UNIVERSITÉ DU QUÉBEC À MONTRÉAL

SYSTÈME DYNAMIQUE D'INCLUSION PARTIELLE DES MÉTHODES DANS L'INTERPRÉTEUR DE LA MACHINE VIRTUELLE JAVA SABLEVM

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR
SÉBASTIEN VÉZINA

JANVIER 2008

Remerciements

La réalisation de ce mémoire n'aurait pas été possible sans l'aide et le soutien de plusieurs personnes. En premier lieu, je voudrais remercier mon directeur de recherche Étienne M. Gagnon pour m'avoir fourni l'inspiration à plusieurs reprises pendant la durée de ma maîtrise. Je tiens à souligner le fait que sans son excellente expertise dans le domaine et sans son encadrement tout ceci n'aurait pas été possible. Son enthousiasme et sa passion m'ont souvent été une grande source de motivation. Je le remercie aussi pour son soutien financier surtout durant la dernière année.

Je tiens aussi à remercier le Conseil de Recherche en Sciences Naturelles et en Génie du Canada (CRSNG) pour les trois bourses d'études de recherche (1er cycle) qu'il m'a octroyées durant mon baccalauréat. Ces expériences de recherche en laboratoire m'ont fourni la motivation nécessaire pour entreprendre des études de deuxième cycle. Je voudrais aussi remercier le Fond Québécois de la Recherche sur Nature et les Technologies (FQRNT) qui ont financé mes études de maîtrise et sans qui mes études graduées auraient été très difficiles. Je souhaite aussi remercier le Laboratoire de Recherche sur les Technologies du Commerce Electronique (LATECE) de l'UQAM qui m'a fourni un lieu de travail et une ambiance propice à la recherche ainsi qu'une partie du matériel nécessaire au bon déroulement de ma maîtrise.

Je remercie aussi mes parents et ma famille pour leur support moral et les encouragements qu'ils m'ont prodigués tout au long de mes études. Je voudrais ajouter un merci spécial à ma tante Pauline pour son soutien financier et sa confiance en moi et un gros merci à ma conjointe Valérie qui m'a aimé, soutenu et encouragé pendant tout ce temps. En terminant, je voudrais aussi remercier mon ami Martin pour tous ces midis où nous avons pris le thé et dîné ensemble, merci pour ton écoute. À tous, merci.

Table des matières

Liste des figures

Liste des tableaux

Lexique

Bassin des constantes : Table des symboles contenus dans le fichier *.class* (*constant pool*) utilisée pour stocker les chaînes de caractères et les opérandes complexes du code cotet.

Bloc d'activation : Un bloc d'activation (*frame*) est une zone de mémoire réservée dans une pile lors de l'exécution d'une méthode. Ce bloc contient les paramètres de la procédure appelée, ses variables locales, sa pile des opérandes ainsi que l'adresse de retour permettant de retrouver le bloc d'activation précédent.

Chargeur de classe : Un chargeur de classe (*class loader*) effectue le chargement dynamique des classes dans la machine virtuelle. Typiquement, les classes sont chargées uniquement sur demande.

Code d'opération : Code qui précise à l'ordinateur la nature de l'opération à effectuer (*opcode*). Le code octet Java est composé d'un peu plus de 200 codes d'opération différents.

Inclusion d'une méthode : Cette opération est mieux connue sous le nom de *method inlining* dans le monde de la compilation. C'est une opération qui consiste à effectuer l'expansion d'une méthode. L'expansion remplace l'appel de la méthode par le corps de celle-ci.

Interpréteur par aiguillage *switch-threaded* : Un interpréteur simple et peu performant. Une boucle permet d'itérer d'une instruction à l'autre pour les exécuter. Pour chaque instruction la structure *switch* est utilisée pour brancher vers la bonne implémentation et accomplir la tâche appropriée.

Interpréteur linéaire direct *(direct-threaded)* : Un interpréteur n'utilisant ni bou-

cle, ni aiguillage pour interpréter une séquence d'instructions. Toutes les instructions possèdent dans le corps de leur implémentation la répartition (`goto *pc++`) vers l'adresse mémoire du corps de la prochaine instruction à interpréter.

Interpréteur linéaire inclusif *(inlined-threaded)* : Un interpréteur qui remplace les instructions comprises dans un même bloc de base par une super-instruction équivalente. Une super-instruction, lorsqu'elle est exécutée, effectue le même travail que les instructions qui la composent. L'avantage d'une super-instruction est d'éliminer les répartitions qui se trouvent habituellement entre les instructions élémentaires qui l'ont formée. La création des super-instructions s'effectue dynamiquement pendant l'interprétation.

JIT : Le terme JIT est souvent utilisé pour faire référence à un compilateur juste-à-temps (*Just-In-Time Compiler*). Ce type de compilateur dynamique est utilisé dans les machines virtuelles pour convertir le code octet à exécuter en code machine pour la plateforme hôte. La compilation d'une méthode se fait habituellement de manière dynamique au moment de son premier appel.

Nettoyeur de mémoire : Le nettoyeur de mémoire (*Garbage Collector*) est une forme automatisée de gestion de la mémoire. Celui-ci tente de récupérer la mémoire utilisée par les objets qui ne sont plus utilisés par l'application en cours.

PC : Abréviation du terme anglais *program counter*. Dans le contexte de l'interprétation du code octet, le `pc` est un pointeur qui indique la position courante dans le tableau de code. L'emplacement mémoire référencé par ce pointeur contient l'instruction à exécuter ou une de ses opérandes. Le `pc` est incrémenté pour avancer d'une instruction à l'autre au cours de l'interprétation du code octet.

Répartition : Opération qui se produit lors de l'interprétation du code octet et qui consiste à rediriger le flot d'exécution vers le prochain code d'opération à exécuter (*dispatch*).

Site d'invocation : Endroit dans un tableau de code où un appel de méthode est effectué par l'entremise d'un des quatre types de *invoke*. Il s'agit de l'emplacement du *invoke* dans le tableau de code. Le *invoke* est une instruction du code octet Java qui permet d'effectuer un appel de méthode.

Super-instruction : Combinaison de plusieurs instructions du code octet Java pour en former une plus grande. L'implémentation d'une super-instruction est la concaténation des implémentations des instructions qui la composent.

Thread : Nous utilisons le terme anglais *thread* pour faire référence à un fil d'exécution dans la machine virtuelle. Nous utilisons aussi ce terme pour référer aux structures et aux informations relatives au *thread* dans la machine virtuelle.

VM : Abréviation du terme anglais *virtual machine*. Dans ce mémoire, l'utilisation de l'abréviation *vm* fait référence à la machine virtuelle Java.

Résumé

La compilation de code source vers du code octet combiné avec l'utilisation d'une machine virtuelle ou d'un interpréteur pour l'exécuter est devenue une pratique courante qui permet de conserver une indépendance face à la plateforme matérielle. Les interpréteurs sont portables et offrent une simplicité de développement qui en font un choix intéressant pour la conception de prototypes de nouveaux langages de programmation. L'optimisation des techniques d'interprétation existantes est un sujet de recherche qui nous intéresse particulièrement. Nous avons voulu, par l'entremise de notre projet de recherche, étudier jusqu'où il est possible de pousser l'optimisation dans un interpréteur.

Après avoir étudié les types d'interpréteurs existants, nous avons constaté que les interpréteurs les plus performants se basent tous sur le même principe : *La réduction du coût associé aux répartitions entre les instructions interprétées.* Ce coût est causé par les instructions de répartitions elles-mêmes, mais surtout par l'augmentation du taux d'erreur qu'elles procurent dans les prédicteurs de branchement qui se trouvent au sein des processeurs modernes. Des mauvaises prédictions de branchements occasionnent des coûts importants sur une architecture pipelinée.

L'interpréteur linéaire inclusif est un des plus performants qui existe. En nous basant sur cet interpréteur, nous avons fait la conception et l'implémentation d'un mécanisme qui lui permet d'augmenter la longueur des ses super-instructions et par le fait même de diminuer le nombre de répartitions pendant l'exécution. Nous avons mis au point un mécanisme dynamique d'inclusion partielle des méthodes dans cet interpréteur. Nous avons aussi conçu un système de profilage qui nous permet de détecter les sites d'invocations chauds et d'y effectuer l'inclusion du chemin le plus fréquenté de la méthode appelée. En brisant ainsi la frontière entre le corps des méthodes, nous parvenons à augmenter la

longueur moyenne des super-instructions. Nous avons surmonté et résolu toutes les difficultés inhérentes à l'implémentation d'un tel système dans une véritable machine virtuelle Java (synchronisation, exceptions, présence d'un nettoyeur de mémoire, présence de sous routines dans le code octet Java).

Nous fournissons une étude empirique de l'impact de notre système sur un interpréteur linéaire inclusif en exécutant des applications Java d'envergure. Nous sommes parvenus, dans tous les cas étudiés, à augmenter la longueur moyenne des super-instructions invoquées et à diminuer le nombre de répartitions pendant l'exécution.

Mots clés : Interpréteur, Inclusion, Inclusion partielle, Profilage, Machine virtuelle, Java, JVM, SableVM.

INTRODUCTION

Contexte et problématique

Au cours de la dernière décennie, le langage de programmation Java a grandi en popularité et est devenu omniprésent. Ce langage, conçu par Sun Microsystems, a été rendu public en 1995. Insatisfaits du langage `C++`, les concepteurs de Java se sont fixé comme objectif de créer un nouveau langage de programmation orienté objet qui serait indépendant de la plateforme du client et qui ne souffrirait pas des lacunes de `C++` au niveau de la sécurité, de la programmation *mutlithread*, de la gestion de la mémoire et de la programmation distribuée.

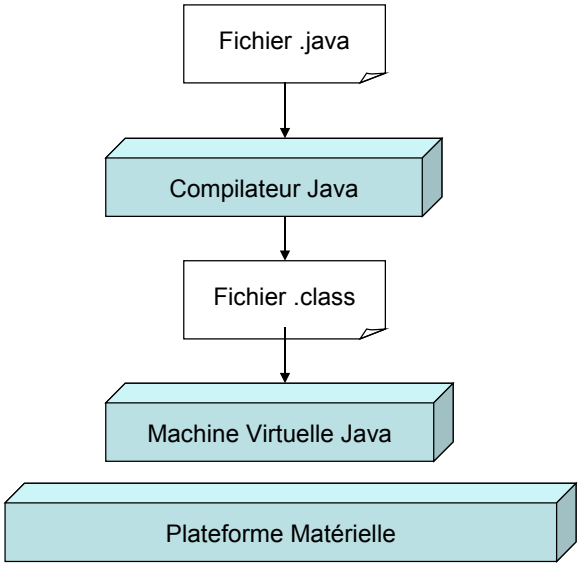

Figure 1 – Paradigme de compilation et d'exécution de Java.

L'indépendance face à la plateforme est obtenue en compilant le programme Java vers un langage intermédiaire (code octet Java) plutôt que vers du code natif non portable (figure 1). Pour exécuter le programme, une machine virtuelle Java [LY99] doit exécuter le code octet. C'est la machine virtuelle et les classes contenues dans les bibliothèques (API) qui fournissent les services relatifs à la gestion automatique de la mémoire, aux *threads*, à la synchronisation, à la sécurité (vérification dynamique des bornes des tableaux, etc.), au chargement dynamique des classes, à la possibilité d'exécuter du code distant et plus encore. Pour un programme donné, le code octet est identique d'une plateforme à l'autre, seul l'implémentation de la machine virtuelle peut différer.

