11. On peut voir que les codes opérationnels les plus courts ont cinq bits. Les plus longs, non présentés dans cette table, ont douze bits. On peut sommairement évaluer le gain en espace de certaines macro-instructions par rapport à la table 4.5. Par exemple, la macro `m1` a un codeop de cinq bits. Elle est une combinaison de deux instructions, dont les codeops avaient trois et cinq bits. Le format de `getfield` est le même, ainsi il y a un gain de trois bits pour chaque application de cette macro.

Les facteurs de compression pour une partie de la librairie JDK 1.0.2 sont présentés par la table 4.13. Ce sont les groupes utilisés par les benchmarks BYTEmark. La moyenne du facteur de compression est 60.8%. Les facteurs de compression pour les benchmarks BYTEmark, voir table 4.8, sont effectivement meilleurs que les facteurs sans macro-instructions (voir table 4.8). Dans le cas de NeuralNet, le changement est drastique, passant de 89.4% à 51.6%. C'est d'ailleurs un benchmark ayant un temps d'exécution relatif inférieur à 1 (voir les tables 4.21 et 4.22). Le facteur moyen de compression sur l'ensemble des benchmarks est 58.8%.

En conclusion, les facteurs de compression se comparent favorablement à d'autres méthodes. En particulier, à la compression par macro-instructions de [CSCM98] où des facteurs de 70% à 80% sont obtenus.

---

1. On pourrait conclure que dans la JVM originelle, ces instructions devraient avoir un argument spécifiant l'indice du champ, ou de la méthode, ainsi que l'objet référencé. En fait, il faudrait ajouter ces versions à deux arguments et garder les versions à un argument, car dans certains cas, la référence à l'objet n'est pas dans l'une des variables locales, mais bien sur le dessus de la pile. Ainsi, le problème devient la limite de 256 codes opérationnels dans la codification code-octet originelle.

	Séquence	Format	Fréquence
1	(aload_0) (getfield *)	(u 5)	1201
2	(aload_0) (getfield *)	(u 6)	741
3	(dup) (bipush *)	(s 6)	406
4	(new *) (dup)	(u 4)	365
5	(ldc2_w *) (lastore) (dup)	(u 9)	324
6	(putfield *) (aload_0)	(u 7)	321
7	(aload_0) (invokevirtual *)	(u 6)	269
8	(aload_0) (invokespecial *)	(u 5)	233
9	(aload_1) (getfield *)	(u 7)	229
10	(dastore) (dup)	()	221
11	(aload *) (getfield *)	(u 4 u 7)	219
12	(aload_1) (invokevirtual *)	(u 6)	189
13	(ldc *) (invokevirtual *)	(u 3 u 6)	188
14	(iconst_0) (ireturn)	()	186
15	(aload_0) (getfield *) (iload_1)... (aaload)	(u 7)	168
16	(dup) (getfield *)	(u 7)	166
17	(aload_0) (invokespecial *)	(u 7)	154
18	(aload_0) (iload_1)	()	153
19	(dastore) (dup) (bipush *)... (dconst_1)	(s 7)	151
20	(aload_0) (aload_1)	()	145
21	(iconst_1) (ireturn)	()	144
22	(iadd) (putfield *)	(u 7)	143
23	(dastore) (dup) (bipush *)... (dconst_1) (dastore) (dup) ... (bipush *) (dconst_0) ... (dastore)	(s 6 s 6)	132
24	(putfield *) (return)	(u 5)	131
25	(aload_0) (iconst_0)	()	128
26	(aastore) (dup)	()	125
27	(aload_0) (getfield *)	(u 7)	124
28	(aload_2) (getfield *)	(u 7)	122
29	(iload *) (iload *)	(u 4 u 4)	122
30	(iconst_0) (istore *)	(u 3)	120
31	(iload_0) (sipush *)	(s 13)	119
32	(iastore) (dup)	()	113
33	(astore_2) (aload_2)	()	112
34	(aload_3) (getfield *)	(u 7)	100
35	(aload_1) (invokespecial *)	(u 6)	98
36	(iload *) (bipush *)	(u 4 s 8)	97
37	(sipush *) (iand)	(s 9)	91
38	(astore_1) (aload_1)	()	89
39	(dup) (iconst_2)	()	88
40	(aload_0) (iconst_1)	()	84

TAB. 4.11 -. Une partie du dictionnaire final de macro-instructions

Instruction	Codeop	Instruction	Codeop
(m1 (u 5))	00000	(invokevirtual (u 6))	00001
(m2 (u 6))	000100	(aload_0 ())	000101
(return ())	000110	(bipush (s 6))	000111
(invokespecial (u 6))	001000	(dconst_0 ())	001001
(iload (u 4))	001010	(iload_3 ())	001011
(iload_2 ())	001100	(iload_1 ())	001101
(iconst_0 ())	001110	(iconst_1 ())	001111
(m3 (s 6))	010000	(aload_2 ())	010001
(dastore ())	010010	(invokestatic (u 6))	010011
(goto (s 6))	010100	(if_icmplt (s 8))	010101
(areturn ())	010110	(m4 (u 4))	010110
(aload_1 ())	0101111	(putfield (u 6))	0110000
(ireturn ())	0110001	(istore (u 4))	0110010
(m5 (u 9))	0110011	(m6 (u 7))	0110100
(m7 (u 6))	0110101	(goto (s 8))	0110110
(ifnull (s 6))	0110111	(ldc (u 4))	0111000
(ifeq (s 7))	0111001	(isub ())	0111010
(iadd ())	0111011	(m8 (u 5))	0111100
(getstatic (u 6))	0111101	(m9 (u 7))	0111110
(m10 ())	0111111	(getfield (u 6))	1000000
(m11 (u 4 u 7))	1000001	(dconst_1 ())	1000010
(aload (u 3))	1000011	(aload_3 ())	1000100
(m12 (u 6))	1000101	(m13 (u 3 u 6))	1000110
(m14 ())	1000111	(istore_2 ())	10010000
(bipush (s 8))	10010001	(iaload ())	10010010
(iinc (u 4 s 2))	10010011	(ifne (s 6))	10010100
(m15 (u 7))	10010101	(m16 (u 7))	10010110
(if_icmpne (s 7))	10010111	(m8 (u 7))	10011000
(m18 ())	10011001	(m19 (s 7))	10011010
(athrow ())	10011011	(ldc2_w (u 8))	10011100
(m20 ())	10011101	(m21 ())	10011110
(m22 (u 7))	10011111	(istore_3 ())	10100000
(aaload ())	10100001	(iconst_2 ())	10100010
(iconst_4 ())	10100011	(m23 (s 6 s 6))	10100100
(m24 (u 5))	10100101	(iconst_3 ())	10100110
(iload_0 ())	10100111	(m25 ())	10101000
(arraylength ())	10101001	(lload_1 ())	10101010
(m26 ())	10101011	(m27 (u 7))	10101100
(m28 (u 7))	10101101	(m29 (u 4 u 4))	10101110
(iastore ())	10101111	(m30 (u 3))	10110000
(m31 (s 13))	10110001	(checkcast (u 5))	10110010

TAB. 4.12 –. Une partie des 362 instructions et de leur code opérationnel de la machine virtuelle avec macro-instructions et nouveaux formats

Groupe	Facteur de Compression	Taille Non Compressé
UTIL	63.7%	7408
IO	60.1%	8655
AWT	58.2%	22908
LANG	63.0%	16796

TAB. 4.13 -. Facteur de compression de la librairie JDK 1.0.2 en utilisant des macros et formats de la table 4.12 et les codes de Huffman

Benchmark	Facteur de Compression	Diminution par rapport à sans macros
NumericSort	56.4%	7.1
StringSort	56.5%	9.8
BitfieldOps	65.8%	4.6
FPemulation	67.0%	4.5
Fourier	64.7%	6.4
Assignment	60.1%	8.2
Encryption	64.2%	5.6
Huffman	60.7%	7.4
NeuralNet	51.6%	37.8
LUdecompose	59.2%	10.0

TAB. 4.14 -. Facteur de compression des benchmarks BYTEmark en utilisant des macros et formats de la table 4.12 et les codes de Huffman

#### 4.4 Vitesses d'exécution de benchmarks Java compressés

Dans cette section, nous présentons des résultats expérimentaux de temps d'exécution de programmes Java compressés. Nous procédons à deux ensembles de tests, avec et sans macro-instructions. Les meilleurs temps obtenus sont avec macros.

Nous utilisons l'implantation Harissa [MMBC97] avec les modifications mentionnées à la section suivante.

##### 4.4.1 Modification de l'implantation Harissa

L'implantation Harissa doit être modifiée pour permettre l'utilisation des décodeurs de programmes compressés.

Cette implantation utilise quatre versions d'un même interprète, adaptées à différents contextes d'exécution d'une méthode Java : synchronisation, exceptions, profile dynamique ou normal. Chaque appel d'une méthode utilise la version appropriée. Pour le code compressé, nous gardons ces quatre versions. L'interprète itère continuellement en décodant et émulant les instructions de la méthode, mais fait un appel récursif lors d'une invocation d'une méthode Java. La partie de décodage et d'interprétation est entièrement implantée dans une seule fonction C. Dans la version originelle, le décodage est implanté par une instruction `switch` en utilisant le code opérationnel. Il y a un `case` pour chacune des instructions virtuelles.

La modification majeure consiste à retirer cette instruction `switch` pour la remplacer par un décodeur généré automatiquement par nos outils. Pour permettre cela, chaque `case` a été transformé en une macro-instruction C. Le décodeur généré référence ces macro-instructions pour implanter le code C à exécuter lors de l'émulation de l'instruction virtuelle. Cela est conforme à la méthode générale présentée à la figure 1.3. Les tables des décodeurs font partie de la fonction contenant l'interprète. Ils sont tous déclarés `static` pour éviter leur allocation sur la pile C à chaque appel d'une méthode Java.

La version compressée ne permet pas l'utilisation des instructions de type « quick ». Cela est nécessaire car le code compressé ne peut plus être modifié comme dans la version originelle. Dans la version originelle d'Harissa, ces instructions sont dynamiquement insérées dans le code-octet par l'interprète pour augmenter leur vitesse d'émulation subséquente. Pour tous les benchmarks, la version originelle utilise ces instructions. Notez qu'un compilateur Java ne génère pas d'instructions de type « quick ».

#### 4.4.2 Les benchmarks Java BYTEmark

Pour mesurer des temps d'exécutions de programmes Java compressés, nous utilisons dix benchmarks de tailles moyennes. Ce sont les benchmarks BYTEmark [BYT99]. Dans leur forme originelle, ces benchmarks effectuent des exécutions préliminaires pour mesurer la précision de l'horloge et établir le nombre d'itérations à effectuer pour rendre le temps d'exécution suffisamment long. Cette phase préliminaire a été éliminée et des nombres fixes d'itérations ont été choisis. La table 4.15 présente de brèves descriptions de ces benchmarks.

Leurs caractéristiques dynamiques seront utiles pour mieux comprendre les résultats de rapidité d'exécution. La table 4.16 présente leur profil dynamique. Ces profils indiquent les fréquences d'exécution des instructions JVM normalisées par rapport à 100. Nous avons retenu les fréquences supérieures à 1. Cette information nous permet de mieux comprendre le comportement du décodage ainsi que les décélérations observées.

Tous les tests sont effectués avec gcc version 2.8.1 pour le Sparc (processeur Sparc Ultra-1, antémémoire 1Mo, SunOS 5.6), et version 2.91.66 pour Pentium (processeur Pentium II, antémémoire de 32Ko, Linux). Nous utilisons une optimisation -O1.

Tous les décodeurs canoniques utilisent la forme-c pour l'accès mémoire. Il en est de même pour les décodeurs automates, avec une exception : la forme-b est utilisée pour le cas  $k_r = 8$ . Cela a été nécessaire à cause de l'espace prohibitif du code C, avec la forme-c pour ce cas.

#### 4.4.3 Vitesses d'exécution sans macro-instructions

Dans un premier temps, nous utilisons les benchmarks sans macro-instructions. Il y a toutefois une extension de l'ensemble des instructions de la machine JVM pour ajouter des instructions ayant de nouveaux formats. Ainsi, quarante nouveaux formats ont été ajoutés. Ce processus est fait automatiquement par l'algorithme de création d'instructions, présenté à la figure 2.7 (sans dictionnaire de macro-instructions).

Pour la conception de ces nouveaux formats, et le calcul des fréquences statiques pour l'attribution des codes opérationnels, nous utilisons la librairie JDK 1.0.2 et les benchmarks comme échantillons. Deux dictionnaires préliminaires sont créés, l'un pour la librairie et l'autre pour les benchmarks. Ces deux dictionnaires sont combinés, dans une proportion 1 pour 1. La table 4.5 présente les nouveaux formats, ainsi que des formats standards, avec leur code opérationnel. On peut y voir de nombreux nouveaux formats, plus courts que les formats standards. Par exemple, `getfield` existe sous la forme de deux nouveaux formats (`u 5`) et (`u 7`). L'instruction `putfield` reçoit la même considération. La plupart reçoivent un seul nouveau format.