Il existe principalement trois approches utilisées par les machines virtuelles pour exécuter le code octet. Le code octet est soit interprété, soit compilé dynamiquement pendant l'exécution par un compilateur juste-à-temps (*Just-In-Time Compiler*) ou soit compilé et optimisé dynamiquement par un compilateur adaptatif.

Un compilateur juste-à-temps est un compilateur intégré à la machine virtuelle qui compile dynamiquement le code octet en code machine. Au fur et à mesure que les méthodes sont appelées, le code octet de la méthode qui provient du fichier *.class* est compilé en langage machine pour la plateforme utilisée. Cette conversion s'effectue méthode par méthode et habituellement lors du premier appel de la méthode. C'est à ce moment que le compilateur peut choisir d'effectuer des optimisations sur la méthode en question. L'intensité des optimisations appliquées varie selon l'implémentation du compilateur juste-à-temps (par exemple : *Kaffe* [Kaf] et *OpenJIT* [Ope]). Ces optimisations, contrairement aux compilateurs adaptatifs, ne reposent pas sur des informations de profilage recueillies dynamiquement. Il s'agit plutôt d'optimisations classiques comme celles effectuées par des compilateurs statiques comme *gcc*.

Il faut comprendre que la machine virtuelle est un environnement complètement dynamique et que losrqu'elle démarre, elle ne possède ni le code, ni les informations relatives aux méthodes qui seront exécutées. Au fur et à mesure que les classes sont nécessaires, la *vm* effectue le chargement dynamique de ces classes (soit du disque, du réseau ou autre). C'est ce que nous appellons le chargement paresseux des classes. Seulement les classes utilisées sont chargées au besoin. Le compilateur dynamique ou l'interpréteur connait le

code octet d'une méthode seulement lorsque celle-ci est invoquée pour la première fois.

Un compilateur adaptatif est un compilateur qui est intégré dans la machine virtuelle et qui utilise des informations de profilage recueillies dynamiquement par la *vm*. Le profilage est utilisé pour détecter les zones de code qui sont le plus utilisées (zones chaudes). Le compilateur compile et optimise les zones chaudes à différents degrés selon le niveau de chaleur du code. Le compilateur peut recompiler et optimiser plus agressivement les zones de code déjà compilées à mesure que celles-ci se réchauffent. *JikesRVM* [AAB+05] et *HotSpot* [Hot] sont des exemples de machines virtuelles qui utilisent un tel système de compilation adaptatif.

Il est aussi fréquent qu'une machine virtuelle utilise une combinaison des trois approches énumérées plus haut. Une *vm* peut commencer, par défaut, par effectuer l'interprétation du code et à mesure que le système de profilage indique que certaines méthodes sont plus utilisées, elle peut effectuer la compilation de celles-ci.

Dans ce mémoire, nous allons nous intéresser principalement à la première approche, c'est-à-dire à l'interprétation du code octet. Un interpréteur est plus facile à implémenter. Il est portable d'une plateforme à l'autre, mais moins performant. Les compilateurs dynamiques offrent des meilleures performances, mais impliquent une phase de développement plus complexe et plus laborieuse. Ils consomment habituellement plus de mémoire lors de l'exécution et ne sont pas portables. Même si la plupart des machines virtuelles Java modernes implémentent leur compilateur dynamique en Java, la partie qui génère le code natif varie d'une plateforme à l'autre. Un résumé des avantages et des inconvénients des interpréteurs et des compilateurs dynamiques est disponible au tableau 1.

	Interpréteurs	**Compilateurs dynamiques**
Portabilité	oui	non
Simplicité	oui	non
Efficacité	peu	plus
Utilisation de mémoire	peu	plus

Tableau 1 – Interpréteurs vs. Compilateurs dynamiques.

Même en étant moins performants, les interpréteurs gardent un intérêt marqué et sont souvent utilisés pour développer des prototypes rapides pour des nouveaux langages de programmation. Cet intérêt s'explique par la simplicité d'implémentation et surtout par la *portabilité* qu'ils procurent. Plusieurs travaux de recherches portent sur le développement et l'amélioration de nouvelles techniques d'interprétation [PR98, Gag02, GEK01]. La plupart des travaux actuels se basent sur l'élimination du surcoût associé aux instructions exécutées entre les codes d'opération (*opcodes*) à interpréter pour aller chercher un gain de performance. Les instructions utilisées pour faire passer le flot d'exécution d'un code d'opération à l'autre dans l'interpréteur sont appelées les instructions de répartition. Celles-ci sont responsables de coûts importants lors du processus d'interprétation. Ces coûts vont même parfois jusqu'à dépasser les coûts du travail réel effectué par les codes d'opération interprétés. Les répartitions ont pour effet d'augmenter le taux d'erreurs dans les prédicteurs de branchements des processeurs. Ceci engendre des coûts importants dans une architecture pipeline. Ertl et Gregg [EG03] ont démontré que l'élimination des répartitions améliore la performance des prédicteurs de branchements dans les processeurs modernes et c'est ce facteur qui est en majeure partie responsable du gain de performance observé.

L'interpréteur linéaire inclusif (section 1.5.3) est un des plus performants et se base sur cette approche [PR98, Gag02]. Cet interpréteur élimine une partie des coûts associés aux instructions de répartition en construisant des super-instructions. Chacune de ces super-instructions est composée d'une séquence d'instructions élémentaires qui se trouvent dans un même bloc de base. Lorsqu'elle est exécutée, une super-instruction effectue le même travail que les instructions de base qui la composent, mais sans les répartitions qui leur sont habituellement associées. Dans un interpréteur linéaire inclusif, plus les super-instructions sont longues, plus le nombre de répartitions diminue et plus les gains de performance sont substantiels. Un tel interpréteur est déjà en place au sein de la machine virtuelle Java SableVM. Nous proposons dans ce mémoire un mécanisme qui vient augmenter la diminution des répartitions présentement offerte par un interpréteur linéaire inclusif.

Objectifs

L'objectif principal de notre étude est de développer et implanter un système qui permettra de créer des super-instructions encore plus longues dans un interpréteur linéaire inclusif et ainsi diminuer le nombre de répartitions. Afin d'atteindre cet objectif plutôt général, voici comment concrètement nous comptons y arriver :

- Concevoir et implanter dans une véritable machine virtuelle un mécanisme dynamique d'inclusion des méthodes.
- Investiguer divers types d'inclusions pour déterminer celui qui saura nous permettre de construire les plus longues super-instructions possibles afin de pouvoir l'intégrer par la suite dans SableVM.
- Chercher et trouver une manière d'effectuer les optimisations dynamiquement, seulement aux endroits les plus souvent exécutés de manière à pouvoir rentabiliser les coûts de l'optimisation. Enquêter sur les façons de détecter les zones de code souvent visitées.
- Lorsqu'il faut optimiser le code d'une méthode, trouver une façon pour que toutes les invocations courantes de cette méthode puissent bénéficier immédiatement de ces optimisations.
- Mesurer plusieurs métriques pendant l'exécution afin d'évaluer l'influence de notre système sur le mécanisme d'interprétation.
- Étudier jusqu'où il est possible d'aller en termes d'optimisation dans un interpréteur sans dégrader les performances de celui-ci.

Contributions

Dans cette section, nous présentons les contributions apportées par notre étude :

- La conception d'un mécanisme d'inclusion dynamique des méthodes dans un interpréteur. Bien que l'inclusion des méthodes soit un concept qui existe déjà dans le monde de la compilation, nous innovons en l'intégrant au sein d'un interpréteur.
- La conception du système d'inclusion partielle. Notre mécanisme d'inclusion n'inclut que le chemin le plus fréquent dans la méthode appelée et prévoit un mécanisme à

coût presque nul pour continuer l'exécution dans le code original de la méthode lorsque le chemin inclus est invalide.

- Une étude de l'impact de la présence d'un mécanisme d'inclusion partielle des méthodes dans l'exécution d'un interpréteur linéaire inclusif.
- Une augmentation de la longueur moyenne des super-instructions appelées par l'interpréteur impliquant une diminution du nombre de répartitions. Nous réussissons à augmenter, dans tous les tests que nous avons effectués, la longueur des super-instructions appelées d'un pourcentage variant entre 32% et 117%. Nous réussisons à diminuer le nombre de répartions dans tous les tests. Cette diminution varie entre 14% et 51%.
- La conception d'un système de profilage dynamique léger, capable de détecter les sites d'invocations chauds et de repérer le chemin le plus fréquemment emprunté dans le flot d'exécution d'une méthode. Nos tests indiquent qu'en moyenne le chemin sélectionné par notre système de profilage est valide dans 77% des cas. Ce système de profilage nous permet, en moyenne, d'inclure la bonne méthode (virtuelle ou d'interface) dans 98% des cas.
- La conception d'un système de mise à jour des invocations. Lorsque nous modifions dynamiquement le code d'une méthode, il est impératif de mettre à jour toutes les invocations de cette méthode en cours d'exécution dans la machine virtuelle. Notre système de mise à jour tient compte de la présence du nettoyeur de mémoire (*Garbage Collector*), de la présence possible de sous-routines (*JSR/RET*) dans le code octet et du fait que nous sommes dans un environnement *multithread* (synchronisation).
- L'implémentation du système de profilage et du mécanisme d'inclusion partielle dans une véritable machine virtuelle Java (SableVM), permettant ainsi de tester le système sur des applications réelles et d'envergure.

Structure du mémoire

Le reste de ce mémoire est structuré de la façon suivante. Dans le chapitre 2, nous présentons les connaissances de base qui sont nécessaires à une bonne compréhension de ce mémoire. Nous présentons la machine virtuelle SableVM ainsi que le fonctionnement du

code octet Java. Nous présentons les trois types d'interprétation disponibles dans SableVM ainsi que les opérations qui doivent être effectuées sur le code octet Java afin de permettre son interprétation. Dans le chapitre 3, nous présentons sommairement notre système d'inclusion partielle. L'objectif est de donner une vue de haut niveau du mécanisme avant de se diriger vers une étude plus approfondie de celui-ci. Le chapitre 4 se consacre entièrement à l'explication du système de profilage des méthodes. Nous expliquons comment s'initie et s'arrête le profilage d'une méthode, comment s'effectue la construction du code profilant, comment sont gérées les exceptions lors de l'exécution du code profilant, comment nous effectuons la modification dynamique du code d'une méthode et comment nous mettons à jour toutes les invocations de la méthode pour qu'elles puissent bénéficier des modifications effectuées. Finalement, nous discutons de l'aspect synchronisation du système et des limites de celui-ci. Le chapitre 5 se consacre entièrement au mécanisme d'inclusion. Nous exposons comment se fait la construction du code à inclure et comment nous reconstruisons le code et la table des exceptions de la méthode appelante. Nous exposons les principales difficultés rencontrées afin de mettre à jour les invocations de la méthode optimisée. Nous terminons le chapitre par une discussion sur les contraintes d'implémentation ainsi que les limites du système. Dans le chapitre 6, nous effectuons une étude empirique de l'impact de notre système sur l'interpréteur linéaire inclusif de SableVM. Nous illustrons le potentiel de notre système à l'aide d'un mini programme Java écrit spécialement pour bénéficier des forces de notre système. Ensuite, nous donnons et discutons des résultats obtenus avec *SableCC*, *Soot* et la suite de tests *SpecJVM98*. Dans le chapitre 7, nous présentons les travaux reliés à l'inclusion partielle et à l'optimisation des interpréteurs. Finalement, le chapitre 8 présente les travaux futurs et tire les conclusions de nos travaux de recherche.