( $B_0$ )NumericSort	Tri par monceau de 10000 nombres aléatoires de 32 bits. (338 lignes)
( $B_1$ )StringSort	Tri par monceau de 5000 chaînes aléatoires. (576 lignes)
( $B_2$ )BitFieldOps	Exécute une série d'opérations binaires sur un vecteur de 3000 bits. (400 lignes)
( $B_3$ )FPemulation	Émulation d'opérations point-flottantes sur un vecteur de 100 nombres. (1114 lignes)
( $B_4$ )Fourier	Approximation de l'intégrale définie de $(x + 1)^x$ entre 0 et 2 par 200 trapèzes. (360 lignes)
( $B_5$ )Assignment	Implante l'algorithme <i>Quantitative Decision Making for Business</i> (Gordon, Pressman, and Cohn; Prentice-Hall) sur un tableau à trois dimensions. (483 lignes)
( $B_6$ )Encryption	Effectue une encryption et une décryption, cinq fois sur un texte de 4000 caractères, selon la méthode IDEA (International Data Encryption Algorithm) (voir Schneier [Sch94]). (685 lignes)
( $B_7$ )Huffman	Effectue une compression et une décompression à la Huffman d'un texte créé aléatoirement utilisant cinquante mots fixes. (599 lignes)
( $B_8$ )NeuralNet	Applique un apprentissage à la « Backpropagation » sur un réseau de neurones. (758 lignes)
( $B_9$ )LUdecompose	Effectue la résolution d'un système de 101 équations linéaires par décomposition LU. (511 lignes)

TAB. 4.15 –. Description des dix benchmarks BYTEmark

StringSort		Encryption		Huffman		BitFieldOps		FPemulation	
Mne	Prob	Mne	Prob	Mne	Prob	Mne	Prob	Mne	Prob
iloal	17.89	iloal	20.74	getfield	16.07	iloal_1	7.95	iconst_0	11.90
aload_0	13.25	istore	9.70	aload_0	12.10	iinc	7.95	iaload	9.14
getfield	12.98	iinc	6.19	iloal	10.60	iloal	6.96	iconst_1	8.84
iloal_1	7.13	ldc2_w	5.23	iloal_1	6.76	istore	6.96	aload_1	7.59
aaload	6.68	i2l	5.23	aaload	5.59	aload_0	3.97	iastore	5.63
iaload	6.44	iaload	4.77	iinc	4.28	getfield	3.97	aload	4.98
iadd	6.37	aload_3	3.99	istore	3.40	iloal_2	3.97	lload	4.32
iinc	5.91	ixor	3.69	if_icmpne	3.15	iloal_3	3.97	lstore	3.93
iastore	5.89	ldc	3.39	iconst_1	2.54	iaload	3.97	aload_0	3.71
if_icmplt	5.88	iand	3.23	baload	2.45	bipush	3.97	ifeq	3.23
dup2	5.72	iadd	2.88	bipush	2.31	irem	3.97	iand	2.66
		l2i	2.61	iloal_3	2.16	iastore	3.97	getfield	2.63
		lmul	2.61	bastore	1.66	ifgt	3.97	ldc	2.48
		land	2.61	ifne	1.58	iconst_5	3.97	iadd	2.03
		lrem	2.61	iloal_2	1.58	dup2	3.97	ishl	2.02
		aload_0	2.35	invoke <sup>1</sup>	1.55	ishr	3.97	i2l	1.89
		getfield	1.83	goto	1.55	ifeq	2.98	invoke <sup>2</sup>	1.64
		iloal_1	1.07	i2b	1.50	iconst_1	2.98	getstatic	1.45
		ireturn	1.03	irem	1.43	ishl	2.98	land	1.45
				istore_3	1.42	iconst_m1	1.50	return	1.38
				iconst_3	1.40	iand	1.50	iloal_2	1.13
				ishr	1.40	ixor	1.50	istore_2	1.13
				ishl	1.40	goto	1.47	aload_2	1.10
				iconst_m1	1.20	ior	1.47	lcmp	1.00
				iand	1.19				
LUdecompose		NumericSort		Fourier		Assignment		NeuralNet	
Mne	Prob	Mne	Prob	Mne	Prob	Mne	Prob	Mne	Prob
iloal	20.66	aload_0	11.45	dload	15.20	iloal	12.44	aload_0	13.89
aaload	15.83	iloal_1	11.45	iloal	13.78	aload_0	12.04	getfield	13.85
aload_0	9.22	iloal	11.39	dload_1	9.20	getfield	12.03	iloal	13.40
getfield	9.18	getfield	11.29	dadd	9.12	aaload	11.63	aaload	8.62
iloal_1	8.03	aaload	11.28	invoke <sup>2</sup>	6.14	iloal_3	8.99	daload	8.29
daload	7.63	iloal_2	9.98	dstore	6.11	if_icmplt	6.86	dadd	4.15
if_icmplt	4.08	iaload	8.39	dmul	6.11	iinc	6.81	dmul	4.13
iinc	3.98	iadd	4.32	if_icmpne	4.56	iloal_1	5.16	iloal_1	4.04
dload	3.90	if_icmpge	4.12	aload_0	3.11	iaload	5.15	dastore	4.00
dmul	3.68	iloal_3	3.00	dload_3	3.08	ifne	4.95	iloal_2	3.77
dstore	3.14	iastore	2.89	iconst_1	3.07	lcmp	1.73	if_icmplt	3.48
dsub	2.92	istore	2.71	invoke <sup>1</sup>	3.07	lconst_0	1.73	dload	3.41
iloal_3	2.42	if_icmple	1.53	dreturn	3.07	i2l	1.73	iinc	3.30
dastore	1.06	goto	1.50	iinc	3.06	saload	1.54	dup2	1.95
		istore_2	1.48	dconst_1	3.05	iconst_1	1.52	dload_2	1.71
		iconst_1	1.41	ifne	3.05	aload_2	1.33	dstore	1.63
				ifgt	3.04	if_icmpne	1.20	iloal_3	1.26
				iconst_2	1.52			dconst_0	1.07

TAB. 4.16 -. Profils dynamiques des benchmarks BYTEmark Java

	Temps Absolu		Temps Relatif			
	Non-compressé		Compressé			
	Pentium	Sparc	Pentium		Sparc	
			$C_{k_r=7}$	$C_{k_r=10}$	$C_{k_r=7}$	$C_{k_r=10}$
NumericSort	2.75	3.99	1.42	1.40	1.26	1.16
StringSort	7.68	10.35	1.29	1.26	1.09	1.17
BitfieldOps	5.11	6.21	1.63	1.60	1.33	1.41
FPemulation	3.82	5.29	1.92	1.47	1.43	1.17
Fourier	1.83	2.24	2.06	1.60	2.30	1.20
Assignment	1.49	2.42	1.25	1.24	1.08	1.11
Encryption	5.40	6.46	1.17	1.07	1.57	1.27
Huffman	2.50	3.98	1.47	1.29	1.28	1.16
NeuralNet	27.8	46.64	1.17	1.21	1.27	1.09
LUdecompose	3.29	4.60	1.46	1.55	1.27	1.22

TAB. 4.17 –. Temps absolus et relatifs des benchmarks Java BY-TEmark, sans macros, pour deux décodeurs canoniques, sur deux processeurs, par l'implantation Harissa modifiée

Les résultats des temps d'exécution des benchmarks sont présentés par les tables 4.17 et 4.18 ; l'une pour des décodeurs canoniques et l'autre pour des décodeurs automates.

Deux décodeurs canoniques sont utilisés, soient  $C_{k_r=7}$  et  $C_{k_r=10}$ . Comme leur nom le suggère, la racine de  $C_{k_r=7}$  a sept bits et  $C_{k_r=10}$  a une racine de dix bits. La structure exacte du décodeur  $C_{k_r=7}$  est  $k_r = 7, L_{8-12}, Lp_{1111101}, Lp_{1111011}$ . La structure de  $C_{k_r=10}$  est  $k_r = 10, L_{10-11}, Lp_{1111101100}$ . La table 4.19 présente les temps **théoriques** de décodage pour les dix benchmarks selon la fonction  $T(A)$  définie en 3.4. Cette évaluation tient compte des fréquences dynamiques des longueurs des codes opérationnels et de la structure du décodeur. Elle ne tient pas compte de l'émulation des instructions virtuelles. Ce sont des temps théoriques car ceux-ci ont été évalués sans tenir compte du processeur hôte. Nous avons fait cette évaluation pour mieux percevoir l'impact des longueurs des codes opérationnels versus la structure des décodeurs canoniques. On peut voir que le benchmark Fourier a le temps de décodage le plus élevé. C'est la raison majeure qui explique les pires temps d'exécutions de ce benchmark. NeuralNet a le deuxième temps de décodage le plus élevé. À l'inverse, son ralentissement en temps d'exécution réel est minime. Cela s'explique par une exécution répétée de l'instruction `getfield` qui a une granularité élevée (cela masque le temps de décodage).

Le ralentissement varie de 9% à 41% pour le décodeur  $C_{k_r=10}$  sur Sparc, mais varie de

	Temps Relatif Compressé					
	Pentium			Sparc		
	$A_{k_r=6}$	$A_{k_r=7}$	$A_{k_r=8}$	$A_{k_r=6}$	$A_{k_r=7}$	$A_{k_r=8}$
NumericSort	1.15	1.23	1.29	1.25	1.21	1.22
StringSort	1.27	1.25	1.20	1.18	1.21	1.16
BitfieldOps	1.77	1.53	1.61	1.55	1.51	1.64
FPemulation	1.40	1.36	1.26	1.25	1.46	1.42
Fourier	1.88	1.68	1.40	1.58	1.54	1.80
Assignment	1.24	1.20	1.17	1.10	1.24	1.24
Encryption	1.74	1.44	1.40	1.57	1.51	1.72
Huffman	1.29	1.30	1.24	1.15	1.25	1.24
NeuralNet	1.20	1.16	1.14	1.11	1.17	1.13
LUdecompose	1.41	1.34	1.39	1.23	1.35	1.34

TAB. 4.18 -. Temps relatifs des benchmarks Java BYTEmark, sans macros, pour trois décodeurs automatés, sur deux processeurs, par l'implantation Harissa modifiée

Décodeur	Benchmark									
	$B_0$	$B_1$	$B_2$	$B_3$	$B_4$	$B_5$	$B_6$	$B_7$	$B_8$	$B_9$
$C_{k_r=7}$	8.88	8.83	9.43	9.52	11.62	9.62	9.37	9.15	10.20	10.17
$C_{k_r=10}$	7.00	7.04	7.52	7.30	8.94	7.10	7.80	7.45	8.29	7.86

TAB. 4.19 -. Temps théorique moyen de décodage des benchmarks BYTEmark Java pour deux décodeurs canoniques

7% à 60% pour ce même décodeur sur Pentium. Dans la plupart des cas, de meilleures performances sont obtenues pour le processeur Sparc. Une analyse du code assembleur explique ce fait : pour le Pentium, la variable `rd` n'est pas attribuée à un registre.

Trois décodeurs automatiques ont été utilisés pour les deux processeurs. Rappelons que le cas  $k_r = 8$  utilise la forme-b, mais que les autres décodeurs utilisent la forme-c ; Ce qui explique le manque de performance du passage de sept à huit bits. Par rapport aux décodeurs canoniques, les performances ne sont pas généralement meilleures. Cela s'explique par le fait que sept bits sont insuffisants pour décoder fréquemment plus d'une instruction par cycle. L'avantage des décodeurs automatiques est dans ce potentiel de décodage multiple, mais la taille du code C générée est trop élevée pour permettre une racine de plus de huit bits. Pour améliorer leur performance et vraiment utiliser leur potentiel, il faudrait analyser une forme hybride de ce décodeur où la racine devrait utiliser un nombre de bits supérieur à la longueur moyenne des codes opérationnels. De plus, il faudrait probablement séparer les arguments des codes opérationnels en deux flots de décodage. Il faudrait aussi fournir au constructeur de décodeurs, un profil dynamique des programmes typiques à exécuter. Cela permettrait de connaître les paires d'instructions adjacentes fréquemment exécutées. Le décodeur serait généré en privilégiant les paires les plus fréquentes.

En se référant à la table 4.16, on remarque que les instructions `aload_0`, `getfield`, `iload_1`, `iload` et `aaload` sont très fréquemment exécutées pour huit des dix benchmarks. Par exemple, pour `StringSort`, c'est 58% des instructions exécutées. Ils ont des codes opérationnels courts ; ce qui diminue le temps d'exécution de ces benchmarks.

`Encryption` utilise fréquemment un autre ensemble d'instructions, mais `iload` est largement utilisée. Cette instruction a un code opérationnel court. (Notez que c'est la version (u 4) qui est fréquemment utilisée.) Mais les autres instructions ont des codes opérationnels longs. Le temps d'exécution devrait être un peu plus lent que les autres benchmarks, mais sans différence majeure.

`BitFieldOps` utilise un large éventail d'instructions. Douze instructions ont la même fréquence dynamique, soit 3.97 ; ce qui totalise 48% des instructions exécutées. Parmi ces instructions, plusieurs ont des codes opérationnels longs ; ce qui explique, en partie, les mauvaises performances de ce benchmark. Un autre facteur majeur est le pourcentage des instructions exécutées de faible granularité, qui est de 78%. Le décodage au niveau du bit apparaît plus lent. Encore une fois, il faudrait ajouter des macro-instructions pour couvrir les paires d'instructions de faibles granularités.

Pour sa part, `Fourier` utilise plutôt les instructions `dload`, `dload_1` et `dadd` (en plus de `iload` comme les autres benchmarks). Ces instructions ont des codes opérationnels longs ; ce qui aura un impact négatif sur le temps d'exécution. Les résultats des tables 4.17 et 4.18 confirment ce fait.