Chapitre 1

Notions Préliminaires

Ce chapitre vise à introduire les notions qui sont préalables à une bonne compréhension des chapitres subséquents. Nous commençons par présenter SableVM qui a servi de cadre pour cette étude. Nous exposons le contexte d'exécution du code octet Java ainsi que son fonctionnement. Nous portons une attention particulière aux différents types d'invocations de méthodes dans le code octet et nous expliquons leur fonctionnement. Une section subséquente est consacrée à la préparation des méthodes. Ensuite, nous abordons le concept de séquence de préparation et nous expliquons ce qui justifie son utilisation dans la machine virtuelle SableVM. Finalement, nous présentons les interpréteurs par aiguillage, linéaire direct et linéaire inclusif.

1.1 La machine virtuelle Java SableVM

SableVM [GH01, Sabb] est une machine virtuelle Java libre mettant en oeuvre la spécification de la machine virtuelle Java [LY99] établie par *Sun Microsystems*. Elle a été développée en langage C et respecte la norme ANSI, minimisant ainsi le code dépendant de la plateforme. Cette machine virtuelle possède donc l'avantage d'être extrêmement facile à porter sur des nouvelles architectures. SableVM comprend trois différentes méthodes d'interprétation du code octet (par aiguillage, linéaire direct et linéaire inclusif). Les trois modes sont expliqués en détail dans la section 1.5. Le moteur d'exécution du code Java de SableVM est donc un interpréteur et il implémente les plus récentes découvertes en terme

d'interprétation de code.

La spécification de la machine virtuelle Java [LY99] consiste en une description fonctionnelle. Elle ne fait que décrire la manière dont les différents sous-systèmes doivent se comporter, mais ne spécifie pas quelles structures de données utiliser, ni comment implanter les fonctionnalités exigées. C'est pourquoi nous allons exposer, tout au long de cette étude, le fonctionnement interne de certaines parties de SableVM afin de permettre la compréhension de notre système d'inclusion partielle des méthodes.

1.2 Code octet Java et contexte d'exécution

Un compilateur Java prend en entrée du code Java et produit un fichier *.class* pour chaque classe qu'il compile. La spécification du fichier *.class* déborde largement du cadre de ce mémoire, mais elle est toutefois disponible dans le chapitre quatre de la spécification de la machine virtuelle Java [LY99]. Il est important de savoir que dans le fichier *.class*, le code de chaque méthode est exprimé sous forme de code octet. Il porte ce nom puisqu'il est constitué d'un tableau d'octet. Chaque octet représente soit le code d'une instruction (entier) ou une opérande de l'instruction qui la précède. Le jeu d'instructions constituant le code octet Java est constitué d'un peu plus de 200 codes d'opération *(opcodes)*. Chaque instruction dans le tableau de code octet est constituée d'un code d'opération suivi de zéro ou plusieurs opérandes. La description de chaque instruction formant le code octet Java est contenue dans le chapitre six de la spécification de la machine virtuelle Java [LY99].

Nous exposerons par quelques exemples, dans les sections qui suivent, les instructions les plus courantes et les plus utiles pour la compréhension de ce mémoire. Nous fournissons aussi en annexe E, à titre de référence et pour une meilleure compréhension des exemples de ce mémoire, la liste complète de chaque code d'opération du code octet Java contenu dans la spécification. Dans cette annexe, chaque code d'opération est accompagné d'une brève description.

1.2.1 Contexte d'exécution

Pour comprendre toutes les subtilités du code octet, il est essentiel de comprendre la composition de l'environnement d'exécution du code octet dans la machine virtuelle Java. Chaque *thread* possède une pile qui stocke les blocs d'activations *(frames)* des méthodes qu'il appelle. Chaque fois qu'une méthode est invoquée, un bloc d'activation est créé et ajouté au sommet de la pile du *thread* courant. Un bloc d'activation est composé d'un tableau de variables locales, de la pile des opérandes, du `pc` (adresse) de retour, des informations permettant de retrouver le bloc d'activation précédent et d'un pointeur vers une structure *(method_info)* contenant les informations de la méthode relative à l'invocation. Le `pc` de retour permet de savoir où doit reprendre le flot d'exécution suite à l'invocation d'une méthode par la méthode courante. La figure 1.1 montre un bloc d'activation pour une invocation donnée.

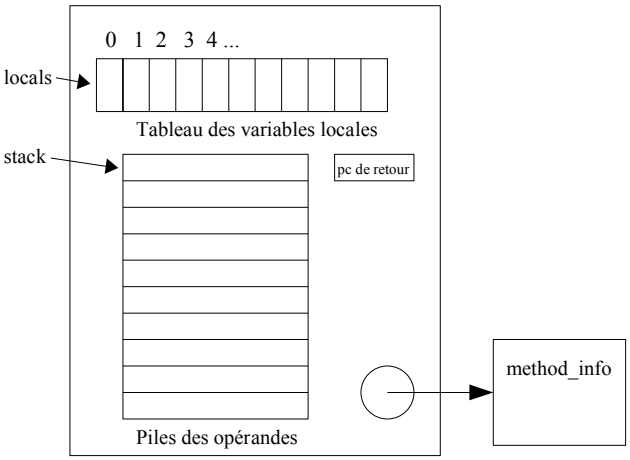

Figure 1.1 – Bloc d'activation.

Chaque bloc d'activation possède son propre tableau des variables locales. Les variables dans la méthode sont identifiées par leur index dans le tableau des variables locales. Chaque variable occupe un seul espace dans ce tableau, à l'exception des variables de type *long* et *double* qui en occupent deux consécutifs. Si la méthode est un constructeur ou une méthode d'instance, la variable à l'index zéro de ce tableau est la référence vers l'objet sur lequel elle a été invoquée (le *this*). Les variables qui suivent cette référence dans le tableau

sont les variables paramètres de cette méthode. Toutes les autres variables utilisées par la méthode seront stockées à la suite des paramètres dans le tableau des variables locales.

Chaque bloc d'activation possède sa propre pile des opérandes. Celle-ci est utilisée pour stocker les opérandes et les résultats des opérations effectués par la méthode. Le code octet Java utilise une pile, plutôt que des registres, pour effectuer ses opérations. Il est important de ne pas confondre la pile des opérandes pour une méthode donnée contenue dans un bloc d'activation et la pile des blocs d'activation pour un *thread* donné.

Le contexte d'exécution d'une méthode Java comprend aussi un tableau de code et le compteur ordinal (le `pc`). Le `pc` indique la position courante dans le tableau code. La pile des opérandes et le tableau des variables locales sont manipulés par l'entremise des pointeurs *stack* et *locals*.

1.2.2 Introduction au code octet Java

Bassin des constantes

Avant de présenter l'exemple qui suit, il est important de mentionner que le format de fichier *.class* est assez complexe. Le fichier comprend, entre autres, un bassin des constantes (*constant pool*) qui est utilisé pour stocker les opérandes complexes du code octet comme les noms de classes et les chaînes de caractères. Le bassin des constantes est une table des symboles de la classe. Le jeu d'instructions du code octet ne dépend pas de la disposition des classes, des interfaces, des objets et des tableaux disponibles seulement à l'exécution, mais plutôt de références symboliques vers le bassin des constantes. Certaines instructions possèdent comme opérande un index vers le bassin des constantes. Souvent, l'index en question utilise 16 bits, donc deux emplacements consécutifs dans le code octet. Pour obtenir l'index en question, une opération sur les bits doit être effectuée `(indexoctet1 << 8) | indexoctet2`.

Exemple

La figure 1.2 illustre le code octet Java généré par le compilateur Jikes[1] [AAB+05] pour la méthode `methode1`, laquelle est une méthode d'instance puisqu'elle ne possède pas le mot clé *static* dans sa signature. Dans une méthode d'instance, la variable locale 0 est utilisée pour la référence de l'objet sur lequel est invoquée la méthode (*this*). Dans le tableau des variables locales, la variable locale 1 correspond au paramètre *s1*, la variable locale 2 correspond à la variable *result* et la variable locale 3 correspond à la variable *s2*.

Code Java	**Code octet**

`int methode1(String s1)`
`{`
`  int result;`
`  String s2 = new String("ab");`
 	
`  if(s1.equals(s2)) {`
`  result = Integer.MAX_VALUE;`
`  }`
`  else {`
`  result = 0;`
`  }`
 	
`  return result;`
`}`
	int methode1(*String*);
	Code:
	00: NEW
	01: #12; *// class String*
	03: DUP
	04: LDC
	05: #14; *// String* ab
	06: INVOKESPECIAL
	07: #18; *// Methode String.<init>*
	09: ASTORE_3
	10: ALOAD_1
	11: ALOAD_3
	12: INVOKEVIRTUAL
	13: #22; *// Methode String.equals()*
	15: IFEQ
	16: 25
	18: GETSTATIC
	19: #28; *// Champ Exemple.value*
	21: ISTORE_2
	22: GOTO
	23: 27
	25: ICONST_0
	26: ISTORE_2
	27: ILOAD_2
	28: IRETURN

Figure 1.2 – Exemple de code octet Java.

Nous allons commenter, une ligne à la fois, le code octet pour expliquer son fonctionnement. Au commencement, la pile des opérandes de la méthode `methode1` est vide.

- Ligne 0 : Cette opération crée une instance de la classe *String* et ajoute sur la pile la référence vers l'objet créé.
- Ligne 1 et 2 : Les octets 1 et 2 sont les paramètres du `NEW` de la ligne 0. Ces deux

[1] Jikes est un compilateur Java open source. Originalement développé par IBM, il fut rapidement transformé en projet open source par une communauté active de développeurs. `http://jikes.sourceforge.net`.

octets contiennent l'index vers le bassin des constantes où se trouve la chaîne de caractères qui indique le type de l'instance à créer. L'index doit être constitué à l'aide de l'opération suivante `(indexoctet1 << 8) | indexoctet2`. Dans ce cas l'index est 12.

- Ligne 3 : Cette instruction va dupliquer l'opérande qui est au sommet de la pile. Nous obtenons donc, à la suite de cette instruction, deux références vers l'instance de la classe *String* créée précédemment.
- Ligne 4 : Cette instruction va aller chercher la valeur qui se trouve à l'index 14 du bassin des constantes et la mettre sur la pile. Ici, il s'agit de la chaîne `"ab"`.
- Ligne 5 : L'index utilisé par le `LDC` pour accéder au bassin des constantes. Dans ce cas, l'index est 14.
- Ligne 6 : Invocation de la méthode qui est identifiée par la chaîne de caractères qui se trouve à l'index 18 du bassin des constantes (*String.<init>*). Il s'agit d'un constructeur de la classe *String*. Cette instruction enlève de la pile la chaîne `"ab"` ainsi qu'une des deux références vers l'objet créé à la ligne 0. La méthode appelée ne retourne rien, car il s'agit d'un constructeur.
- Ligne 7 et 8 : L'index 18 qui est utilisé par le `INVOKESPECIAL`.
- Ligne 9 : Stocke la référence qui reste sur la pile, à l'index 3 du tableau des variables locales.
- Ligne 10 : Ajoute sur la pile la référence qui se trouve à l'index 1 du tableau des variables locales.
- Ligne 11 : Ajoute sur la pile la référence qui se trouve à l'index 3 du tableau des variables locales.
- Ligne 12 : Invocation de la méthode qui est identifiée par la chaîne de caractères qui se trouve à l'index 22 du bassin des constantes (*String.equals()*). Cette méthode enlève de la pile les deux références qui s'y trouvent et y met la valeur de retour de la méthode appelée.
- Ligne 13 et 14 : L'index 22 qui est utilisé par le `INVOKEVIRTUAL`.
- Ligne 15 : Si la valeur sur la pile est égale à zéro, cette instruction fait sauter le flot d'exécution à la ligne 25.