Identification	Noeuds de type 2	Noeuds de type 3
$C_{k_r=12}$	aucun	aucun
$C_{k_r=11}$	L <sub>12</sub>	aucun
$C_{k_r=10}$	L <sub>11-12</sub>	Lp_1111100101
$C_{k_r=9}$	L <sub>10-12</sub>	Lp_111110010
$C_{k_r=8}$	L <sub>9-12</sub>	Lp_11111001 Lp_11110100 Lp_11101001
$C_{k_r=7}$	L <sub>8-12</sub>	Lp_1111010 Lp_1111100 Lp_1110100 Lp_1100010
$C_{k_r=6}$	L <sub>7-12</sub>	Lp_111110 Lp_111110010 Lp_111101 Lp_111010 Lp_110001

TAB. 4.20 – Structures des sept décodeurs canoniques

#### 4.4.4 Vitesses d'exécution avec macro-instructions

Cette section présente des vitesses d'exécution des benchmarks en utilisant des machines virtuelles avec macro-instructions. Nous utilisons seulement des décodeurs canoniques, car les tailles des interprètes avec macros et automates sont prohibitives. Non seulement les meilleurs facteurs de compression sont obtenus avec macro-instructions, mais aussi les performances de vitesse.

Les macro-instructions sont implantées par concaténation du code des instructions de base. Nous n'avons fait aucune modification aux instructions de base combinées. Ainsi, la génération des machines virtuelles, à partir des instructions de base, peut se faire simplement et automatiquement sans connaître la sémantique des instructions. Cela est possible car les macro-instructions ne contiennent pas d'instructions de branchement. Les vitesses reportées sont donc conservatrices, car il serait possible d'optimiser le code des instructions combinées pour augmenter la vitesse. Toutefois, nous ne le faisons pas pour maintenir une génération automatique de l'interprète. Pour optimiser les instructions combinées et les générer automatiquement, il faudrait concevoir un langage descriptif de la sémantique des instructions de base.

Nous analysons sept décodeurs canoniques, dont les racines varient de six à douze bits. Puisque le code opérationnel le plus long à douze bits, il n'y a aucun avantage à en utiliser plus. Les structures des sept décodeurs sont présentées par la table 4.20. Les tables 4.21 et 4.22 présentent tous les temps relatifs pour deux processeurs.

Six benchmarks ont des ralentissements d'exécution en-deça de 10%, pour plusieurs décodeurs. Ce qui démontre l'avantage d'ajouter des instructions à la machine virtuelle, à la fois pour compresser et diminuer le temps de décodage.

Pour les benchmarks *Assignment* et *NeuralNet* une légère augmentation en vitesse se produit pour le *Pentium*. Même phénomène pour le benchmark *LUdecompose*, sur le *Sparc*.

Les cas les plus lents sont *BitfieldOps*, *FPemulation*, *Fourier* et *Encryption*.

Le benchmark *Encryption* a l'une des pires performances pour le *Pentium*. Il est de 31% plus lent, relatif à la version non compressée, dans les meilleurs cas. Mais pour le *Sparc*, il a de bonnes performances pour trois décodeurs. Une trace d'exécution révèle qu'il y a plusieurs macro-instructions exécutées, dans des itérations répétitives. Parmi celles-ci, les macros (*istoreiload (u 5 u 5)*), (*iloadiinc (u 4 u 4 s 2)*), (*iinciload (u 4 s 5 u 4)*), (*iloadiload u 4 u 4*), (*iloadiadd (u 5)*). Il demeure qu'un grand nombre d'instructions de faible granularité sont exécutées sans être dans des macro-instructions. Notamment, *i2l*, *l2i*, *lrem*, *ixor*, *land* et *lmul*. Pour résoudre ce problème, il faudrait ajouter des macro-instructions similaires à (*(iaload) (i2l)*), (*(ixor) (istore)*), (*(ldc2\_w) (lrem)*) et (*(ldc2\_w) (land) (l2i)*). Son comportement, pour le *Sparc*, se dégrade assez rapidement si le nombre de bits à la racine du décodeur est inférieur à neuf. Cela s'explique facilement, car un bon nombre d'instructions exécutées ont plus de huit bits.

Une analyse des benchmarks *BitfieldOps*, *FPemulation* et *Fourier* présentent des situations similaires. Pour augmenter leur vitesse, il faudrait des macro-instructions mieux adaptées à leur particularité.

Nous ne présentons pas les performances des décodeurs automates pour le cas avec macro-instructions, car les performances ne sont pas appréciablement meilleures que les décodeurs canoniques.

En conclusion, en utilisant des décodeurs appropriés, les ralentissements observés sont minimes pour six des dix benchmarks. Les ralentissements se comparent favorablement à d'autres méthodes n'utilisant pas la compression par codage de Huffman. Par exemple, elles se comparent bien au résultat de [CSCM98] tout en obtenant un meilleur facteur de compression.

#### 4.5 Conclusion pour ce chapitre

L'une des machines virtuelles les plus connues et répandues est sans contredit la *JVM*. La conception de son code-octet est, à notre avis, relativement bien fait pour obtenir un code compact<sup>2</sup>. Néanmoins, comme le démontrent les sections précédentes, il est possible

---

2. Toutefois, le bassin de constantes est loin d'être conçu pour obtenir des fichiers objets compacts ; mais cela sort du cadre de notre étude. Rayside *et al.* [RMH99], Horspool et Corless [HC99] discutent de méthodes

	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$	$C_{k_r=11}$	$C_{k_r=12}$
NumericSort	1.09	1.11	1.05	1.05	1.05	1.07	1.07
StringSort	1.09	1.08	1.05	1.02	1.02	1.02	1.02
BitfieldOps	1.44	1.42	1.36	1.28	1.32	1.28	1.24
FPemulation	1.31	1.25	1.21	1.18	1.17	1.17	1.15
Fourier	1.38	1.30	1.32	1.24	1.24	1.19	1.24
Assignment	1.03	1.02	1.01	1.00	0.97	0.98	0.98
Encryption	1.51	1.44	1.41	1.31	1.33	1.36	1.31
Huffman	1.13	1.11	1.08	1.08	1.09	1.08	1.10
NeuralNet	1.03	1.03	1.01	1.00	0.99	0.99	0.99
LUdecompose	1.13	1.09	1.09	1.03	1.03	1.01	1.00

TAB. 4.21 – Temps relatifs des benchmarks BYTEmark compressés avec macros, pour sept décodeurs canoniques, sur Pentium, par l’implantation Harissa modifiée

	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$	$C_{k_r=11}$	$C_{k_r=12}$
NumericSort	1.13	1.21	1.00	1.04	1.03	1.03	1.01
StringSort	1.15	1.20	1.05	1.09	1.03	1.06	1.10
BitfieldOps	1.71	1.43	1.36	1.55	1.27	1.28	1.27
FPemulation	1.38	1.31	1.21	1.22	1.15	1.16	1.20
Fourier	1.57	1.44	1.30	1.25	1.24	1.22	1.22
Assignment	1.15	1.22	1.02	1.03	1.02	1.02	1.00
Encryption	1.67	1.38	1.28	1.12	1.09	1.09	1.19
Huffman	1.23	1.23	1.17	1.20	1.09	1.08	1.15
NeuralNet	1.09	1.13	1.05	1.03	1.03	1.03	1.02
LUdecompose	1.11	1.16	1.25	1.04	0.98	0.98	1.12

TAB. 4.22 – Temps relatifs des benchmarks BYTEmark compressés avec macros, pour sept décodeurs canoniques, sur Sparc, par l’implantation Harissa modifiée

de faire mieux.

L'une des limitations de tout code-octet est le nombre de codes opérationnels disponibles sur un octet, soit  $2^8$ . Bien entendu, il est possible d'ajouter un mécanisme pour augmenter cette limite, même pour le code-octet. Mais le fait demeure que les concepteurs sont réticents à le faire pour ne pas augmenter la complexité des décodeurs, et la perte potentielle en temps de décodage. Cette limitation ne permet pas d'augmenter la machine par de nombreuses instructions permettant un code plus compact.

Pour la JVM, le gain en espace obtenu, se situant autour de 60%, est principalement attribuable aux quatre méthodes : compression des codes opérationnels, compaction des opérandes, macro-instructions et, bien sûr, instructions et opérandes non alignés sur une frontière d'octet.

## Chapitre 5

# MACHINA, UNE MACHINE VIRTUELLE GÉNÉRALE

---

Au chapitre précédent, nous avons appliqué les méthodes de compression et de décodage à une machine existante. Dans ce cas, la conception d'un ensemble d'instructions de base avait déjà fait l'objet d'une attention de la part de leur concepteur. Nous avons dû garder les instructions de base dans les machines virtuelles.

Dans ce chapitre, notre objectif est de montrer qu'il est possible de construire automatiquement une machine virtuelle pour exécuter des programmes compressés à partir d'instructions de base générales. C'est par un échantillonnage approprié qu'il sera possible de concevoir d'autres instructions adaptées aux programmes à exécuter. Nous appelons *Machina* la machine générale avec ses instructions de base. Notez que cette machine n'a pas d'environnement complexe, dont un gestionnaire de monceau (« garbage-collector »).

Les programmes échantillons objets analysés proviennent de programmes Scheme[KCE98]. Nous avons conçu un compilateur Scheme générant des instructions *Machina*. À partir de benchmarks, il sera possible de concevoir de nouvelles instructions pour mieux satisfaire les besoins particuliers du compilateur Scheme conçu ; bien entendu, avec l'objectif de générer des programmes compressés tout en maintenant un temps d'exécution rapide.

Il est important de lire la section 5.2 pour mieux comprendre l'objectif des expériences proposées dans ce chapitre. Les résultats expérimentaux se trouvent dans les sections 5.7 et 5.8. Nous présentons la structure générale de *Machina* à la section suivante.

### 5.1 La structure générale de *Machina*

Notre machine virtuelle générale utilise une pile pour les opérandes des instructions et un monceau pour les objets à durée de vie indéterminée. La pile sera dénotée par  $P$  et le monceau par  $M$ . Il y a deux tables globales, statiquement allouées au chargement d'un programme : la table des constantes ( $C$ ) et la table des variables globales ( $G$ ). La table des constantes peut contenir des adresses et des chaînes de caractères.

Il y a trois registres spéciaux :  $sp$ ,  $hp$  et  $pc$ . Le registre  $sp$  pointe le dernier mot empilé sur la pile, le registre  $hp$  pointe le prochain mot disponible dans le monceau et le registre  $pc$  pointe la prochaine instruction à exécuter. Le registre  $pc$  est automatiquement augmenté après le chargement d'une instruction, et avant son exécution, pour pointer sur l'instruction suivante. Ainsi, lors de l'exécution d'une instruction accédant le registre  $pc$ , celui-ci pointe

déjà l'instruction suivante. Notez le fait technique important que nous parlons ici du *pc* de la machine virtuelle tel que perçu par le programmeur (constructeur de compilateur). Dans l'implantation effective de la machine virtuelle, il y aura un *pc*, mais celui-ci ne pointe pas nécessairement l'instruction suivante car, à ce niveau, le programme est compressé. Toutefois, cela ne peut être détecté par le programmeur qui ne peut directement accéder ce registre. L'instruction JSR empile le *pc*, et, effectivement, un traitement spécial doit être fait au niveau de l'implantation de cette instruction pour donner l'illusion que le *pc* pointe l'instruction suivante (voir la section 5.5).

Nous utilisons un nombre restreint d'instructions de base, car c'est la tâche de notre système d'étendre cet ensemble. Il ne faut donc pas s'attendre, au départ, à un ensemble complexe d'instructions spécialisées.

La table 5.1 expose les instructions élémentaires à partir desquelles nous pouvons faire la conception d'une machine virtuelle. Si une instruction a un opérande, c'est un entier. La représentation des entiers dans la machine, c'est-à-dire dans la pile, la table des constantes, et la table des variables globales est le complément à deux.

La codification de base des opérandes des instructions avec paramètres est présentée par la table 5.2. Le format standard est celui utilisé lors des comparaisons des mesures de compacité et de rapidité. Par ce standard, nous n'avons pas utilisé de formats trop long pour rendre l'étude de la compression des formats plus significative. Il s'agit des formats les plus courts possibles pour coder tous les benchmarks sous forme code-octet. Pour les machines exécutant des programmes compressés, les formats par défaut seront les maximums. Ces machines seront donc plus générales en ce qui a trait aux formats des instructions.

## 5.2 La compilation de Scheme vers Machina

Nous utilisons les instructions Machina comme instructions de base pour compiler des programmes Scheme. Machina n'a aucune instruction particulière pour Scheme. Le compilateur Scheme doit faire usage des instructions de base pour implanter des opérations comme la création de fermeture, l'appel de fonction, l'étiquetage des données, etc.

Nous utilisons la partie frontale du compilateur Gambit [FMRW97] pour effectuer la compilation d'un programme Scheme. Nous ne compilons pas tous les types de base (nombres inexacts, nombres complexes).

Le compilateur génère des programmes de type « prgm » qui sont traduits, par la suite, sous la forme « code ». Nous présentons, en annexe, le résultat de la compilation du programme fib. En particulier, la représentation intermédiaire « prgm » est utilisée pour faciliter la mise au point du compilateur.

En annexe, à la figure 3, nous présentons un exemple d'un programme complet résultant

add	$P[sp - 1] \leftarrow P[sp] + P[sp - 1]; sp \leftarrow sp - 1.$
sub	$P[sp - 1] \leftarrow P[sp - 1] - P[sp]; sp \leftarrow sp - 1.$
mul	$P[sp - 1] \leftarrow P[sp] \times P[sp - 1]; sp \leftarrow sp - 1.$
div	$P[sp - 1] \leftarrow P[sp - 1] \text{ div } P[sp]; sp \leftarrow sp - 1.$
rem	$P[sp - 1] \leftarrow P[sp - 1] \text{ rem } P[sp]; sp \leftarrow sp - 1.$
and	$P[sp - 1] \leftarrow P[sp] \wedge P[sp - 1]; sp \leftarrow sp - 1.$
or	$P[sp - 1] \leftarrow P[sp] \vee P[sp - 1]; sp \leftarrow sp - 1.$
xor	$P[sp - 1] \leftarrow P[sp] \oplus P[sp - 1]; sp \leftarrow sp - 1.$
asl	$P[sp - 1] \leftarrow P[sp - 1] \ll P[sp]; sp \leftarrow sp - 1.$
asr	$P[sp - 1] \leftarrow P[sp - 1] \gg P[sp]; sp \leftarrow sp - 1.$
lsr	$P[sp - 1] \leftarrow (\text{non-signé})P[sp - 1] \gg P[sp]; sp \leftarrow sp - 1.$
eq	$P[sp - 1] \leftarrow \text{Si } P[sp] = P[sp - 1] \text{ Alors vrai Sinon faux}; sp \leftarrow sp - 1.$
neq	$P[sp - 1] \leftarrow \text{Si } P[sp] \neq P[sp - 1] \text{ Alors vrai Sinon faux}; sp \leftarrow sp - 1.$
gt	$P[sp - 1] \leftarrow \text{Si } P[sp] < P[sp - 1] \text{ Alors vrai Sinon faux}; sp \leftarrow sp - 1.$
lt	$P[sp - 1] \leftarrow \text{Si } P[sp] > P[sp - 1] \text{ Alors vrai Sinon faux}; sp \leftarrow sp - 1.$
not	$P[sp] \leftarrow \neg P[sp]; sp \leftarrow sp + 1.$
exg	$tmp \leftarrow P[sp]; P[sp] \leftarrow P[sp - 1]; P[sp - 1] \leftarrow tmp.$
dup	$P[sp + 1] \leftarrow P[sp]. sp \leftarrow sp + 1$
pushli <i>i</i>	$P[sp] \leftarrow P[sp - (P[sp] + i)].$
pushi <i>i</i>	$P[sp + 1] \leftarrow i; sp \leftarrow sp + 1.$
push <i>i</i>	$P[sp + 1] \leftarrow \&C_i; sp \leftarrow sp + 1.$
pushl <i>i</i>	$P[sp + 1] \leftarrow P[sp - i]; sp \leftarrow sp + 1.$
pushg <i>i</i>	$P[sp] \leftarrow G[i]; sp \leftarrow sp + 1.$
pusha	$P[sp - 1] \leftarrow M[P[sp] \times 4 + P[sp - 1]]; sp \leftarrow sp - 1.$
pushac	$P[sp - 1] \leftarrow M[P[sp] + P[sp - 1]]; sp \leftarrow sp - 1.$
pop <i>i</i>	$sp \leftarrow sp - i;$
popv	$sp \leftarrow sp - (1 + P[sp]);$
storea	$M[P[sp] \times 4 + P[sp - 1]] \leftarrow P[sp - 2]; sp \leftarrow sp - 3.$
alloc <i>i</i>	$hp \leftarrow P[sp]; P[sp + 1] \leftarrow sp \leftarrow sp + 1$
storeac	$M[P[sp] + P[sp - 1]] \leftarrow P[sp - 2]; sp \leftarrow sp - 3.$
storel <i>i</i>	$P[sp - i] \leftarrow P[sp]; sp \leftarrow sp - 1.$
storeli <i>i</i>	$P[sp - (P[sp] + i)] \leftarrow P[sp]; sp \leftarrow sp - 2.$
storeg <i>i</i>	$G[i] \leftarrow P[sp]; sp \leftarrow sp - 1.$
br <i>d</i>	$pc \leftarrow pc + d.$
bf <i>d</i>	$sp \leftarrow sp - 1; \text{Si } P[sp + 1] = \text{faux} \text{ Alors } pc \leftarrow pc + d.$
jmp	$pc \leftarrow P[sp]; sp \leftarrow sp - 1.$
jsr	$tmp \leftarrow P[sp]; P[sp] \leftarrow pc; pc \leftarrow tmp.$
ret <i>i</i>	$pc \leftarrow P[sp]; sp \leftarrow sp - i - 1.$
readc	$P[sp + 1] \leftarrow \text{lecture d'un caractère}; sp \leftarrow sp + 1.$
writec	Afficher le caractère $P[sp]. sp \leftarrow sp - 1$
stop	Arrêt du programme en exécution.

TAB. 5.1 – Les instructions de base de Machina

Instruction	Format standard	Format maximum
<code>br d</code>	(s 24)	(s 32)
<code>bf d</code>	(s 24)	(s 32)
<code>pushi i</code>	(s 24)	(s 32)
<code>pushli i</code>	(u 8)	(u 32)
<code>push i</code>	(u 16)	(u 32)
<code>pushl i</code>	(u 8)	(u 32)
<code>pushg i</code>	(u 16)	(u 32)
<code>pop i</code>	(u 8)	(u 32)
<code>alloc i</code>	(u 8)	(u 32)
<code>storel i</code>	(u 8)	(u 32)
<code>storeli i</code>	(u 8)	(u 32)
<code>storeg i</code>	(u 16)	(u 32)
<code>ret i</code>	(u 8)	(u 32)

TAB. 5.2 –. Les formats de base de Machina

d'un petit programme Scheme pour calculer la fonction Fibonacci. On peut y voir quelques caractéristiques du compilateur.

Notez certaines particularités importantes pour la suite. Par exemple, l'étiquetage des données Scheme s'effectue à l'aide d'instructions générales `pushi` et `or`. La ligne 4, de la figure 3, en présente un exemple. Il s'agit, dans ce cas, de l'étiquetage d'une fermeture. De même pour l'élimination d'une étiquette de type, il y a usage de `pushi` et `and`. Un autre exemple est l'implantation des opérateurs relationnels. L'opérateur `##fixnum<` produit *vrai* (constante 26) ou *faux* (constante 10), selon le résultat de la comparaison des deux opérandes du haut de la pile. Mais dans la majeure partie des cas, cette valeur générée doit servir à un branchement. Toutefois, pour implanter les branchements, il y a empilement de la valeur *faux* et vérification par égalité pour déterminer le branchement. Tous ces détails démontrent que rien de particulier n'a été introduit dans la machine virtuelle Machina pour implanter le langage Scheme.

Ce point est très important pour la suite, car si nous introduisons d'emblée des instructions spéciales adaptées à Scheme, notre système n'a plus l'utilité prétendue ; c'est-à-dire l'habileté de découvrir des séquences d'instructions adaptées au langage à implanter et aux programmes à exécuter.

### 5.3 L'édition des liens

L'édition des liens fusionne plusieurs programmes pour n'en former qu'un seul pour l'exécution. Il reçoit en entrée plusieurs programmes de type « code ». Deux opérations majeures sont appliquées :

- la combinaison de plusieurs tables de constantes, ainsi que les tables des variables globales.
- la traduction des instructions en chaîne de bits de façon compacte.

L'opération de combinaison des tables des variables globales est nécessaire pour leur unicité ; clairement, pour une même variable, toutes les parties du programme final ne doivent référencer qu'un seul et même emplacement. Il se produit donc, dans ce cas, une union des tables, c'est-à-dire une élimination des duplicata des variables. La structure de la table des variables globales doit donc garder l'identificateur de ces variables. Une fois le code exécutable produit, cette information n'est plus utile pour l'exécution. Bien entendu, certaines instructions référençant des variables globales doivent être modifiées pour tenir compte de l'union des tables.

Il en est de même pour les constantes : il ne peut y avoir deux constantes identiques dans la table finale lors de l'exécution. Ainsi, il y a union des tables des constantes, et les indices des instructions `push` doivent être modifiés pour tenir compte de cette union.

C'est l'éditeur de liens qui fait la compression du code. C'est donc dire que le dictionnaire de macro-instructions, ainsi que le dictionnaire de formats sont utilisés par ce programme. Les macros sont préalablement appliquées en favorisant les plus longues. Dans une seconde phase, chaque instruction du programme est codée en utilisant son format `supremum` par rapport à l'ensemble des formats disponibles dans le dictionnaire des formats.

### 5.4 La codification binaire des programmes compressés Machina

Les instructions de branchement nécessitent un traitement particulier lors de la codification en binaire des programmes. Rappelons qu'une telle instruction doit avoir en opérande, durant l'exécution, le nombre de bits à franchir pour atteindre l'instruction cible du branchement. Toutefois, le compilateur génère plutôt, comme opérande, le nombre d'instructions à franchir. La difficulté est la suivante : pour les instructions branchant vers l'avant, on ne connaît pas le nombre de bits à franchir avant d'avoir accompli la tâche de coder les instructions en binaire. Ainsi, ne connaissant pas le nombre de bits exact occupé par l'opérande, on ne connaît pas le format le plus compact à utiliser. Bien entendu, il y a une difficulté si plusieurs formats sont disponibles pour le même mnémonique. La difficulté est récursive.

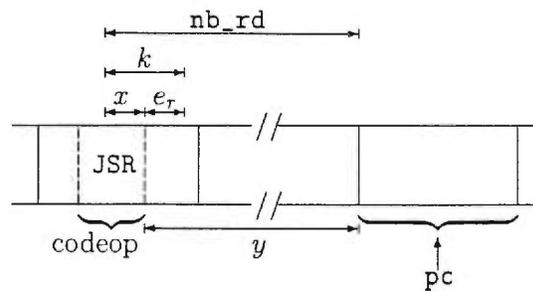


FIG. 5.1 –. Le calcul de l'adresse de retour pour JSR

En effet, il peut y avoir des instructions de branchement dans les instructions à franchir. C'est un problème classique, existant aussi pour les programmes non compressés. Les langages assembleurs ayant des instructions de branchement à plus d'un format ont ce problème particulier.

Nous utilisons la méthode classique habituelle permettant de coder, de la façon la plus compacte possible, chaque instruction de branchement. C'est-à-dire qu'au départ, les instructions de branchement sont codées dans le format le plus large. Une fois le nombre de bits à franchir connu, l'instruction est révisée pour être codée dans un format plus petit. Cette modification réduit le nombre de bits pour cette instruction<sup>1</sup>. Tant qu'il y a une réduction appliquée, les autres instructions de branchement peuvent être affectées. Ainsi, le processus peut être répété plus d'une fois. Il y a arrêt des réductions seulement si aucune instruction de branchement ne peut être réduite.

### 5.5 L'instruction JSR

L'instruction JSR doit empiler l'adresse de retour, mais puisque le décodeur a déjà chargé plus que l'instruction dans le registre de décodage `rd`, le pointeur réel du programme ne pointe pas nécessairement le prochain octet suivant le code opérationnel de JSR. Rappelons que l'instruction suivant une instruction JSR est alignée sur le prochain octet, ainsi c'est vraiment l'adresse de l'octet suivant le code opérationnel de JSR qui doit être empilé.

La solution à ce problème doit considérer les deux types de decodeurs : automate ou canonique. Le code implantant JSR doit être indépendant du décodeur, ainsi un ajustement au `pc` réel doit être effectué avant ce code.

1. La somme de la longueur du code opérationnel et du format ne peut que diminuer, sinon ce format n'a pas d'utilité.

<b>libScheme</b>	Les définitions des fonctions R <sup>4</sup> RS.
<b>fib</b>	Une évaluation récursive de Fibonacci(28).
<b>conform</b>	Des vérifications de type.
<b>earley</b>	Une construction d'analyseurs syntaxique.
<b>tak</b>	Une évaluation de Takeuchi(18,12,4).
<b>qsort</b>	Un tri rapide à la QuickSort de mille entiers.
<b>mm</b>	Mise au carré d'une matrice 100 par 100.
<b>destruct</b>	Une série d'opérations répétitives sur des listes.

TAB. 5.3 –. Les benchmarks utilisés pour l'expérimentation

La figure 5.1 présente la situation générale pour un décodeur automate. Dans le cas de l'automate, le bloc de décodage n'est pas nécessairement aligné au début du code opérationnel de l'instruction. La valeur  $k$  est le nombre de bits de la racine du décodeur, la valeur  $e_r$ , quant à elle, est le nombre de bits n'appartenant pas au code opérationnel de JSR mais faisant partie du bloc de décodage de  $k$  bits. Cette valeur est nécessairement comprise entre  $k - 1$  et 0. La valeur  $x$  est le nombre de bits appartenant au code opérationnel et faisant partie du préfixe du bloc de décodage. Nous avons  $k = x + e_r$ . La variable `nb_rd` contient le nombre de bits dans le registre décodeur `rd` au début du cycle de décodage. La valeur de  $y$  est le nombre de bits présents dans `rd`, suivant le code opérationnel de JSR. Ainsi, `nb_rd` contient la valeur  $x + y$ . Le pointeur de programme `pc` doit être diminué de la valeur de  $\lfloor y/8 \rfloor$ .

Le constructeur d'automate détermine la chaîne de bits suivant l'instruction JSR; c'est l'état résiduel. Cette information est gardée dans l'automate (c'est le terme  $b'$  de la ligne (18) de l'algorithme de la figure 3.3), ainsi la génération du code C connaît la valeur  $e_r$ . La génération du code C produit une instruction virtuelle JSR\_1 $e_r$  pour chaque valeur de  $e_r$ .