- Ligne 16 et 17 : Paramètre du `IFEQ` qui indique la destination du saut.
- Ligne 18 : Mets sur la pile la valeur du champ statique représenté par la chaîne de caractères situé à l'index 28 du bassin des constantes. Il s'agit du champ *value* de la classe *Exemple*.
- Ligne 19 et 20 : Stocke l'index 28 qui est utilisé par le `GETSTATIC`.
- Ligne 21 : Stocke l'entier qui est sur la pile dans le tableau des variables locales à l'index 2.
- Ligne 22 : Saute à la ligne 25.
- Ligne 23 et 24 : La destination du `GOTO`.
- Ligne 25 : Charge la constante entière `1` sur la pile.
- Ligne 26 : Stocke l'entier qui est sur la pile dans le tableau des variables locales à l'index 2.
- Ligne 27 : Charge l'entier qui se trouve dans le tableau des variables locales à l'index 2 sur la pile.
- Ligne 28 : Le `IRETURN` est utilisé pour quitter la méthode et retourner un entier à la méthode appelante. Ce code d'opération détruit le bloc d'activation courant et met sur la pile des opérandes de la méthode appelante l'entier de retour. Il va ajuster le `pc` pour que le flot d'exécution poursuive son cours au bon endroit dans la méthode appelante. Il modifie aussi les pointeurs *stack* et *locals* pour qu'ils pointent vers les bonnes structures dans la méthode appelante.

Dans le code octet, les codes d'opération sont typés. Dans l'exemple de la figure 1.2, lorsque nous manipulons des entiers, les codes d'opération sont préfixés d'un `i` (lignes 18, 19, 23, 25, 25, 26). Lorsque nous manipulons des références, les codes d'opération sont préfixés d'un `A` (lignes 9, 10, 11). La valeur du préfixe de l'instruction indique le type de la valeur manipulé. Les préfixes possible sont `B` pour *byte* ou *boolean*, `C` pour *char*, `S` pour *short*, `L` pour *long*, `I` pour *integer*, `F` pour *float*, `D` pour *double* et finalement `A` pour *reference*. Nous avons produit, dans l'annexe E, une description réduite de chaque instruction constituant le code octet Java. Une description sommaire de l'annexe permet de constater que la grande majorité des instructions sont préfixées d'une de ces lettres.

Les codes d'opération de type `LOAD` (codes 21 à 53 de l'annexe E) servent à prendre

une valeur dans le tableau des variables locales et à la mettre sur le haut de la pile des opérandes. Les codes d'opération de type `STORE` (codes 54 à 86 de l'annexe E) servent à prendre la valeur sur le haut de la pile et à la mettre dans le tableau des variables locales. Les codes d'opération `CONST_<i>` (codes 1 à 15 de l'annexe E) servent à mettre sur le haut de la pile une constante de valeur `i`. Ces trois types d'opérations sont de loin les plus fréquemment utilisés.

1.2.3 Les quatre types d'invocations

Le jeu d'instructions du code octet Java possède quatre codes d'opération différents pour gérer les appels de méthodes. Ces codes d'opération sont `INVOKEVIRTUAL`, `INVOKESTATIC`, `INVOKESPECIAL` et `INVOKEINTERFACE`.

Le `INVOKEVIRTUAL` est utilisé pour les invocations des méthodes d'instances. Pour ce type d'appel, la pile des opérandes doit préalablement contenir la référence vers l'objet courant ainsi que les valeurs passées en paramètres à la méthode.

Avant d'exécuter une méthode, nous devons en faire la résolution. La résolution consiste à déterminer dynamiquement des valeurs concrètes, à partir de valeurs symboliques se trouvant dans le bassin des constantes. Dans le cas d'une méthode, il s'agit de passer de la chaîne de caractères qui la représente à une représentation plus concrète dans la *vm*. La résolution d'une méthode provoque le chargement de la classe qui la contient ainsi que la création des structures de données associées à cette classe et à la méthode. Une explication plus exhaustive sur le sujet est contenue dans le chapitre cinq de la spécification de la machine virtuelle Java [LY99]. La résolution de la méthode est la première chose qui est faite par un `INVOKE`.

Une fois la résolution terminée, le `INVOKEVIRTUAL` doit rechercher la bonne méthode à invoquer. Cette méthode est déterminée par le type de l'objet courant. Ensuite, il faut effectuer la préparation de la méthode qui consiste à construire le tableau de code réel qui sera exécuté par SableVM. Nous verrons plus en détail, à la section 1.3, en quoi consiste la préparation des méthodes. S'il s'agit d'une méthode synchronisée (mot clé *synchronized* en Java), le verrou pour l'objet courant doit être acquis.

Par la suite, le `INVOKEVIRTUAL` va créer un bloc d'activation pour cette méthode et

l'ajouter sur la pile du *thread* courant. Le tableau des variables locales ainsi que la pile des opérandes sont ensuite initialisés. Les paramètres qui étaient sur la pile des opérandes de la méthode appelante sont ensuite copiés au bon endroit dans le tableau des variables locales à l'intérieur du bloc d'activation nouvellement créé. Une fois que le bloc d'activation a été créé et initialisé, le `pc` est ajusté vers la première instruction du tableau de code de la méthode appelée et ceci permet à l'interpréteur de continuer son travail au début du code de la méthode invoquée.

Les autres types de `INVOKE` font essentiellement le même travail que le `INVOKEVIRTUAL` sauf qu'ils sont adaptés à certains types précis de méthodes. Le `INVOKEINTERACE` est utilisé pour invoquer des méthodes d'interfaces. Le `INVOKESPECIAL` est utilisé s'il s'agit d'une méthode d'une super classe, d'une méthode privée ou d'un constructeur (méthode d'initialisation d'instance). Le `INVOKESTATIC` est utilisé pour faire l'invocation des méthodes de classes statiques. Ce dernier est le seul type de `INVOKE` qui ne doit pas contenir de référence vers l'objet courant dans la pile des opérandes de la méthode appelante avant son utilisation. Pour ce type de `INVOKE`, si la méthode est synchronisée, on doit acquérir le verrou pour la classe, au lieu de celui de l'objet courant.

1.3 Préparation des méthodes

Il est important de comprendre que le code octet tel que présenté jusqu'à maintenant est celui que nous retrouvons dans les fichiers *.class* et que celui-ci n'est pas interprété tel quel par SableVM.

Avant l'exécution d'une méthode, SableVM effectue la préparation de celle-ci. La préparation consiste principalement à créer le tableau de code qui sera véritablement exécuté par l'interpréteur. Au cours de ce processus, il faut effectuer un reformatage du code qui va permettre une exécution plus rapide. Les sous-sections qui suivent exposent les modifications qui sont effectuées au cours de la conversion du tableau de code, ainsi que les raisons qui motivent ces altérations.

1.3.1 Alignement mémoire et taille des éléments du tableau de code

L'objectif de cette modification est d'effectuer l'alignement mémoire du tableau de code. La spécification du fichier *.class* n'impose pas d'alignement mémoire, car cette caractéristique dépend de la plateforme. Comme les architectures modernes sont optimisées pour la manipulation de mots machine, il convient de modifier le tableau de code. Le but est que le nouveau tableau de code ne soit plus composé d'éléments de taille égale à 1 octet (8 bits), mais plutôt d'éléments de taille équivalente à un mot mémoire sur la plateforme hôte (souvent 32 ou 64 bits). L'alignement élimine ainsi le surcoût associé à la manipulation de valeurs de 8 bits. Ceci facilite la manipulation du tableau de code et l'incrémentation du `pc` entre les différents éléments du tableau. Il n'est plus nécessaire d'utiliser plusieurs emplacements pour stocker des valeurs de plus de 8 bits comme c'était le cas dans le code octet original.

1.3.2 Conversion des adresses relatives en adresses absolues

Les sauts et les branchements conditionnels utilisent normalement des adresses relatives. Lors de la préparation du code SableVM, ces adresses sont converties en adresses absolues de manière à accélérer les sauts lors de l'exécution.

1.3.3 Élimination des références vers le bassin des constantes

SableVM a modifié certains codes d'opération pour qu'ils n'utilisent plus comme opérandes des références vers le bassin des constantes, mais bien les valeurs elles-mêmes. En éliminant les références vers le bassin des constantes, la *vm* pourra interpréter le code plus rapidement, puisqu'elle élimine ainsi certaines indirections.

Par exemple, l'instruction `LDC` utilise normalement comme opérande l'index vers le bassin des constantes où est stockée une valeur *int* ou *float* à mettre sur le haut de la pile. SableVM replacera, au moment de la préparation de la méthode, ce code d'opération par un `LDC_INTEGER` ou `LDC_FLOAT`, selon le cas. L'opérande de ce nouveau code d'opération sera la valeur elle-même à mettre sur le haut de la pile. SableVM utilise aussi cette astuce pour les `INVOKE` présentés dans la section 1.2.3. Par exemple, au lieu d'utiliser comme opérande, l'index dans le bassin des constantes du nom de la méthode à invoquer, le

`INVOKESTATIC` possède directement comme opérande, un pointeur vers le *method_info* de la méthode. Le *method_info* est la structure qui contient toutes les informations relatives à une méthode. Ceci viendra accélérer l'invocation de la méthode lors de l'exécution du `INVOKE`. L'interpréteur n'a plus besoin de fouiller dans le bassin des constantes, car le travail a été préalablement effectué lors de la préparation de la méthode. Le travail est donc effectué une seule fois lors du premier appel de la méthode, plutôt qu'à chaque passage sur le code d'opération en question.

1.3.4 Typage additionnel des instructions

L'exemple précédent (avec le `LDC`) de la section 1.3.3, nous donne un exemple du typage additionnel que fait SableVM avec certaines familles de codes d'opération. L'annexe A liste les codes d'opération qui ont été modifés par SableVM afin de fournir le typage additionnel. Prenons l'exemple du `NEWARRAY` tiré de l'annexe A. Le code d'opération `NEWARRAY` est utilisé pour construire un objet de type tableau unidimensionnel. Grâce aux `NEWARRAY<X>` typés, la nécessité de tester, dans l'implantation du code d'opération, le type du tableau à créer est éliminée. Ceci entraîne une exécution plus rapide, car certains tests sont éliminés lors de l'interprération. SableVM effectue donc, au moment de la préparation d'une méthode, la conversion des codes d'opération normaux en codes d'opération typés (voir annexe A).

1.3.5 Ajout des informations nécessaires au nettoyeur de mémoire

SableVM ajoute à même le tableau de code, comme opérandes de certaines instructions clés, des informations *(stack_gc_maps)* utilisées par le nettoyeur de mémoire. Un *stack_gc_map* est un champ de bits qui sert à identifier, pour un point d'exécution donné dans le code, quels sont les éléments dans la pile des opérandes qui sont des références. Ces informations sont utiles pour identifier les objets qui sont toujours vivants dans le tas *(heap)* et ainsi permettre la libération de l'espace mémoire utilisé par ceux qui ne le sont plus. Un objet est considéré vivant si on trouve au moins une référence qui s'y réfère. Nous n'élaborerons pas davantage sur ce point. Une explication plus complète sur le sujet est disponible dans le chapitre six de la thèse de Étienne M. Gagnon [Gag02].

1.3.6 Codes d'opération à deux versions

Chargement paresseux et dynamique des classes

Avec Java, les classes sont chargées dynamiquement. La spécification de la machine virtuelle [LY99] permet à la *vm* d'effectuer le chargement des classes, soit de manière paresseuse, soit de manière vorace. Malheureusement, cette flexibilité ne s'applique pas à l'initialisation des classes[2]. L'initialisation des classes doit se faire à des endroits spécifiques et déterminés comme la première invocation d'une méthode statique ou le premier accès à un de ses champs statiques.

Dans SableVM, les classes sont chargées de manière paresseuse. Le chargement d'une classe est effectué seulement lorsque celle-ci est requise. Par exemple, lors du passage sur un `GETSTATIC` dans le tableau de code, il faut vérifier si la classe contenant le champ statique en question a été chargée, créée et initialisée avant de pouvoir effectuer l'opération requise. Si jamais la classe n'a pas été chargée par le chargeur de classe, il faut la charger et l'initialiser avant de pouvoir obtenir la valeur du champ. Le chargement des classes est effectué seulement à la dernière minute et de façon dynamique. Le chargement paresseux comporte plusieurs avantages comme la réduction de l'utilisation de la mémoire, la réduction du débit sur le réseau et du coût de démarrage de la machine virtuelle.

Les codes d'opération comme le `NEW`, les `PUTFIELD`, les `PUTSTATIC`, les `GETFIELD`, les `GETSATIC`, les `NEWARRAY` et les `INVOKE<X>` sont des exemples susceptibles de provoquer le chargement et l'initialisation d'une classe pendant leur exécution. Comme nous sommes dans un environnement avec possiblement plusieurs *threads*, pour maintenir l'intégrité de la *vm*, la préparation d'une classe doit se faire de manière synchronisée (en accès exclusif). L'exclusion mutuelle est implantée en utilisant les verrous d'exclusion mutuelle de POSIX (*pthead mutexes*). Les codes d'opération énumérés ci-haut doivent donc être synchronisés. Le problème ici est que le coût additionnel de la synchronisation devra être payé à toutes les exécutions de ces instructions, alors qu'il est nécessaire de préparer la classe seulement qu'une seule fois.

Un surcoût supplémentaire est associé à la vérification qui permet de savoir si une

[2]L'initialisation d'une classe consiste en l'initialisation des champs statiques ainsi qu'à l'exécution des initialiseurs statiques associés à cette classe.

classe a été chargée, créée et initialisée. Ce coût est infligé chaque fois qu'il faut exécuter un de ces codes d'opération. Une fois la classe chargée, pourquoi continuer à effectuer ce test ? La solution à ces deux problèmes est la création de deux versions d'instructions pour chacun des codes d'opération énumérés ci-haut.

Codes d'opération à deux versions

SableVM possède donc, dans son jeu d'instructions, deux versions de ces codes d'opération : une version lente synchronisée avec vérification et une version rapide non synchronisée sans vérification. La première fois que nous rencontrons un de ces codes d'opération, la version lente est exécutée. Ensuite, le code d'opération (version lente) est remplacé dans le tableau de code par le code d'opération de la version rapide. De cette façon, lors des prochaines visites il ne sera plus nécessaire de payer le coût de la synchronisation et d'effectuer la vérification inutilement. Nous expliquons en détail, dans la section 1.4 (séquences de préparation), comment fonctionne ce mécanisme.

1.3.7 Formatage du code pour le mode d'interprétation désiré

Le formatage du tableau de code dépend intrinsèquement du mode d'interprétation sélectionné. Par exemple, avec l'interpréteur par aiguillage (section 1.5.1), les codes d'opération seront représentés dans le tableau par les entiers qui les représentent. Avec l'interpréteur linéaire direct (section 1.5.2), les codes d'opération sont représentés dans le tableau par l'adresse du début de leur implémentation. Nous reviendrons en détail sur ceci dans la section 1.5. Nous le mentionnons dans cette section uniquement pour indiquer que la construction du tableau de code, selon le mode d'interprétation, se fait aussi au moment de la préparation de la méthode.

1.4 Les séquences de préparation

Comme nous l'avons expliqué dans la section 1.3.6, SableVM possède deux versions pour les instructions pouvant provoquer le chargement et l'initialisation de classes. Par défaut, on utilise la version lente. Après avoir exécuté la version lente au moins une fois,

celle-ci est dynamiquement substituée par la version rapide. Cette section est consacrée à l'explication de ce mécanisme. Celui-ci est mis en place par l'entremise des séquences de préparation [Gag02, GH03].

L'idée derrière le concept des séquences de préparation est de dupliquer certaines parties du tableau de code, relatives aux codes d'opération à deux versions. Nous laissons dans le tableau de code la version rapide, non synchronisée et sans vérification de l'instruction, tandis que nous plaçons à la fin du tableau de code, dans une sorte de prolongation du tableau, la version lente, synchronisée et avec vérification. Une substitution atomique est ensuite utilisée pour diriger le flot de contrôle sur la bonne instruction à exécuter lors de l'interprétation du tableau de code.

La figure 1.3 illustre le code Java, le code octet du fichier *.class* et le code[3] créé par SableVM au moment de la préparation de la méthode *getValue()*. La figure illustre le code avant la première exécution et après la première exécution.

Dans la figure 1.3, nous constatons que la première instruction du code SableVM est un `LDC_INTEGER` plutôt qu'un `BIPUSH`. Les opérations qui s'effectuent sur les *bytes*, *shorts* et *integers* dans le code octet sont converties en opérations sur des entiers (le type *integer*). Dans SableVM, toutes les opérandes des codes d'opération du jeu d'instructions sont de même taille, c'est-à-dire de la taille d'un mot mémoire sur la plateforme en question.

À la ligne 03 du code SableVM, avant la première exécution, le `INVOKESTATIC` du code octet original a été remplacé par un `GOTO` vers la séquence de préparation qui commence à la ligne 10. Lorsque le flot d'exécution atteindra le `GOTO` de la ligne 03, il sautera à la ligne 10 pour exécuter la version lente du `INVOKESTATIC`. Le `PREPARE_INVOKESTATIC` est la version lente et synchronisée du `INVOKESTATIC` qui charge et initialise, si nécessaire, la classe qui contient la méthode *getValue()*. Losrque le `PREPARE_INVOKESTATIC` sera exécuté, il provoquera la préparation de la méthode appelée et de la classe qui la contient. Par la suite, l'instruction mettra à jour les lignes 05 et 06 du tableau de code. Ces lignes constituent en partie les opérandes du `INVOKESTATIC`. La ligne 05 est le nombre de paramètres que contient la méthode appelée et la ligne 06 est un pointeur vers le *method_info* de la méthode appelée. La mise à jour de ces champs permettra à la version rapide d'avoir

[3]Lorsque nous présentons du code créé par SableVM dans ce mémoire, nous utilisons toujours la même notation. Les instructions sont entre crochets et les opérandes des instructions sont entre parenthèses.

Code Java	Code octet du fichier .class

public int *getValue()*
{
  return 12 + getNum();
}
	public int *getValue();*
	  Code:
	  0: bipush
	  1: 12
	  2: invokestatic
	  3: #15; //Method getNum()
	  5: iadd
	  6: ireturn

Code SableVM avant la 1ère exécution	**Code SableVM après la 1ère exécution**
01 [LDC_INTEGER] 02 (12) 03 [GOTO] 04 (10) ← @ sequence Prep. 05 (0) 06 (null) 07 (@stack_gc_map) 08 [IADD] 09 [IRETURN] 10 [PREPARE_INVOKESTATIC] 11 (05) 12 (@methodref_info) 13 (@stack_gc_map) 14 (REPLACE] 15 (03) 16 (INVOKESTATIC) 17 [GOTO] 18 (08)	01 [LDC_INTEGER] 02 (12) 03 [INVOKESTATIC] 04 (10) 05 (0) 06 (@method_info) 07 (@stack_gc_map) 08 [IADD] 09 [IRETURN] 10 [PREPARE_INVOKESTATIC] 11 (05) 12 (@methodref_info) 13 (@stack_gc_map) 14 (REPLACE] 15 (03) 16 (INVOKESTATIC) 17 [GOTO] 18 (08)

Figure 1.3 – Séquence de préparation pour un appel statique.

un accès instantané à la méthode qu'il faut invoquer lors des prochains passages sur ce site d'invocation. Une fois la mise à jour de ces valeurs effectuée, l'instruction va lancer l'exécution de la méthode appelée.

Lorsque le flot d'exécution reviendra de la méthode appelée, le `pc` sera sur l'instruction de la ligne 14. Le `REPLACE` sera alors exécuté et remplacera le `GOTO` (ligne 03) par un `INVOKESTATIC`. Une fois la substitution terminée, le `GOTO` de la ligne 17 sera exécuté et le flot d'exécution reprendra son cours à la suite du `INVOKESTATIC` qui vient d'être préparé. Le `GOTO` sautera donc à la ligne 08.

Lors des prochains passages sur le site d'invocation situé à la ligne 03, le `INVOKESTATIC` (rapide) sera exécuté. L'implémentation de cette instruction commence par un `pc++` qui lui permet de sauter par dessus l'opérande maintenant inutilisé à la ligne 04, qui était utilisée par l'ancien `GOTO`.

Nous sommes dans un contexte avec potentiellement plusieurs *threads* exécutant le

même tableau de code et l'intégrité du tableau de code est toujours conservée. Les modifications des lignes 04 et 05 effectuées par le `PREPARE_INVOKESTATIC` ne viennent pas invalider le tableau de code, puisque si un autre *thread* exécute le code de cette méthode au même moment, il sera obligé de sauter par-dessus ces lignes à cause du `GOTO` de la ligne 03. Il se verra obligé de passer par la séquence de préparation. Il sera probablement obligé d'attendre pour le verrou, qui protège la préparation de la classe (qui contient la méthode), pour se rendre compte par la suite que la classe a déjà été préparée (chargée et initialisée). La deuxième modification est effectuée par le `REPLACE` et elle est atomique. SableVM prend les mesures nécessaires pour s'assurer de l'atomicité de cette opération [Gag02]. Le `REPLACE` est la dernière modification faite au tableau de code par la séquence de préparation. À ce moment, les opérandes de la version rapide (ligne 04 et 05) ont déjà été mises à jour.

L'utilisation des séquences de préparation se résume de la façon suivante. On remplace l'instruction originale par un `GOTO` vers la séquence de préparation. La séquence est le plus souvent constituée de la version lente de l'instruction, qui est ensuite suivie d'un `REPLACE`, qui a pour rôle de remplacer le `GOTO` par la version rapide de l'instruction. La séquence se termine presque toujours par un `GOTO` qui ramène le flot d'exécution au bon endroit, c'est-à-dire à la suite de l'instruction que la séquence vient de préparer.

Quand SableVM utilise son interpréteur par aiguillage ou linéaire direct, une séquence de préparation prépare une seule instruction comme dans la figure 1.3. Par contre, il est possible qu'une séquence de préparation prépare plusieurs instructions et ceci se produit à l'occasion quand SableVM utilise son interpréteur linéaire inclusif. Dans SableVM, nous utilisons les séquences de préparation afin d'accélérer l'exécution des instructions `LDC`, `NEW`, `ANEWARRAY`, `CHECKCAST`, `INSTANCEOF`, `MULTIANEWARRAY`, tous les types de `GETSTATIC`, `PUTSTATIC`, `GETFIELD`, `PUTFIELD` et de `INVOKE`.