Pour ajuster le `pc`, il y a donc, préalablement, la soustraction de la constante  $k - e_r$  à `nb_rd` pour ensuite effectuer l'opération `pc -= nb_rd >> 3`.

Dans le cas du décodeur canonique, la situation est moins compliquée car le nombre de bits à éliminer de `nb_rd` est exactement le nombre de bits du code opérationnel. Puisque la longueur du code opérationnel est une constante, il n'y a alors qu'une implantation de l'instruction JSR pour un tel décodeur.

Mnémonique	Fréq	Mnémonique	Fréq	Mnémonique	Fréq
pushi	20318	pushl	6353	dup	4566
pusha	3520	and	3382	not	3282
jsr	2733	pushg	2429	storea	1938
exg	1728	storel	1632	bf	1057
br	916	push	878	or	822
alloc	798	neq	726	ret	626
pop	432	storeg	401	eq	240
sub	192	storeli	140	add	139
pushli	70	asr	53	gt	46
lt	45	popv	35	lsr	29
writec	24	mul	20	asl	18
pushac	11	div	7	storeac	5
rem	3	stop	1		

TAB. 5.4 –. Les fréquences des mnémoniques de base de Machina pour l'ensemble des benchmarks de la table 5.3

### 5.6 Les programmes benchmarks Scheme

Pour les expériences de compression et de vitesse d'exécution, nous utilisons un ensemble de programmes-pilotes : ce sont les *benchmarks* présentés à la table 5.3. En fait, l'un des benchmarks n'est pas un programme, mais une librairie de fonctions. Elle provient du système de Danny Dubé [Dub96], c'est-à-dire `libScheme`, construit pour l'élaboration d'une implantation de Scheme pour micro-contrôleur. C'est l'ensemble des fonctions et procédures du R<sup>4</sup>RS. Elle fut modifiée pour en améliorer la rapidité au détriment de l'espace. Essentiellement, toutes les fonctions de base (e.g. `+`, `cons`, `car`) existant comme primitives dans notre compilateur ont été directement employées.

Ces benchmarks ont été compilés en des programmes de type « code ». Il en résulte un total de 58194 instructions de base. Trois mnémoniques ne sont pas utilisés : `readc`, `xor` et `jmp`.

### 5.7 Compression des benchmarks Scheme sans macro-instructions

Dans cette section, nous présentons les résultats de compression des benchmarks, sur Machina, sans utiliser de macro-instructions. Il s'agit de compression utilisant les spécialisations de formats et de valeurs ainsi que la codification de Huffman sur les codes opérationnels.

	ret	pushg	push	pushl	pushli	storeli	storel	storeg	pop	alloc
(u 1)		580	16	970			761	7	374	
(u 2)	263	336	14	1051		35	83	7	46	
(u 3)	308	425	26	2370	70	105	655	9		
(u 4)	37	365	91	1373			75	20		778
(u 5)	3	339	104	413			6	48		
(u 6)		258	166	7				99		
(u 7)		99	345					151		

Format	pushi	br	bf	Format	pushi	br	bf
(s 1)	1282			(s 8)	33	55	99
(s 2)	4771	400		(s 9)	18	25	48
(s 3)	3378	46	392	(s 10)	89	35	9
(s 4)	3853	38	15	(s 11)	114	6	4
(s 5)	1945	56	141	(s 12)	128	1	2
(s 6)	3269	119	117	(s 13)	257		
(s 7)	229	98	192	(s 14)	488		

TAB. 5.5 – Les formats et fréquences des instructions de base pour les benchmarks

Les fréquences des mnémoniques sont présentées par la table 5.4. L'instruction la plus fréquente est `pushi`. L'entropie des mnémoniques est environ 3.57. C'est donc dire que la longueur moyenne des codes opérationnels compressés pour tous les benchmarks pourrait être 3.57 bits. Toutefois, nous n'utilisons pas ces fréquences pour construire les codes opérationnels, car les tests de rapidité d'exécution se feront en utilisant de nouveaux formats. Ce premier tableau présente une vue générale de la répartition des instructions des benchmarks.

Notre première expérience consiste à former deux dictionnaires de formats : l'un construit à partir de la librairie `libScheme`, et l'autre par tous les benchmarks. Il n'y a pas, ici, de dictionnaire de macros. Nous combinons ces deux dictionnaires en supposant la librairie entièrement présente dans la machine virtuelle, mais seulement 20% pour la partie benchmarks.

Le deuxième résultat est la fréquence des formats des instructions de base. La table 5.5 donne une liste de ces formats avec leur fréquence d'occurrence. Ce sont les formats possibles lors de la spécialisation des instructions de base. Nous avons appliqué l'algorithme de création de nouvelles instructions en utilisant ce dictionnaire de formats. La table 5.6 présente les nouveaux formats des instructions ainsi que leur code opérationnel.

Mnémonique	Format	Codeop	Mnémonique	Format	Codeop
pushi	(c 8)	0000	dup	( )	0001
pusha	( )	0010	and	( )	0011
not	( )	0100	pushi	(c 7)	0101
jsr	( )	0110	pushi	(c 18)	0111
pushl	(u 4)	10000	pushi	(s 5)	10001
storea	( )	10010	exg	( )	10011
pushi	(c 24)	10100	pushg	(u 3)	10101
pushi	(c 16)	10110	pushi	(s 14)	10111
pushi	(s 1)	110000	pushg	(u 6)	110001
storel	(c 1)	110010	pushl	(c 3)	110011
pushi	(c 10)	110100	pushl	(c 4)	110101
neq	( )	110110	or	( )	1101110
pushl	(c 5)	1101111	alloc	(c 8)	1110000
pushl	(u 5)	1110001	pushi	(c 1)	1110010
bf	(s 10)	1110011	push	(u 6)	1110100
bf	(s 7)	1110101	pushi	(s 6)	11101100
push	(u 7)	11101101	pop	(c 1)	11101110
br	(s 7)	11101111	br	(s 4)	11110000
pushi	(s 7)	11110001	eq	( )	11110010
storel	(c 5)	11110011	ret	(c 3)	11110100
storeg	(u 6)	11110101	pushg	(c 0)	111101100
storel	(c 6)	111101101	ret	(u 4)	111101110
ret	(c 4)	111101111	storel	(u 4)	111110000
pushi	(s 3)	111110001	rem	( )	111110010
sub	( )	111110011	storeg	(u 8)	111110100
br	(s 10)	111110101	storel	(c 4)	111110110
pushg	(u 7)	111110111	add	( )	1111110000
storel	(c 2)	1111110001	push	(u 8)	1111110010
storeac	( )	1111110011	pushac	( )	1111110100
lsr	( )	1111110101	gt	( )	11111101100
push	(c 11)	11111101101	pop	(u 2)	11111101110
asr	( )	11111101111	asl	( )	11111110000
pushli	(c 7)	11111110001	storeli	(c 6)	11111110010
pushli	(c 6)	11111110011	storeli	(c 5)	11111110100
storeli	(c 3)	11111110101	storeli	(c 4)	11111110110
pushi	(s 24)	11111110111	br	(s 24)	111111110000
lt	( )	111111110001	ret	(u 24)	111111110010
stop	( )	111111110011	popv	( )	111111110100
div	( )	111111110101	mul	( )	111111110110
pushl	(u 24)	111111110111	bf	(s 24)	111111111000
pushli	(u 24)	111111111001	push	(u 24)	111111111010
storeli	(u 24)	111111111011	pushg	(u 24)	111111111100
storel	(u 24)	111111111101	pop	(u 24)	111111111110
storeg	(u 24)	111111111111			

TAB. 5.6 – Les codes opérationnels des nouveaux formats et des instructions de base standard

	libScheme	fib	tak	earley	conform	mm	destruct	qsort
Code-octet	32040	169	582	26271	28599	2550	3371	5827
Compressé	39%	36%	40%	41%	39%	42%	38%	52%

TABLE 5.7 – Tailles absolues (code-octet, non compressé) et relatives (compressé) des benchmarks Scheme sans macro-instructions

Les facteurs de compression des benchmarks sont représentés à la table 5.7. Ces facteurs tiennent compte de la codification de Huffman sur les codes opérationnels et la compaction des arguments dans les nouveaux formats. Les facteurs sont très bons car les formats initiaux, bien que conçus au minimum pour permettre d'exécuter le code-octet des benchmarks, sont souvent trop larges pour la plupart des arguments. De plus, il y a peu d'instructions de base, donc peu de codes opérationnels, ce qui désavantage le code-octet. La compression prend avantage à cela.

### 5.7.1 Vitesse d'exécution des benchmarks compressés

Dans cette section, nous présentons les vitesses d'exécution des benchmarks compressés selon les instructions de la table 5.6.

Les tables 5.9 et 5.8 présentent les temps d'exécution des benchmarks utilisant plusieurs décodeurs. Les temps d'exécution des programmes compressés sont comparés aux temps d'exécution des programmes non compressés. Dans tous les cas, les décodeurs utilisent l'accès mémoire forme-c.

Pour tous les décodeurs canoniques, sur Pentium, nous utilisons le compilateur gcc version 2.91.66 avec optimisation -O1. Mais pour les décodeurs automates sur processeur Pentium, nous avons utilisé le compilateur gcc version 2.7.2.3, car la version 2.91.66 ne complète pas la compilation à cause d'un usage excessif de mémoire (plus de 600 Mo). Cela se produit pour les automates  $k_r > 6$ , mais nous utilisons la version 2.7.2.3 pour tous les automates afin de permettre des comparaisons. Ainsi, les temps absolus diffèrent entre les décodeurs automates et les décodeurs canoniques pour le Pentium.

Pour le processeur Sparc, nous utilisons le compilateur gcc version 2.8.1, avec option -O1, même pour les décodeurs automates. Toutefois, pour l'automate  $k_r = 8$ , nous avons dû diminuer son espace code en utilisant plus soigneusement le prédicat *Iz* lors de la construction de l'automate, et cela au détriment de sa performance en vitesse.

Pour les décodeurs canoniques, les meilleures performances se produisent pour les décodeurs  $C_{k_r=10}$  et  $C_{k_r=11}$  sur le Pentium. C'est similaire pour le Sparc, bien que  $C_{k_r=12}$  est

	Temps Non Compressé	Temps relatif						
		$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$	$C_{k_r=11}$	$C_{k_r=12}$
fib	<b>1.76</b>	1.31	1.25	1.22	1.18	1.13	1.17	1.16
tak	<b>1.55</b>	1.48	1.42	1.37	1.34	1.32	1.30	1.33
earley	<b>1.70</b>	1.40	1.33	1.30	1.29	1.29	1.26	1.27
conform	<b>11.70</b>	1.39	1.33	1.25	1.22	1.21	1.22	1.22
mm	<b>3.70</b>	1.45	1.43	1.35	1.34	1.31	1.30	1.30
destruct	<b>0.79</b>	1.31	1.30	1.26	1.19	1.19	1.19	1.21
qsort	<b>1.83</b>	1.33	1.27	1.24	1.22	1.23	1.22	1.24

TAB. 5.8 – Temps d'exécution des benchmarks Scheme compressés, sans macros, avec décodeurs canoniques pour le processeur Pentium

	Temps Non Compressé	Temps relatif						
		$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$	$C_{k_r=11}$	$C_{k_r=12}$
fib	<b>4.74</b>	1.33	1.22	1.16	1.11	1.11	1.11	1.08
tak	<b>4.35</b>	1.45	1.37	1.32	1.25	1.25	1.23	1.21
earley	<b>4.14</b>	1.29	1.20	1.17	1.15	1.14	1.13	1.12
conform	<b>32.84</b>	1.31	1.24	1.20	1.16	1.15	1.17	1.16
mm	<b>9.26</b>	1.48	1.42	1.40	1.34	1.31	1.28	1.27
destruct	<b>2.16</b>	1.34	1.25	1.19	1.13	1.14	1.13	1.13
qsort	<b>5.54</b>	1.17	1.11	1.06	1.03	1.04	1.04	1.03

TAB. 5.9 – Temps d'exécution des benchmarks Scheme compressés, sans macros, avec décodeurs canoniques pour le processeur Sparc

encore plus rapide. Les temps relatifs se situent dans le meilleur cas,  $C_{k_r=10}$ , pour le Pentium, entre 13 et 31%. Ceci est causé par la faible granularité des instructions. Dans le cas du Sparc, le benchmark qsort se démarque par un temps d'exécution rapide.

Les décodeurs automates sont moins rapides pour le Pentium. Toutefois, les décodeurs automates ont de bonnes performances pour le processeur Sparc. La vitesse d'exécution augmente de façon continue, du plus petit au plus gros décodeur. Le décodeur  $A_{k_r=8}$  a des performances comparables aux décodeurs  $C_{k_r=10}$  et  $C_{k_r=11}$ , mais les tailles de ces deux décodeurs sont inférieures au décodeur  $A_{k_r=8}$ .

En conclusion, les benchmarks compressés en n'utilisant que les instructions de base, pour les meilleurs décodeurs, ont des ralentissements variant de 3 à 27%.