1.5 Les trois modes d'interprétation de SableVM

Dans cette section, nous allons mettre en lumière les trois modes d'interprétation de SableVM. Grâce à une option de compilation, il est possible de choisir le mode d'interprétation. SableVM peut être compilé avec un interpréteur par aiguillage, linéaire direct

ou linéaire inclusif. Comme notre projet de recherche est une extension de l'interpréteur linéaire inclusif, il est important de comprendre son fonctionnement. Pour ce faire, il est utile de comprendre les deux autres types d'interpréteurs (par aiguillage et linéaire direct).

1.5.1 Interpréteur par aiguillage *(switch)*

L'interpréteur par aiguillage est un interpréteur naïf et il est extrêmement simple à mettre en place. Il est cependant le type d'interpréteur le moins performant. Son nom vient du fait qu'on utilise la structure de contrôle *switch* pour l'implémenter. Chaque code d'opération composant le jeu d'instructions est représenté par un entier. Le *switch* comprend un *case* pour chacun des codes d'opération composant le jeu d'instructions. La position courante dans le tableau de code est maintenue par le `pc`. Une boucle est utilisée pour itérer sur le tableau de code à interpréter. Le *switch*, contenu dans la boucle, se chargera de sauter à l'implantation (le *case*) du code d'opération à exécuter. L'implantation de l'instruction en question se charge d'incrémenter le `pc` afin de permettre la lecture de ses propres opérandes.

Le corps de la boucle de répartition doit faire la lecture en mémoire de la prochaine instruction à exécuter *(fetch)* et déléguer le flot d'exécution au *case* qui implante cette instruction. Chaque instruction saute à l'instruction suivante en effectuant un *break* qui permet de sortir du *switch* et de passer le contrôle au début de la boucle de répartition. La figure 1.4 illustre un interpréteur *switch* minimaliste écrit avec le langage C.

La mauvaise performance de cet interpréteur avec un langage comme le code octet Java est causée par la combinaison de deux facteurs. Le corps des instructions composant le code octet est relativement petit et il existe un coût important associé à la répartition *(dispatchs)* entre chaque instruction du tableau de code. Plus le rapport entre le coût de la réparation et le coût de l'instruction à exécuter est grand, moins ce type d'interpréteur est performant.

Si on suppose que le compilateur optimise la chaîne de sauts nécessaire pour passer des *breaks* jusqu'au saut implicite à la fin de la boucle qui permet de retourner au début de celle-ci, le surcoût associé au mécanisme de répartition peut être expliqué par les opérations suivantes : incrémentater le `pc`, aller chercher et faire la lecture de la prochaine instruction

```
char code[CODE_SIZE];
char *pc = code;
int stack[STACK_SIZE];
int *sp = stack;

/* charger le code dans le tableau de code */
..

/* boucle de répartition */
while(1) {
   switch(*pc++) {
      case ICONST_1: *sp++ = 1; break;
      case ICONST_2: *sp++ = 2; break;
      case IADD: --sp; sp[-1] += *sp; break;
      case POP: sp--; break;
      ..
      case END: exit(0);
   }
}
```

Figure 1.4 – Interpréteur par aiguillage (*switch*).

en mémoire *(fetch)*, vérifier de façon redondante les bornes (des valeurs des *cases*) avec l'argument du *switch*, aller chercher l'adresse du bon *case* dans une table, sauter à cette adresse. À la fin de chaque instruction, il faut sauter au début de la boucle pour aller chercher en mémoire la prochaine instruction.

Selon Piumarta et Riccardi [PR98], pour l'exécution d'un code d'opération, il peut y avoir jusqu'à 9 instructions machines utilisées pour la répartition sur un PowerPC alors que le travail réel effectué ne représente dans certains cas qu'une seule instruction machine. La majeure partie du temps est donc passé à mettre en marche le mécanisme de répartition. La stratégie utilisée par les interpréteurs plus performants est de réduire le coût engendré par le mécanisme de répartition.

1.5.2 Interpréteur linéaire direct (*direct-threaded*)

Cette technique, connue sous le nom de *threaded code*, est assez vieille et a été introduite au début des années 70 [Bel73]. Les deux principaux types de *threaded code* sont le *indirect-threading* et le *direct-threading*. Nous allons nous concentrer sur le deuxième, car c'est le plus performant des deux et c'est celui qui est utilisé dans SableVM. La technique linéaire directe (*direct-threading*) fut rendue populaire par le langage de programmation *Forth* [Ert93].

Ce type d'interpréteur est plus performant que l'interpréteur par aiguillage, car il diminue le coût associé à la répartition entre chaque instruction du code octet en éliminant la boucle centrale. Le format du tableau de code diffère un peu de celui utilisé dans l'interpréteur par aiguillage. La principale différence réside dans le fait qu'il n'utilise plus un entier pour représenter l'instruction à exécuter, mais plutôt l'adresse de son implémentation. La deuxième différence avec l'interpréteur par aiguillage est que chaque implémentation des instructions se termine par le code nécessaire pour effectuer la répartition vers la prochaine instruction.

```
char *code[] = {&&ICONST_2, &&POP, &&ICONST_2,
                &&ICONST_1, &&IADD, ..};
char *pc;
int stack[STACK_SIZE];
int *sp = stack;

/* remplir le tableau de code */
..

pc = code;

/* commencer à la 1ère instruction */
goto **(pc++);

/* implémentations */
ICONST_1: { *sp++ = 1; goto **(pc++); }
ICONST_2: { *sp++ = 2; goto **(pc++); }
IADD: {--sp; sp[-1] += *sp; goto **(pc++); }
POP: { sp--; goto **(pc++); }
..
```

Figure 1.5 – Interpréteur linéaire direct (*direct-threaded*).

La figure 1.5 montre une implémentation minimaliste, en langage C, d'un interpréteur linéaire direct. Lors de la préparation du code, il faut mettre l'adresse de l'implémentation d'une instruction plutôt que l'entier la représentant dans le tableau de code. Chaque implémentation dans le code C commence par une étiquette désignant l'instruction. Par exemple, pour mettre un `IADD` dans le tableau de code, il suffit de mettre `&&IADD` à l'emplacement voulu. Ceci désigne l'adresse en mémoire de la première instruction de l'implémentation. Dans cet exemple et dans SableVM nous utilisons la fonctionnalité *label-as-value* qui est une extension disponible avec GNU C. Cette fonctionnalité permet la manipulation des étiquettes telle que présentée dans la figure 1.5. Il serait aussi

possible d'implanter un interpréteur linéaire direct sans cette fonctionnalité, en ajoutant des macros contenant quelques lignes d'assembleur spécifique à la plateforme.

Exécution

En nous basant sur la figure 1.5, voici comment se déroule l'exécution. On commençe par initialiser le `pc` pour qu'il pointe sur le premier emplacement du tableau de code `[ pc = code; ]`. Le flot d'exécution se dirige ensuite à l'adresse stockée dans le premier élément du tableau. Ce saut est réalisé par la ligne `[ goto **(pc++); ]`. L'instruction cible est alors exécutée (`ICONST_2` dans notre exemple). La répartition vers la prochaine instruction `(POP)` est effectuée par la dernière ligne de l'implémentation de `ICONST_2`, c'est-à-dire `[ goto **(pc++); ]`. L'exécution se poursuit de cette façon jusqu'à la fin du tableau de code.

Ce type de répartition, entre les instructions exécutées, élimine le surcoût associé à la boucle centrale et à la consultation de la table qui implémente le *switch* de l'interpréteur par aiguillage. Ce type d'interpréteur est relativement simple à implanter, mais il a l'inconvénient de ne pas être à 100% portable. Pour le porter, il est nécessaire d'écrire quelques lignes d'assembleur spécifique à la plateforme, si jamais la fonctionnalité *label-as-value* de GNU C n'est pas disponible.

1.5.3 Interpréteur linéaire inclusif *(inlined-threaded)*

L'interpréteur linéaire inclusif est le plus performant des trois interpréteurs implantés par SableVM. Il va chercher un gain de performance supplémentaire en éliminant les répartitions entre les instructions contenues dans un même bloc de base. Cet interpréteur se base sur une technique moderne d'interprétation, le *inline-threading* [PR98].

Le concept de bloc de base

Un bloc de base est une séquence d'instructions dans le tableau de code. Cette séquence ne doit pas contenir d'instruction de saut ou de branchement dans une position autre que la dernière et seulement la première instruction de la séquence peut être la destination

d'un saut où d'un branchement conditionnel. La figure 1.6 donne l'algorithme qui permet d'obtenir les blocs de bases pour un tableau de code donné.

Entrée: Une séquence d'instruction.
Résultat: Une liste de blocs de base où chaque instruction appartient à un seul bloc de base.

Algorithme:
1. On détermine l'ensemble des instructions de tête de chaque bloc. Les règles utilisées sont les suivantes:
    a. La première instruction est une instruction de tête.
    b. Toute instruction qui peut être atteinte par une instruction de branchement conditionnel ou inconditionnelle est une instruction de tête.
    c. Toute instruction qui suit immédiatement un branchement conditionnel ou inconditionnel est une instruction de tête.
    d. Toute instruction qui est le début d'un *handler* pour une exception est une instruction de tête.
    e. Toute instruction qui marque le début d'un bloc *try/catch* est une instruction de tête.
    f. Toute instruction qui suit la dernière instruction d'un bloc *try/catch* est une instruction de tête.
2. Pour chaque instruction de tête, le bloc de base qui lui correspond débute par celle-ci et est constitué par toutes les instructions suivantes qui ne sont pas une autre instruction de tête.

Figure 1.6 – Algorithme de segmentation en blocs de base.

Cet algorithme est tel que nous pouvons le trouver dans la littérature [ASU86], à la différence que nous lui avons ajouté les clauses nécessaires (d, e, f dans la figure 1.6), pour tenir compte de l'existence des exceptions en Java. Cet algorithme doit être appliqué sur le code octet pour pouvoir déterminer les blocs de base qui le compose. La figure 1.7 illustre du code octet Java généré par *Jikes*, sur lequel nous avons appliqué l'algorithme pour en déterminer les blocs de base.

La création de super-instructions

L'idée derrière l'interpréteur linéaire inclusif est la suivante. On commençe par identifier les blocs de base dans le tableau de code. Ensuite, on crée dynamiquement, pour chaque bloc de base, une nouvelle implémentation pour toute la séquence composant ce bloc. On accomplit ceci en copiant le corps de chaque implémentation pour chaque instruction de la séquence. On copie ensuite, à la fin de cette nouvelle implémentation, le

Code octet Java	Blocs de bases	Code composé des super instructions
Code: 00: bipush 01: (12) 02: istore_3 03: iload_1 04: iload_2 05: iadd 06: istore 07: (4) 08: iload 09: (4) 10: iload_3 11: if_icmpge 12: (21) 14: iload_3 15: iconst_m1 16: imul 17: istore_3 18: goto 19: (24) 21: iload 22: (4) 23: istore_3 24: iload_3 25: iload_1 26: iload_2 27: iadd 28: iadd 29: istore_3 30: iload_3 31: ireturn	**/* Bloc_1 */** 00: bipush 01: (12) 02: istore_3 03: iload_1 04: iload_2 05: iadd 06: istore 07: (4) 08: iload 09: (4) 10: iload_3 11: if_icmpge 12: (21) **/* Bloc 2 */** 14: iload_3 15: iconst_m1 16: imul 17: istore_3 18: goto 19: (24) **/* Bloc 3 */** 21: iload 22: (4) 23: istore_3 **/* Bloc 4 */** 24: iload_3 25: iload_1 26: iload_2 27: iadd 28: iadd 29: istore_3 30: iload_3 31: ireturn	**Code:** 00: [@super_instruction1] 01: (12) 07: (4) 09: (4) 12: (21) 14: [@super_instruction2] 19: (24) 21: [@super_instruction3] 22: (4) 24: [@super_instruction4] Ici la super_instruction<i> a été créée à partir du bloc de base i. Elle contient donc toutes les instructions contenues dans le bloc de base i.

Figure 1.7 – Exemple de segmentation en blocs de base.

code nécessaire à la répartition vers la prochaine instruction, par exemple le code pourrait être `[ goto **(pc++); ]`. Ensuite, on met dans le tableau de code un pointeur vers cette nouvelle implémentation, plutôt que les instructions composant la séquence elle-même. Nous venons donc de créer ce que nous appelons une super-instruction qui est composée de toutes les instructions de la séquence formant le bloc de base.

La figure 1.7 donne un exemple de construction d'un tableau de code linéaire inclusif. Illustrons la création d'une super-instruction en utilisant le bloc de base 3. Lors de la préparation du tableau de code, au moment de traiter le bloc de base 3, on commençe

par allouer dynamiquement un espace pour stocker l'implantation de la super-instruction qu'il faudra créer. Ensuite, on copie le corps des implantations des instructions `ILOAD` et `ISTORE_3` dans le nouvel emplacement. On copie à la suite de ceux-ci le corps du code de répartition vers la prochaine instruction. Chaque super-instruction se termine par le code de répartition, de manière similaire à ce qui était fait dans la section 1.5.2 pour le mode linéaire direct. Une fois la super-instruction créée, on ajoute dans le tableau de code un pointeur vers le début de la super-instruction et on plaçe à la suite de ce pointeur la valeur des opérandes utilisées par les instructions qui composent la super-instruction.

La figure 1.8 illustre le code partiel et simplifié utilisé pour la création de l'implantation d'une super-instruction qui serait composée des codes d'opération `ICONST_1`, `ICONST_2` et `IADD`. Une fois la séquence de la figure 1.8 construite, il serait possible de l'exécuter en exécutant `[ goto **buf; ]`.

La construction de telles super-instructions permet d'éliminer toutes les répartitions qui existent dans les instructions qui la composent. Selon Gagnon et Hendren [GH03], l'interpréteur linéaire inclusif de SableVM interprète le code jusqu'à 2.41 fois plus vite que son interpréteur par aiguillage et jusqu'à 2.14 fois plus vite que son interpréteur linéaire direct.

```
/* instructions */
ICONST_1_START: *sp++ = 1;
ICONST_1_END: ;

ICONST_2_START: *sp++ = 2;
ICONST_2_END: ;

IADD_START: --sp;
   sp[-1] += *sp;

IADD_END: ;

DISPATCH_START: goto **(pc++);
DISPATCH_END: ;
```

```
/* construction de la super instruction */
size_t iconst1_size = (&&ICONST_1_END - &&ICONST_1_START);
size_t iconst2_size = (&&ICONST_2_END - &&ICONST_2_START);
size_t iadd_size = (&&IADD_END - &&IADD_START);
size_t dispatch_size = (&&DISPATCH_END - &&DISPATCH_START);

void *buf = malloc(iconst1_size + iconst2_size + iadd_size + dispatch_size);
void *current = buf;

memcpy(current, &&ICONST_1_START, iconst1_size);
current += iconst1_size;
memcpy(current, &&ICONST_2_START, iconst2_size);
current += iconst2_size;
memcpy(current, &&IADD_START, iadd_size);
current += iadd_size;
memcpy(current, &&DISPATCH_START, dispatch_size);
current += dispatch_size;
..
```

Figure 1.8 – Création d'une super-instruction.

Chapitre 2

Sommaire du système d'inclusion partielle des méthodes

Dans ce chapitre, nous allons présenter sommairement le système d'inclusion partielle des méthodes que nous avons élaboré. Le but de ce chapitre est de donner une vue d'ensemble du système et d'expliquer quels sont les objectifs visés par celui-ci. La première partie de ce chapitre présente les buts que nous voulons atteindre et les stratégies que nous comptons utiliser pour les atteindre. Nous présentons ensuite le concept général d'inclusion des méthodes sur lequel nous nous sommes basé pour élaborer notre système d'inclusion partielle. Nous expliquons ensuite pourquoi nous n'incluons qu'une partie du code de la méthode appelée. Nous présentons aussi la notion de point de vérification qui est utilisée au moment de l'exécution pour la validation de l'identité de la méthode incluse ainsi que pour la validation du chemin sélectionné lors de l'inclusion. Nous terminons le chapitre en expliquant pourquoi nous avons besoin de dupliquer certaines séquences en mode linéaire inclusif pour assurer l'intégrité de notre système.

2.1 Objectifs du mécanisme d'inclusion des méthodes

Nous avons vu dans la section 1.5 que la stratégie la plus souvent utilisée pour optimiser la performance des interpréteurs est de réduire le coût associé aux répartitions entre chaque instruction interprétée. Dans le cas d'un interpréteur linéaire inclusif, on élimine

complètement les répartitions entre les codes d'opération d'un même bloc de base. C'est cette réduction des répartitions qui explique la popularité de ce type d'interpréteur. Plus la longueur moyenne des séquences est grande, plus le nombre de répartitions exécutées diminue. Notre système est bâti entièrement sur ce principe. Nous avons conçu un mécanisme d'inclusion des méthodes qui vise à éliminer la frontière entre les méthodes et ainsi permettre d'augmenter la longueur moyenne des super-instructions exécutées par l'interpréteur linéaire inclusif. Ce faisant, nous réduisons directement le nombre de répartitions exécutées au cours de l'interprétation.

2.2 Inclusion des méthodes

Le procédé d'inclusion des méthodes (ou expansion des appels de méthodes) est une optimisation bien connue et utilisée dans le domaine de la compilation [ASU86]. Il s'agit de remplacer l'appel d'une méthode par le code de celle-ci, éliminant ainsi le surcoût associé à cet appel. On élimine de cette façon la création, l'initialisation et la destruction du bloc d'activation associé à l'appel. Ce procédé est particulièrement utile pour les méthodes qui possèdent des petits corps et dont le code est beaucoup plus petit que celui utilisé pour la création et la destruction des blocs d'activation. Ceci permet aussi au compilateur d'effectuer de meilleures optimisations qui seront spécifiques au site d'invocation. Normalement, ce traitement est effectué statiquement au moment de la compilation. Le principal inconvénient de ce procédé est qu'il peut générer du code machine plus long. La figure 2.1 donne un exemple d'inclusion simple. La figure illustre bien les possibilités d'optimisations qu'occasionne l'élimination de la frontière entre les méthodes.

Nous nous sommes inspiré de ce procédé pour élaborer notre stratégie d'augmentation de la longueur moyenne des séquences. Ce qui différencie notre type d'inclusion avec celui qui est utilisé en compilation est la manière dont nous comptons en tirer avantage et le fait que normalement l'inclusion se fait de manière statique tandis que notre système effectue l'inclusion de manière dynamique pendant l'interprétation. Nous comptons sur l'allongement des super-instructions pour obtenir une réduction des répartitions plutôt que sur l'élimination du surcoût associé à la création du bloc d'activation.

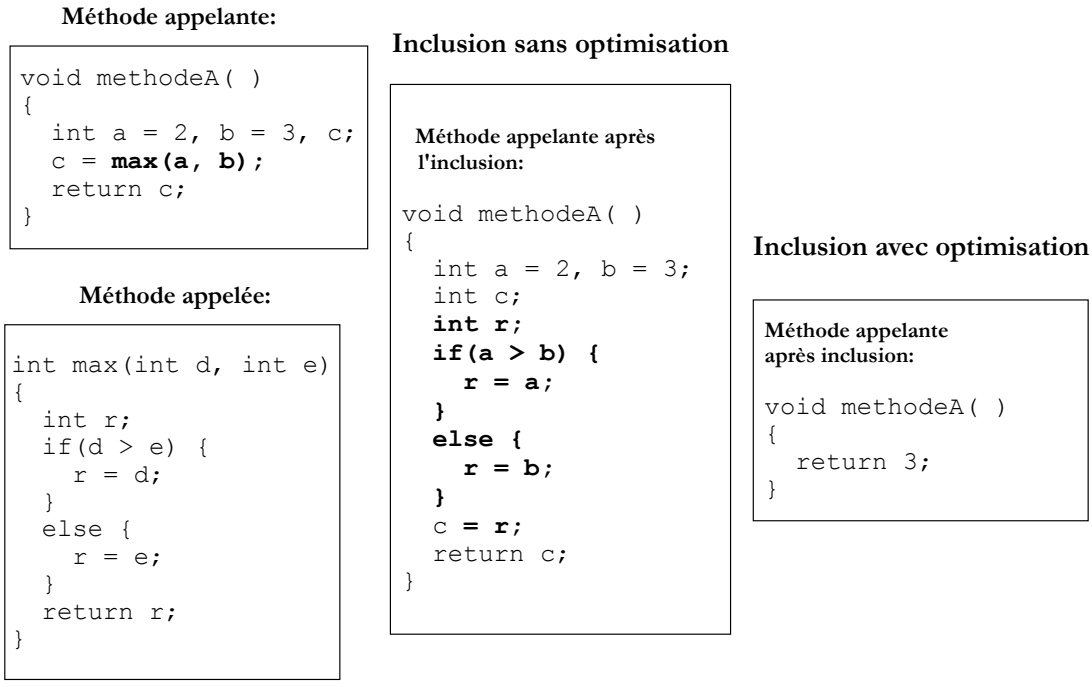

Figure 2.1 – Exemple classique d'inclusion effectuée par un compilateur.

2.2.1 Conservation du bloc d'activation

Nous ne prévoyons pas faire des gains de performance par l'élimination de la création des blocs d'activations puisque nous n'éliminons pas les blocs d'activations lors de l'exécution du code inclus. Nous effectuons l'inclusion de la méthode appelée, mais lors de l'interprétation de ce code, nous conservons quand même le bloc d'activation associé à la méthode invoquée. Ce choix nous permet de conserver, dans un espace distinct, les variables locales et la pile des opérandes de la méthode appelante et de la méthode appelée. Sans la conservation de cet espace distinct, nous devrions effectuer une analyse approfondie du code de la méthode appelée afin de modifier l'indexation de ses variables locales pour éviter les collisions avec celles de la méthode appelante. Les coûts associés à des changements majeurs dans le code de la méthode appelée auraient vite fait de dépasser les gains que nous espérons par la construction des séquences plus longues. Sans compter le fait que nous devrions modifier la taille du tableau des variables locales et de la pile des opérandes de la méthode appelante pour permettre l'interprétation du code inclus. Nous

avons voulu préserver la simplicité de notre mécanisme d'inclusion au maximum afin de minimiser le coût associé à celui-ci.

La figure 2.2 illustre l'inclusion d'une méthode statique dans le contexte de l'interpréteur par aiguillage de SableVM. Nous y voyons que le `INVOKESTATIC` de la méthode appelante (adresse 05) a été remplacé par une partie du code de la méthode appelée. Dans cet exemple, le code inclus commence par le code d'opération `INLINED _INVOKESTATIC` qui sert à créer le bloc d'activation de la méthode invoquée. Nous remarquons aussi que le code inclus se termine par le code d'opération `INLINED_IRETURN` qui est utilisé pour détruire le bloc d'activation de la méthode invoquée et mettre la valeur de retour sur la pile des opérandes de la méthode appelante. Ces deux instructions sont utilisées pour conserver des contextes d'exécution distincts pour les deux méthodes. Comme mentionné plus haut, l'exécution du code inclus de l'exemple 2.2 ne nous offre pas de diminution du nombre de répartitions. Pour bénéficier d'une diminution, nous devons examiner le mécanisme d'inclusion en mode linéaire inclusif. La figure 2.3 reprend exactement le même exemple que celui de la figure 2.2, mais illustre l'inclusion en mode linéaire inclusif.