	Temps	Temps relatif				
	Absolu	$A_{k_r=4}$	$A_{k_r=5}$	$A_{k_r=6}$	$A_{k_r=7}$	$A_{k_r=8}$
fib	<b>1.63</b>	1.77	1.61	1.48	1.39	1.20
tak	<b>1.41</b>	2.16	1.89	1.75	1.73	1.63
earley	<b>1.25</b>	2.04	1.74	1.72	1.63	1.50
conform	<b>10.86</b>	1.86	1.66	1.56	1.49	1.37
mm	<b>3.40</b>	2.23	2.05	1.82	1.74	1.48
destruct	<b>0.71</b>	1.88	1.66	1.52	1.49	1.30
qsort	<b>1.71</b>	1.78	1.58	1.48	1.38	1.21

TAB. 5.10 – Temps d'exécution des benchmarks Scheme compressés, sans macros, avec décodeurs automatés pour le processeur Pentium

	Temps	Temps relatif				
	Absolu	$A_{k_r=4}$	$A_{k_r=5}$	$A_{k_r=6}$	$A_{k_r=7}$	$A_{k_r=8}$
fib	<b>4.74</b>	1.54	1.30	1.23	1.20	1.07
tak	<b>4.35</b>	1.77	1.45	1.39	1.35	1.28
earley	<b>4.14</b>	1.51	1.36	1.27	1.18	1.16
conform	<b>32.84</b>	1.50	1.34	1.22	1.17	1.14
mm	<b>9.26</b>	1.83	1.51	1.48	1.50	1.33
destruct	<b>2.16</b>	1.53	1.29	1.22	1.10	1.07
qsort	<b>5.54</b>	1.40	1.17	1.10	1.00	1.01

TAB. 5.11 – Temps d'exécution des benchmarks Scheme compressés, sans macro-instructions, avec décodeurs automatés pour le processeur Sparc

### 5.8 Conception d'instructions virtuelles pour Scheme

Cette section pourrait s'intituler "Compression de programmes avec macro-instructions pour Scheme", mais il est préférable de percevoir les macro-instructions comme des instructions virtuelles adaptées au langage Scheme ; car, la forme des macro-instructions générées automatiquement par l'algorithme de création des instructions, dépend fortement du compilateur et du langage Scheme.

La conception des instructions virtuelles pour Scheme consiste à augmenter le jeu d'instructions de base de Machina à partir d'échantillons de programmes typiques. Pour ce faire, nous utilisons la spécialisation d'instructions et les macro-instructions. Cela sera effectué en utilisant l'algorithme général de création d'instructions du chapitre 2.

L'échantillon utilisé est la librairie libScheme sans les autres benchmarks. Nous avons limité la taille de la base de macro-instructions de plusieurs façons : un maximum de trois cents macro-instructions, d'au plus quinze instructions de base, avec un maximum de vingt formats ; de plus, un format, pour une séquence, doit avoir une fréquence d'occurrence supérieure à vingt. Ce contrôle de la taille de la base réduit substantiellement le temps de sa création, ainsi que le temps d'exécution de l'algorithme de création des instructions. La construction de la base de formats s'effectue rapidement et ne demande pas une telle intervention. Le temps total de la création des instructions, incluant les bases, se situe autour de vingt minutes en mode interprété Scheme sur un ordinateur 400 MHz.

Nous ne nous attarderons pas sur le contenu de la base de macro-instructions, mais plutôt sur le dictionnaire final, qui est plus pertinent.

La table 5.12 présente une partie du dictionnaire final de macro-instructions pour la machine virtuelle adaptée à Scheme. Ce dictionnaire a quatre-vingt macro-instructions, dont plusieurs sont très particulières. Les constantes 10, 18, 26, 7, 3, 8, 0, 1 et 2 reviennent fréquemment. Ces constantes sont souvent utilisées par le compilateur Scheme pour faire des opérations d'étiquetage ou pour représenter des objets de base du langage Scheme. En particulier, les constantes 10, 26, 18 représentent les constantes #f, #t et <void> respectivement. La constante 3 est l'étiquette d'un pointeur dont le type complet se trouve en mémoire. La constante 8 est, en fait, la constante 1 décalée de trois bits. C'est la représentation de 1 dans notre implantation de Scheme. Les constantes 0, 1 et 2 sont souvent utilisées dans les opérations sur la pile, ou pour accéder aux descripteurs de vecteurs ou de fermetures.

Voici quelques explications sur la signification des macro-instructions en relation avec le langage Scheme, et son implantation telle qu'effectuée par notre compilateur.

La macro-instruction `m1` a une raison d'être simple : il s'agit de l'appel d'une fonction associée à une fermeture. La constante 7 permet d'éliminer l'étiquette, et la constante 1 permet d'accéder au deuxième mot du descripteur de la fermeture contenant l'adresse du

	Séquence	Format
1	(pushi *) (not) (and) (dup) ... (pushi *) (pusha) (jsr)	(c 7 c 1)
2	(pushi *) (alloc *) (dup) (pushi *)... (exg) (pushi *) (storea) (dup)	(c 8 c 8 c 3 c 0)
3	(pushi *) (not) (and)	(c 7)
4	(bf 2) (pushi *) (br 1) (pushi *)	(c 26 c 10)
5	(pushi *) (pushl *)	(c 18 c 2)
6	(pushl *) (pushi *) (not) (and) ... (pushi *) (pusha) (pushi *)	(u 3 c 7 s 2 s 5)
7	(dup) (pushi *) (exg) (pushi *) ... (storea) (dup)	(c 3 c 0)
8	(pushi *) (alloc *) (dup)	(c 8 c 8)
9	(pushl *) (pushi *) (not) (and) ... (pushi *) (pusha) (pushi *)	(u 3 s 4 s 1 s 2)
10	(pushi *) (exg) (pushi *) (storea)	(c 34 c 1)
11	(pushi *) (storea)	(c 0)
12	(pushl *) (pushi *)	(c 2 c 1)
13	(exg) (pushi *) (storea)	(c 2)
14	(pushl *) (pushl *)	(c 0 u 2)
15	(pushi *) (not) (and) (pushi *) ... (pusha) (pushi *)	(c 7 c 0 c 7)
16	(push *) (exg) (pushi *) (storea)	(u 8 c 1)
17	(pushi *) (pushl *) (pushi *)	(s 3 u 3 c 2)
18	(pushi *) (pushl *) (pushi *)	(s 7 u 2 s 2)
19	(pushi *) (pushl *)	(c 18 c 1)
20	(pushi *) (pushl *)	(c 0 u 3)
21	(pushi *) (pushg *)	(c 18 u 7)
22	(pushi *) (or)	(c 3)
23	(dup) (pushi *)	(c 0)
24	(push *) (exg) (pushi *) (storea)	(u 6 c 1)
25	(pushi *) (pushg *)	(s 3 u 3)
26	(pushi *) (alloc *)	(c 12 c 8)
27	(pushi *) (pushi *)	(c 10) (c 10)
28	(storea) (dup) (push *) (exg)... (pushi *) (storea) (pushi *) ... (or) (storeg *) (pushi *)	(u 6 c 1 c 3 u 8 s 5)

TAB. 5.12 -. Une partie du dictionnaire final de macro-instructions

code de la fonction. À l'aide de cette macro-instruction, cet appel se réduit à un seul code opérationnel.

La macro-instruction **m2** a quatre paramètres, mais ce ne sont que des constantes. Elle alloue deux cellules mémoires dans le monceau, et stocke un descripteur étiqueté (3) dans la première cellule.

La macro-instruction **m4** est un autre cas très fréquent. Elle correspond à la génération de **#t** ou **#f**, selon le résultat de la dernière opération relationnelle; car les opérations de base **eq**, **lt**, **gt**, etc., ne génèrent pas les constantes **#t** et **#f** sur la pile. Ce serait supposer que Machina est construite pour Scheme. Il faut donc générer ces valeurs booléennes explicitement par une séquence d'instructions similaire à **m4**.

La macro-instruction **m6** accède une variable locale, d'un déplacement d'au maximum trois bits (0 à 7), qui est nécessairement un pointeur dans le monceau, qui élimine son étiquette et accède à l'une des cellules située près de ce pointeur. Notez que le type (**s 2**) pour **pushi** permet une valeur de -2 à 1, mais en fait c'est soit 0 ou 1 qui est utilisé.

Il y a de nombreuses macro-instructions courtes utilisant les instructions de base **pushi**, **pushl**, **exg** et **dup**. Par exemple, **m12**, **m14**, **m17**, **m18**, **m19** et **m20**. Ce sont typiquement des séquences apparaissant entre les macro-instructions plus longues.

L'allocation de trois cellules, soit douze octets, dans le monceau est effectuée par la macro-instruction **m26**. Cette allocation se produit pour les fermetures sans variables libres.

La macro-instruction **m28** apparaît fréquemment dans la partie principale des programmes. C'est la suite de l'allocation d'une fermeture où le pointeur de code est stocké dans la deuxième cellule (déplacement 1) et où le pointeur de fermeture est étiqueté de la valeur 3 pour être stocké dans une variable globale. C'est typiquement le code d'initialisation de la fermeture d'une fonction globale Scheme. Cette séquence apparaît plus de deux cents fois dans la partie principale de **libScheme**.

On pourrait s'attendre à voir directement des macro-instructions pour des primitives telles **car** et **cdr**. Celles-ci sont effectivement couvertes par des macros-instructions, mais d'une façon indirecte, et difficile à percevoir à première vue. Par exemple, la primitive **car** est implantée dans le compilateur par la séquence d'instructions **pushi 7**, **not**, **and pushi 0** et **pusha**. La macro-instruction **m6** contient cette séquence d'instructions. Elle peut donc planter une partie des usages de cette primitive. La séquence d'instructions de **cdr** est identique à **car** à l'exception de **pushi 0** qui est **pushi 1** pour **cdr**. La même macro-instruction couvre donc aussi une partie des usages de **cdr**.

La section suivante analyse les performances de compression avec ces macro-instructions.

	Code-octet	Compressé sans Macro-instructions	Compressé avec Macro-instructions
<b>libScheme</b>	32040	39%	23%
<b>fib</b>	169	36%	18%
<b>tak</b>	582	40%	26%
<b>earley</b>	26271	41%	31%
<b>conform</b>	28599	39%	23%
<b>mm</b>	2550	42%	30%
<b>destruct</b>	3371	38%	22%
<b>qsort</b>	5827	52%	57%

TAB. 5.13 – Facteur de compression des benchmarks Scheme

### 5.8.1 Compression des benchmarks Scheme avec macro-instructions

Dans le cas de Machina/Scheme, l'utilisation de macro-instructions augmente substantiellement la compression des programmes. La table 5.13 présente les facteurs de compression des sept benchmarks Scheme. Nous y avons reproduit les tailles absolues des programmes code-octet, ainsi que les facteurs de compression sans macro-instructions présentés dans une section précédente. Rappelons que ces programmes sont auto-suffisants, c'est-à-dire qu'ils peuvent être exécutés sans l'apport d'une librairie.

À l'aide des macro-instructions, la librairie libScheme a atteint la taille de 7427 octets, c'est-à-dire 23% de sa taille initiale en code-octet. Le benchmark fib n'a que 31 octets.

En conclusion, mis à part le cas atypique de `qsort`, le facteur de compression par rapport au code-octet se situe entre 20% et 30%.

### 5.8.2 Comparaison des facteurs de compression avec le système BIT

La section précédente a fait une comparaison des facteurs de compression par rapport aux code-octets des benchmarks ; mais, par rapport à d'autres systèmes Scheme, sont-ils vraiment compacts ?

Pour effectuer une telle comparaison, nous référençons un autre système, développé dans le seul but d'obtenir des programmes Scheme très compacts. Il s'agit du système BIT de Danny Dubé [Dub96], conçu pour exécuter des programmes Scheme sur un micro-contrôleur de petite taille. Quelques mots à propos de ce système.

Il est composé d'une machine virtuelle générale écrite en C, et d'un compilateur de code-octet écrit en Scheme. Le compilateur génère une représentation du code-octet sous la forme d'un vecteur C. Pour produire un programme exécutable, il suffit de compiler le vecteur C

généralisé avec la machine virtuelle générale. La machine virtuelle résultante est particulière, car elle ne peut seulement exécuter le programme Scheme original. Cela remplit les exigences des systèmes embarqués, où la machine virtuelle ne charge pas dynamiquement des programmes.

La machine virtuelle ne contient pas de bibliothèque de fonctions. Il faut donc ajouter, au programme Scheme à compiler, les fonctions R<sup>4</sup>RS qu'il référence. Ce travail est effectué automatiquement par le compilateur BIT. Pour mieux faire les comparaisons d'espace, nous distinguerons deux versions d'un même programme. La première, c'est la version sans l'ajout des fonctions R<sup>4</sup>RS lors de la compilation. La seconde version est le code-octet tel qu'il s'exécute dans la machine virtuelle. C'est la version normale d'un programme généré par le compilateur BIT.

La table 5.14 présente les tailles, en octet, des benchmarks. La taille « normale » signifie que le code résultant est exécutable dans la machine virtuelle sans aucun ajout. La taille « Sans Bibliothèque », signifie qu'il n'y a aucune fonction du R<sup>4</sup>RS qui a été ajoutée au code source.

En comparant les tailles normales entre BIT et Machina, il y a une grande différence pour fib, tak, mm et destruct. Cela tient au fait que, pour Machina, il y a peu de fonctions de la bibliothèque qui sont utilisées. La plus grande différence en faveur de BIT est le benchmark earley. Rappelons que dans la version Machina, les opérations de base (car, cdr, etc.) sont explicitement compilées dans le code objet ; mais pour BIT, il y a un appel à une fonction générale en utilisant un seul octet. Cela augmente la vitesse pour Machina, mais diminue ces performances en compression.

En conclusion, notre approche arrive, pour certains programmes, à des résultats similaires au système BIT. Celui-ci a un ensemble d'instructions et un modèle d'exécution bien adaptés à une représentation compacte des programmes. Il est très difficile de concevoir automatiquement des instructions virtuelles bien adaptées à un langage spécialisé. Notre approche ne prétend pas obtenir de telles instructions mais bien d'améliorer le code objet au niveau le plus bas de la codification.

Pour terminer la comparaison entre Machina/Scheme et BIT, nous présentons les temps d'exécution des benchmarks pour BIT (voir table 5.15). De plus, nous avons jugé utile d'ajouter les temps d'exécution de ces mêmes benchmarks pour Gambit version 3.0, en mode interprété. Les temps de Machina sont ceux des programmes compressés s'exécutant avec macro-instructions pour un décodeur canonique dont la racine a huit bits. En fait, ces comparaisons ne donnent qu'une indication assez générale des performances, car ces systèmes ont des modèles d'exécution très différents. En particulier, Gambit est désavantagé, puisqu'il exécute les benchmarks avec toutes les vérifications de type. Machina/Scheme et BIT ne font pas cela. Dans le cas de earley et qsort, BIT manque de mémoire et ne termine pas les benchmarks. Nous avons pu soustraire les temps de GC de Gambit, mais pas dans le

	BIT		Machina	
	Normale	Sans Librairie	Normale	Sans Librairie
<b>fib</b>	1372	115	31	31
<b>tak</b>	1363	209	152	152
<b>earley</b>	6217	4613	8155	6947
<b>conform</b>	6492	3722	6482	4007
<b>mm</b>	1749	409	762	489
<b>destruct</b>	1894	555	755	497
<b>qsort</b>	4318	2943	4370	3355

TAB. 5.14 –. Tailles des benchmarks Scheme pour BIT et Machina

	BIT	Gambit	Machina
<b>fib</b>	15.05	6.27	1.29
<b>tak</b>	14.93	4.58	1.80
<b>earley</b>	–	1.86	1.48
<b>conform</b>	79.42	6.74	13.03
<b>mm</b>	56.97	12.22	4.55
<b>destruct</b>	4.45	1.69	0.77
<b>qsort</b>	–	3.30	4.47

TAB. 5.15 –. Temps d'exécution en seconde des benchmarks pour BIT, Gambit et Machina sur Pentium

cas de BIT. Ce qui désavantage BIT, car aucun GC n'est effectué par Machina/Scheme. On peut tout de même voir que les temps de Machina/Scheme sont inférieurs aux autres systèmes. Cela confirme que les temps de décodage des instructions virtuelles compressées de Machina/Scheme ne sont pas « noyés » dans un modèle d'exécution inefficace.

### 5.8.3 Vitesse d'exécution des benchmarks avec macro-instructions

L'ajout de macro-instructions augmente la vitesse d'exécution. Nous utilisons les décodeurs canoniques, pour la forme-b et la forme-c. Il s'avère que la forme-b est plus rapide. Ceci tient au fait que la majeure partie des macro-instructions sont codifiées sur plus d'un octet. Les tables 5.16 et 5.17 rapportent les temps pour la forme-c, tandis que les tables 5.18 et 5.19 sont pour la forme-b. Les temps sont relatifs aux temps absolus du décodeur de code-octet. On peut voir que les temps d'exécution sont meilleurs avec les macro-instructions.

	Temps relatif				
	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$
fib	0.88	0.85	0.82	0.85	0.85
tak	1.34	1.30	1.28	1.29	1.29
earley	0.96	0.93	0.90	0.90	0.90
conform	1.12	1.10	1.08	1.06	1.01
mm	1.36	1.31	1.28	1.30	1.29
destruct	1.10	1.05	1.02	1.04	1.04
qsort	0.90	0.89	0.87	0.87	0.87

TAB. 5.16 – Temps relatifs d'exécution des benchmarks Scheme, compressés avec macro-instructions, pour des décodeurs canoniques forme-c sur processeur Pentium

	Temps relatif				
	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$
fib	0.80	0.72	0.71	0.85	0.66
tak	1.07	1.00	1.00	1.11	0.88
earley	1.00	0.94	0.93	0.94	0.82
conform	1.08	1.06	1.01	0.97	0.95
mm	1.24	1.14	1.18	1.11	1.04
destruct	0.94	0.88	0.87	0.85	0.79
qsort	0.73	0.68	0.70	0.67	0.60

TAB. 5.17 – Temps relatifs d'exécution des benchmarks Scheme, compressés avec macro-instructions, pour des décodeurs canoniques forme-c sur processeur Sparc

	Temps relatif				
	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$
fib	0.84	0.79	0.79	0.78	0.76
tak	1.35	1.29	1.28	1.27	1.18
earley	1.25	1.21	1.18	1.18	1.11
conform	1.25	1.22	1.20	1.16	1.09
mm	1.39	1.33	1.32	1.32	1.23
destruct	1.08	1.08	1.08	1.06	0.98
qsort	0.88	0.86	0.85	0.80	0.75

TAB. 5.18 –. Temps relatifs d'exécution des benchmarks Scheme, compressés avec macro-instructions, pour des décodeurs canoniques forme-b sur processeur Pentium

	Temps relatif				
	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$
fib	0.78	0.70	0.70	0.68	0.65
tak	1.04	0.96	0.94	0.95	0.84
earley	0.97	0.92	0.90	0.89	0.79
conform	1.11	1.07	1.02	1.01	0.97
mm	1.20	1.08	1.04	1.04	0.97
destruct	0.92	0.90	0.85	0.83	0.75
qsort	0.72	0.68	0.67	0.66	0.60

TAB. 5.19 –. Temps relatifs d'exécution des benchmarks Scheme, compressés avec macro-instructions, pour des décodeurs canoniques forme-b sur processeur Sparc

## TRAVAUX FUTURS ET CONCLUSION

---

### 6.1 Conclusion

Le problème abordé a été la réduction de l'encombrement de programmes interprétés tout en portant une attention au temps d'exécution. Nous devons aussi tenir compte de la taille de l'interprète. Ce travail a été effectué au niveau de la codification des instructions, ainsi que sur le choix des opérations élémentaires à émuler pour chaque instruction virtuelle.

Nous avons appliqué la combinaison de quatre méthodes afin de compresser des programmes à interpréter : création de macro-instructions, utilisation de formats compacts, codification de Huffman des codes opérationnels et le non alignement des instructions sur une frontière d'octet.

À notre connaissance, aucun travail n'avait abordé le problème de l'interprétation, au niveau logiciel, de programmes utilisant des codes opérationnels compressés. En fait, aucune publication ne traite du problème de décodage, au niveau logiciel, d'instructions virtuelles non alignées sur une frontière d'octet.

Nous avons développé des algorithmes de construction de deux types de décodeurs rapides : automate et canonique. Les décodeurs canoniques se sont avérés les plus pratiques et les plus compacts dans des situations réelles comme la JVM. Ils permettent d'utiliser la codification de Huffman sur les codes opérationnels tout en gardant un temps de décodage raisonnable. Les décodeurs automates augmentent, dans certains cas, la vitesse de décodage ; mais leur plein potentiel est complexe à mettre en oeuvre. À notre avis, une solution complète et satisfaisante pour ces décodeurs, nécessite des modifications radicales du stockage des programmes ; entre autre, la séparation des arguments et des codes opérationnels en deux séquences de bits à décoder séparément.

Nous avons appliqué notre approche à une machine générale, c'est-à-dire la *Machina*, dans le but d'obtenir des programmes *Scheme* compressés exécutables. Les résultats expérimentaux démontrent la capacité de l'approche, à la fois pour la compression et la vitesse d'exécution.

Nous avons fait des expériences dans un environnement complexe, c'est-à-dire la JVM, afin de montrer les performances en facteurs de compression et en temps d'exécution de ces méthodes. Sur plusieurs benchmarks de taille moyenne, nous avons obtenu des pertes de vitesse inférieures à 10%. Dans quelques cas, une légère augmentation de vitesse fut observée. Les facteurs de compression obtenus se comparent favorablement à d'autres méthodes

employées. Pour la JVM, un facteur moyen, pour dix benchmarks, est légèrement inférieur à 60%. Ce facteur se compare favorablement à d'autres méthodes, par exemple à [CSCM98] où des facteurs de 70% à 80% sont obtenus avec des ralentissements similaires.

## 6.2 Travaux futurs

### 6.2.1 Décodeur hybride automate/canonique

Les automates décodeurs ont un potentiel fort intéressant. Ils peuvent décoder plus d'une instruction par cycle. Cet avantage est difficile à obtenir à cause de la taille des automates et du code de l'interprète. Ainsi, il faudrait mettre au point une forme hybride, composée de la technique des décodeurs canoniques et des décodeurs automates dans le but de diminuer cet espace. Il serait alors possible d'utiliser un  $k_r$  plus élevé que la longueur moyenne des codes opérationnels, et ainsi augmenter la fréquence de décodage de multiple instructions par cycle. D'autre part, on perçoit qu'il serait prometteur de séparer les codes opérationnels de leurs arguments afin d'améliorer cette possibilité.

### 6.2.2 Modèles de Markov d'ordre supérieur

Nous n'avons pas analysé l'utilisation de modèles de Markov d'ordre supérieur pour la compression des codes opérationnels. Comme contexte de tels modèles, nous pensons surtout à l'utilisation des codes opérationnels précédents. Comme mentionné par Bennet [Ben87, BS89], l'entropie diminue considérablement en utilisant un tel contexte. En effet, on peut concevoir de nombreux exemples où la probabilité d'occurrence d'un mnémonique est influencée par les mnémoniques le précédant. Le problème majeur demeure le compromis de rapidité d'exécution *versus* la compacité de représentation des états.

### 6.2.3 Application au langage C

Il serait intéressant d'appliquer les méthodes présentées dans cette thèse, à la compilation de programmes écrits en C pour les comparer à des méthodes commerciales récemment développées afin de compresser le code exécutable. Par exemple, IBM a développé CodePack [IBM98, KMH<sup>+</sup>98] pour compresser des programmes exécutables du processeur PowerPC. Une telle comparaison pourrait être effectuée en utilisant un compilateur générant une forme intermédiaire simple (e.g. lcc), et par la conception d'un ensemble d'instructions pour interpréter les opérations utilisées dans cette représentation. Il serait alors possible d'utiliser les outils développés dans cette thèse pour générer, sur chaque programme compilé, une machine différente.



## ANNEXE

## UN EXEMPLE DE PROGRAMME Machina

---

Cette annexe présente un exemple de programme objet produit par le compilateur Scheme utilisé dans cette thèse.

Le programme calculant `fib(25)`, écrit en Scheme, est présenté à la figure 1. Nous utilisons les primitives `##fixnum` plutôt que les opérateurs généraux. La version assembleur, générée par le compilateur, est présentée à la figure 2, et sa version Machina à la figure 3. La partie principale alloue et initialise la fermeture pour `fib` et fait l'appel de `fib(25)`. Il n'y a qu'un segment, celui implantant la fonction `fib` elle-même. Nous avons maintenu une numérotation synchronisée des lignes entre les versions assembleur et Machina.

Chaque segment débute par une identification. Cette identification permet de référer à l'adresse du segment par l'instruction `push` (ligne 4). Cette instruction réfère au bassin de constantes. L'éditeur des liens initialise ces adresses. Dans la version assembleur, les références des variables globales et locales sont faites de façon symbolique (lignes 9, 24). Dans la version Machina, un indice numérique est utilisé. Pour diminuer la tâche de génération d'étiquettes au compilateur, nous utilisons des *if-then-else* (lignes 15, 16, 18) qui sont convertis en branchements explicites dans la version Machina.

Lors d'un appel de fonction, la valeur `void` est empilée, ensuite vient les arguments, le nombre d'arguments et finalement un pointeur sur la fermeture (ligne 7). À l'entrée d'une fonction, des déclarations identifient les objets sur la pile et les symboles pour les référencer (lignes 12 à 14). La directive `rev` (lignes 9, 37) permet d'indiquer à l'assembleur les dépilements implicites. Car il y a, dans l'assembleur traduisant vers les instructions Machina, un mécanisme de vérification de la hauteur de la pile à chaque étiquette, directive *if-then-else* et fin d'un segment. Par exemple, si la pile n'a pas la même hauteur entre le début et la fin d'un segment, l'assembleur produit un message d'avertissement. Ce mécanisme permet de vérifier des erreurs internes dans le compilateur Scheme.

```
1 (define (fib n)
2   (if (##fixnum< n 2)
3       n
4       (##fixnum+ (fib (##fixnum- n 1)) (fib (##fixnum- n 2))))
5   ))
6 (fib 25)
```

FIG. 1 – Le programme `fib` en Scheme

```

1  Partie principale
2  Allocation et création d'une fermeture pour fib
3  (pushi 8) (alloc 8) (dup) (pushi 3) (exg) (pushi 0) (storea) (dup)
4  (push (label 0 0 0)) (exg) (pushi 1) (storea) (pushi 3) (or) (storeg fib)
5  Début de l'appel de fib
6  Le résultat par défaut (void), l'argument, le nombre d'arguments et la fermeture
7  (pushi 18) (pushi 200) (pushi 1) (pushg fib) (pushi 7) (not) (and) (dup)
8  Empile l'adresse de fib et fait l'appel
9  (pushi 1) (pusha) (jsr) (rev 3)

10 Fonction fib
11 (debut-segment (label 0 0 0))
12 (stk-ref-force (loc 0 result)) (stk-ref-force (loc 3 0 n))
13 (stk-ref-force (loc 3 nb-args)) (stk-ref-force (loc 1 ferm))
14 (stk-ref-force (loc 2 adr-ret))
15 (if)
16   (if) Implantation de #fixnum<
17   (pushl (loc 3 0 n)) (pushi 16) (lt) (then) (pushi 26) (else) (pushi 10)
18   (endif)
19   (pushi 10) (neq)
20 (then)(pushl (loc 3 0 n))
21 (else)
22   Appel non terminal sur fib
23   (pushi 18) Résultat par défaut
24   (pushl (loc 3 0 n)) (pushi 8) (sub) Argument n-1
25   (pushi 1) Un argument
26   (pushg fib) (pushi 7) (not) (and) (dup) (pushi 1) (pusha)
27   (jsr) (rev 3)
28   Appel non terminal sur fib
29   (pushi 18) Résultat par défaut
30   (pushl (loc 3 0 n)) (pushi 16) (sub) Argument n-2
31   (pushi 1) Un argument
32   (pushg fib) (pushi 7) (not) (and) (dup) (pushi 1) (pusha)
33   (jsr) (rev 3)
34   (add)
35   (endif)
36 (storel (loc 0 result)) Stocke le résultat
37 (ret 3) (rev 1)
38 (fin-segment (label 0 0 0))

```

FIG. 2 -. Le code assembleur du programme fib

```

1  Partie principale
2  Allocation et création d'une fermeture pour fib
3  (pushi 8) (alloc 8) (dup) (pushi 3) (exg) (pushi 0) (storea) (dup)
4  (push 0) (exg) (pushi 1) (storea) (pushi 3) (or) (storeg 0)
5  Début de l'appel de fib
6  Le résultat par défaut (void), l'argument, le nombre d'arguments et la fermeture
7  (pushi 18) (pushi 200) (pushi 1) (pushg 0) (pushi 7) (not) (and) (dup)
8  Empile l'adresse de fib et fait l'appel
9  (pushi 1) (pusha) (jsr)

10 Fonction fib
17  Implantation de #fixnum<
18  (pushl 3) (pushi 16) (lt) (bf 2) (pushi 26) (br 2) (pushi 10)
19  (pushi 10) (neq)
20  (bf 2) (pushl 3)
21  (br 28)
22  Appel non terminal sur fib
23  (pushi 18)
24  (pushl 4) (pushi 8) (sub)
25  (pushi 1)
26  (pushg 0) (pushi 7) (not) (and) (dup) (pushi 1) (pusha)
27  (jsr)
28  Appel non terminal sur fib
29  (pushi 18)
30  (pushl 5) (pushi 16) (sub)
31  (pushi 1)
32  (pushg 0) (pushi 7) (not) (and) (dup) (pushi 1) (pusha)
33  (jsr)
34  (add)
35  C'est la fin du if
36  (storel 5) Stocke le résultat
37  (ret 3)

```

FIG. 3 -. Le code Machina du programme fib

## RÉFÉRENCES

---

- [ACCP98] Guido Araújo, Paulo Centoducatte, Mario Côrtes, and Ricardo Pannain. Code compression based on operand factorization. In *micro*, pages 194–201, December 1998.
- [ARM95] ARM. *An Introduction to Thumb*. Advanced RISC Machines Ltd., March 1995.
- [BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.
- [Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, June 1973.
- [Ben87] J. P. Bennet. *A Methodology for Automated Design of Computer Instruction Sets*. PhD thesis, University of Cambridge, 1987.
- [Ber78] R. E. Berry. Experience with the Pascal P-compiler. *Software & Practice and Experience*, 8(5):617–627, September/October 1978.
- [BJ86] David H. Bartley and John C. Jensen. The implementation of PC scheme. *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93, 1986.
- [BJS97] Mauricio Breternitz Jr. and Roger Smith. Enhanced compression techniques to simplify program decompression and execution. In *Proc. Int'l Conf. on Computer Design*, pages 170–176, October 1997.
- [BNW98] Martin Benes, Steven M. Nowick, and Andrew Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, September 1998.
- [Bra82] James Brakefield. Just what is an op-code? or a universal computer design. *Computer Architecture News*, 10(4):31–34, 1982.
- [BS89] J. P. Bennet and G. C. Smith. The need for reduced byte stream instruction sets. *The Computer Journal*, 32:370–373, 1989.
- [BWN97] Martin Benes, Andrew Wolfe, and Steven M. Nowick. A high-speed asynchronous decompression circuit for embedded processors. In *Proc. Conf. on Advanced Research in VLSI*, September 1997.
- [BYT99] Java BYTEmark benchmarks: source code and results. <http://www.igd.fhg.de/~zach/benchmarks>, 1999.

- 
- [Chu97] Kuo-Liang Chung. Efficient Huffman decoding. *Information Processing Letters*, 61(2):97–99, February 1997.
- [CM99] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. Conf. on Programming Languages Design and Implementation*, 1999.
- [CSCM98] Lars Raeder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for embedded systems. Technical Report RR-3578, Inria, Institut National de Recherche en Informatique et en Automatique, 1998.
- [Cur98] Stephen M. Curry. Java developer: An introduction to the Java Ring. *Java-World: IDG's magazine for the Java community*, 3(4), April 1998.
- [DEM99] Saumya Debray, William Evans, and Robert Muth. Compiler techniques for code compression. In *Workshop on Compiler Support for System Software*, 1999.
- [Dub96] Danny Dubé. Un système de programmation Scheme pour micro-contrôleur. Master's thesis, Université de Montréal, April 1996.
- [Dub99] Danny Dubé. Communication personnelle, 27 August 1999.
- [Ear82] W. Earle. Compress rom programs with a math-function interpreter. *Electronic design news*, March 1982.
- [EFE<sup>+</sup>97] Jens Ernst, Christopher W. Fraser, William Evans, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 358–365, June 1997.
- [FG71] Caxton C. Foster and Robert H. Gonter. Conditional interpretation of operation codes. *IEEE Transactions on Computers*, 20(1):108–111, January 1971.
- [FK96] Michael Franz and Thomas Kistler. A tree-based alternative to java bytecodes. techreport 96-58, Department of Information and Computer Science, University of California, Irvine, December 1996.
- [FK97] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.
- [FMRW97] Marc Feeley, James S. Miller, Guillermo J. Rozas, and Jason A. Wilson. Compiling higher-order languages into fully tail-recursive portable c. Technical Report 1078, Université de Montréal, DIRO, August 1997.
- [FMW84] Christopher W. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984.

- [FP95] Christopher W. Fraser and Todd A. Proebsting. Custom instruction sets for code compression. <http://research.microsoft.com/~cwfraser/vita.htm#Papers>, 1995.
- [Fra99] Christopher W. Fraser. Automatic inference of models for statistical code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, 1999.
- [GB98] Mark Game and Alan Booker. *CodePack: Code Compression for PowerPC Processors*. International Business Machines (IBM) Corporation, 1998.
- [GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java<sup>TM</sup> Language Specification*. Sun Microsystems, 1.0 edition, August 1996. appeared also as book with same title in Addison-Wesleys 'The Java Series'.
- [Has97] Reza Hashemian. Memory efficient and high-speed search Huffman coding. *IEEE Transactions on Communications*, 43(10):2576–2581, October 1997.
- [HATvdW99] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software - Practice and Experience*, 29(11):1005–1023, September 1999.
- [HC99] R. Nigel Horspool and Jason Corless. Tailored compression of java class files. *Software - Practice and Experience*, 28(12):1253–1268, 1999.
- [Heh76] Eric C. R. Hehner. Computer design to minimize memory requirements. *Computer*, 9(8):65–70, August 1976.
- [Heh77] Eric C. R. Hehner. Information content of programs and operation encoding. *Journal of the ACM*, 24(2):290–297, April 1977.
- [HL90] Daniel S. Hirschberg and Debra A. Lelewer. Efficient decoding of prefix codes. *Computing Practices*, 33(4):449–458, April 1990.
- [Huf52] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. IRE*, volume 40, pages 1098–1101, September 1952.
- [IBM98] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0*. International Business Machines (IBM) Corporation, 1998.
- [KCE98] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised<sup>5</sup> report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.
- [Kis97] K. Kissell. *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

- 
- [Kle97] Shmuel T. Klein. Space- and time-efficient decoding with canonical Huffman trees. In Alberto Apostolico and Jotun Hein, editors, *Combinatorial Pattern Matching, 8th Annual Symposium*, volume 1264 of *Lecture Notes in Computer Science*, pages 65–75, Aarhus, Denmark, 30 June–2 July 1997. Springer.
- [Kli81] Paul Klint. Interpretation techniques. *Software - Practice and Experience*, 11:963–973, 1981.
- [KMH<sup>+</sup>98] T.M. Kemp, R.M. Montoye, J.D. Harper, J.D. Palmer, and D.J. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6), November 1998.
- [Knu99] Donald K. Knuth. The MMIX home page. July 1999.
- [KW94] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proc. Int'l Conf. on Computer Design*, pages 270–277, 1994.
- [KW95] Michael Kozuch and Andrew Wolfe. Performance analysis of the compressed code RISC processor. Technical Report CE-A95-2, Princeton University Computer Engineering, 1995.
- [LBCM97] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. Improving code density using compression techniques. In *Proc. Int'l Symp. on Microarchitecture*, December 1997.
- [LDK99] Stan Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 4(1):12–38, January 1999.
- [Ler90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report RT-0117, Inria, Institut National de Recherche en Informatique et en Automatique, 1990.
- [LH90] L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited huffman codes. *Journal of the ACM*, 37:464–473, July 1990.
- [Lia96] Stan Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, June 1996.
- [LW98] Haris Lekatsas and Wayne Wolf. Code compression for embedded systems. In *Proc. ACM/IEEE Design Automation Conference*, 1998.
- [LY97] Tim Lindholm and Frank Yellin. Inside the Java Virtual Machine. *UNIX review*, 15(1):31, 32, 34–36, 38, 39, January 1997.
- [LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.
- [Mar80] Brian Marks. Compilation to compact code. *IBM Journal of Research and Development*, 24(6):684–691, November 1980.

- 
- [MMBC97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Berkeley, June 16–20 1997. Usenix Association.
- [MT97] Alistair Moffat and Andrew Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, October 1997.
- [MTK95] Alistair Moffat, Andrew Turpin, and Jyrki Katajainen. Space-efficient construction of optimal prefix codes. In *Proc. Data Compression Conference*, pages 192–201, March 1995.
- [Nek98] Yakov Nekritch. On efficient decoding of huffman codes. Technical Report 85190-CS, Department of Computer Science, University of Bonn, May 1998. Wed, 12 Nov 98 14:00:00 GMT.
- [Pit87] Thomas Pittman. Two-level hybrid interpreter. native code execution for combined space-time program efficiency. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, pages 150–152, 1987.
- [PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct-threaded code by selective inlining. *ACM SIGPLAN Notices*, 33(5):291–300, May 1998.
- [Pro95] Todd A. Proebsting. Optimizing a ANSI C interpreter with superoperators. In *Proc. Symp. on Principles of Programming Languages*, pages 322–332, 1995.
- [Pug99] William Pugh. Compressing java class files. In *Proc. Conf. on Programming Languages Design and Implementation*, 1999.
- [Que94] Christian Queinnec. *Les langages Lisp*. InterEditions, 1994.
- [RMH99] Derek Rayside, Evan Mamas, and Erik Hons. Compact java binaries for embedded systems. In *Cascon*, pages 1–14, November 1999.
- [Sal98] David Salomon. *Data compression: the complete reference*. Springer, 1998.
- [Say96] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 1996.
- [Sch94] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and Sons, Inc., New York, NY, USA, 1994.
- [Sie88] Andrzej Sieminski. Fast decoding of the Huffman codes. *Information Processing Letters*, 26(5):237–241, January 1988.
- [SK64] Eugene S. Schwartz and Bruce Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, March 1964.
- [Tan87] Tanaka. Data structure of huffman codes and its application to efficient encoding and decoding. *IEEETIT: IEEE Transactions on Information Theory*, 33, 1987.

- 
- [TM98] Andrew Turpin and Alistair Moffat. Comment on “Efficient Huffman decoding” and “An efficient finite-state machine implementation of Huffman decoders”. *Information Processing Letters*, 68(1):1–2, 15 October 1998.
- [Tur86] N. Turner. The right to assemble. code compression with mini-interpreters. *Dr. Dobbs journal*, pages 110–111, May 1986.
- [Tur95] J. L. Turley. Thumb squeezes ARM code size. *Microprocessor Report*, 9(4), March 1995.
- [WC92] Andrew Wolfe and Alex Chanin. Executing compressed programs on an embedded RISC architecture. In Wen mei Hwu, editor, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 81–91, Portland, OR, December 1992. IEEE Computer Society Press.
- [Weg98] Maarten Wegdam. Compact code through custom instruction sets. Nat. Lab. Unclassified Report UR 822/98, Philips Research Laboratories, Eindhoven, The Netherlands, 1998.
- [Wil72] W. T. Wilner. Burroughs b1700 memory utilization. *AFIPS FJCC*, 41:579–586, 1972.
- [Wor72] D. B. Wortman. *A study of language directed machine design*. PhD thesis, University of Stanford, 1972.
- [Zas95] M. Zastre. Compacting object code via parameterized procedural abstraction. Master’s thesis, University of Victoria, 1995.