2.2.2 Construction de plus longues séquences (super-instructions)

Dans l'interpréteur linéaire inclusif, pour un site d'invocation donné, si on remplaçe le code d'opération faisant l'appel de la méthode (un des quatre types de `INVOKE` section 1.2.3) par les codes d'opération contenus dans le code de la méthode appelée, nous pourrons possiblement construire une séquence[1] plus longue. Les codes d'opération de type *invoke* se retrouvent toujours au sein de séquences de longueur égale à 1. Si nous effectuons l'inclusion de la méthode appelée à un site d'invocation donné, nous pourrons construire une nouvelle séquence constituée de la séquence précédant le *invoke*, du code de la méthode appelée et de la séquence qui suit le *invoke* en question. Dans la figure 2.3, on remarque dans la méthode appelante après l'inclusion, qu'une super-instruction a été créée à l'adresse 225 et qu'elle contient : la super-instruction qui précédait le *invoke*, le code de la méthode appelée ainsi que la super-instruction qui suivait le *invoke*. Nous avons donc fusionné plusieurs séquences et éliminé les répartitions qui les séparaient. Puisque ce

[1] Dans ce mémoire, nous utilisons souvent le terme séquence pour parler d'une super-instruction.

Méthode appelante
(avant l'inclusion)

```
Method void main(java.lang.String[])
00: [BIPUSH]
01: (12)
02: [ISTORE_1]
03: [ILOAD_1]
04: [ILOAD_1]
--
05: [INVOKESTATIC]
06: (@21) <- adresse seq. preparation
07: (args_count)
08: (@method_info)
09: (@stack_gc_map)
--
10: [IADD]
11: [ISTORE_1]
12: [ILOAD_0]
13: [IF_EQ]
14: (@20)
15: [IINC]
16: (1)
17: (-1)
18: [GOTO]
19: (@12)
20: [RETURN]
21: [PREPARE_INVOKESTATIC]
22: ...
```

Méthode appelée
(les * indiquent le chemin sélectionné)

```
Method int method1(int)
* 30: [ILOAD_0]
* 31: [ISTORE_1]
* 32: [LDC]
* 33: (11)
* 34: [ISTORE_0]
* 35: [ILOAD_0]
* 36: [ICONST_4]
* 37: [IF_ICMPGE]
* 38: (@45)
  39: [ILOAD_1]
  40: [ILOAD_0]
  41: [IADD]
  42: [ISTORE_1]
  43: [GOTO]
  44: (@49)
* 45: [ILOAD_1]
* 46: [ILOAD_0]
* 47: [IADD]
* 48: [ISTORE_1]
* 49: [ILOAD_1]
* 50: [IRETURN]
```

Méthode appelante
(après l'inclusion)

```
Method void main(java.lang.String[])
70: [BIPUSH]
71: (12)
72: [ISTORE_1]
73: [ILOAD_1]
74: [ILOAD_1]
--
75: [INLINED_INVOKESTATIC]
76: (16) <- taille du code inclus
77: (args_count)
78: (@method_info)
79: (@stack_gc_map)
80: [ILOAD_0]
81: [ISTORE_1]
82: [LDC]
83: (11)
84: [ISTORE_0]
85: [ILOAD_0]
86: [ICONST_4]
87: [INLINED_IF_ICMPLT] <- check point
88: (@39) <- dans code original
89: [ILOAD_1]
90: [ILOAD_0]
91: [IADD]
92: [ISTORE_1]
93: [ILOAD_1]
94: [INLINED_IRETURN]
95: (@stack_gc_map)
--
96: [IADD]
97: [ISTORE_1]
98: [ILOAD_0]
99: [IF_EQ]
100: (@106)
101: [IINC]
102: (1)
103: (-1)
104: [GOTO]
105: (@98)
106: [RETURN]
107: [PREPARE_INVOKESTATIC]
108: ...
```

Figure 2.2 – Exemple simple d'inclusion d'une méthode statique (interpréteur *par aiguillage*).

site d'invocation est un point chaud (zone de code souvent visitée), nous pensons pouvoir bénéficier rapidement de la diminution du nombre de sauts entre les séquences à cet endroit. Dans cette figure, les super-instructions portent le nom des instructions élémentaires qui les composent (séparés par des virgules).

2.2.3 Élimination des branchements dans le code inclus

Notre objectif ultime est de s'assurer que le code de la méthode incluse puisse être entièrement compris dans la nouvelle séquence. Si le code de cette méthode est lui-même constitué de plusieurs séquences, nous ne pourrons pas atteindre l'objectif de créer une seule séquence. Il ne faut pas oublier que nous visons l'élimination du plus grand nombre de répartitions possible et que celles-ci se trouvent entre les séquences. Pour atteindre notre but, le code inclus de la méthode appelée ne doit pas contenir de sauts, de branchements ou de séquences de préparations (section 1.4). Nous n'effectuons pas l'inclusion systématique du code de la méthode appelée, car nous voulons que le code inclus puisse le plus souvent possible respecter ces contraintes.

Afin de permettre l'inclusion de code qui respecte les conditions énoncées ci-haut, nous incluons le chemin le plus fréquenté dans la méthode appelée. En sélectionnant un tel chemin, nous nous assurons que la séquence que nous créerons ne contiendra pas de sauts, de branchement et de séquences de préparation. Nous avons mis en place dans SableVM un système de profilage dynamique du code interprété qui nous permet de déterminer les sites d'invocations chauds ainsi que le chemin le plus fréquenté dans une méthode donnée. Nous détaillons le système de profilage dans le chapitre 4. Ce système nous permet, pour chaque instruction de branchement contenue dans le code de la méthode appelée, de déterminer la destination la plus fréquente.

Dans l'exemple 2.2, notre système de profilage nous a permis de déterminer que l'instruction à l'adresse 37 (`IF_ICMPGE`) branchait à l'adresse 45 plus souvent qu'autrement. Le chemin le plus fréquenté dans cette méthode saute habituellement par-dessus le code situé entre les adresses 39 et 44 inclusivement. En nous basant sur les informations de profilage, nous avons effectué une inclusion partielle de la méthode. Nous avons inclus uniquement le chemin le plus fréquenté.

Méthode appelante (avant l'inclusion)

```
201: [BIPUSH, ISTORE_1, ILOAD_1, ILOAD_1]
202: (12)
--
203: [INVOKESTATIC]
204: (@216) <- adresse seq. de preparation
205: (args_count)
206: (@method_info)
207: (@stack_gc_map)
--
208: [IADD, ISTORE_1]
209: [ILOAD_0, IF_EQ]
210: (@215)
211: [IINC, GOTO]
212: (1)
213: (-1)
214: (@209)
215: [RETURN]
216: [PREPARE_INVOKESTATIC]
217: ...
```

Méthode appelée (les * représentent le chemin sélectionné)

```
*215: [ILOAD_0, ISTORE_1, LDC, ISTORE_0, ILOAD_0, ICONST_4, IF_ICMPGE]
*216: (11)
*217: (@220)
 218: [ILOAD_1, ILOAD_0, IADD, ISTORE_1, GOTO]
 219: (@221)
*220: [ILOAD_1, ILOAD_0, IADD, ISTORE_1]
*221: [ILOAD_1, IRETURN]
```

Méthode appelante (après l'inclusion)

```
225: [BIPUSH, ISTORE_1,ILOAD_1, INLINED_INVOKESTATIC, ILOAD_0,ISTORE_1,
     ISTORE_1, LDC, ISTORE_0, ILOAD_0, ICONST_4, INLINED_IF_ICMPLT,
     ILOAD_1, ILOAD_0, IADD, ISTORE_1, ILOAD_1, INLINED_IRETURN,
     IADD, ISTORE_1, INLINED_SKIP_NEXT_SEQ]
226: (12)
227: (args_count)
228: (@method_info)
229: (@stack_gc_map)
230: (11)
231: (@218) <- adresse dans le code original
232: (@235)
233: (@stack_gc_map)
234: [IADD, ISTORE_1]
235: [ILOAD_0, IF_EQ]
236: (@241)
237: [IINC, GOTO]
238: (1)
239: (-1)
240: (@235)
241: [RETURN]
242: ...
```

Figure 2.3 – Reprise de l'exemple de la figure 2.2 avec l'interpréteur linéaire inclusif.

En procédant de cette façon, lors de la prochaine exécution du site d'invocation, si notre profilage est bon et que le flot d'exécution reste sur le chemin que nous avons inclus, nous parviendrons à exécuter la totalité de la méthode incluse sans effectuer de répartitions (voir figure 2.3). Ceci est possible parce que le chemin en entier sera compris dans une seule séquence. Dans ce cas, nous éliminons aussi les répartitions qui sont placées à la suite des instructions de branchements dans le code original, ce qui vient contribuer encore une fois à notre objectif : *la réduction du nombre de répartitions.*

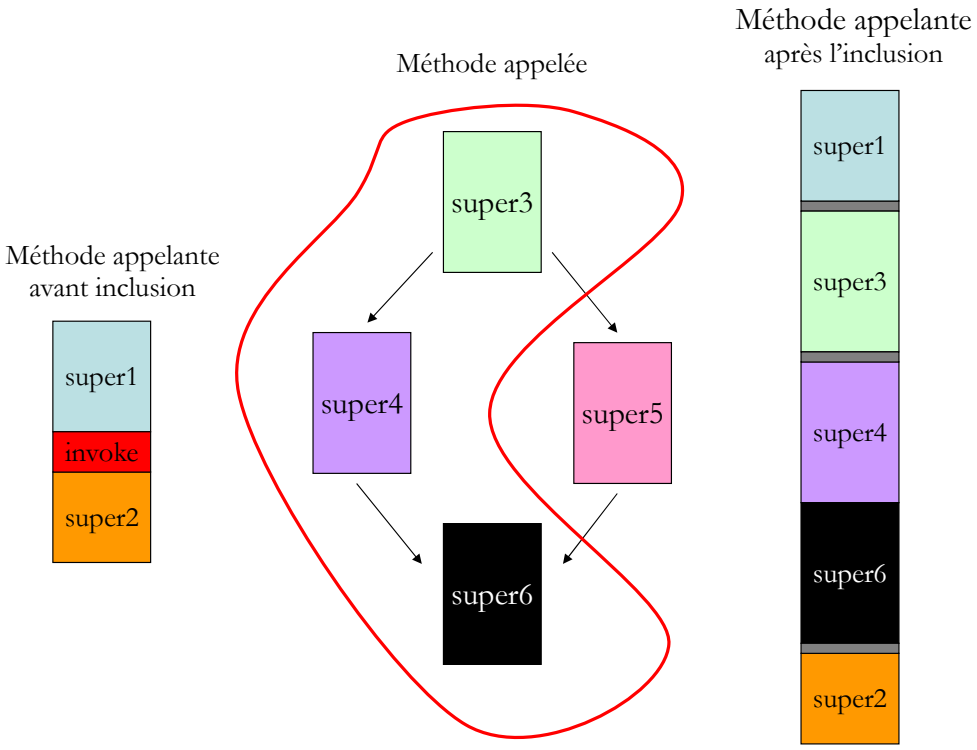

Figure 2.4 – Exemple imagé de l'inclusion partielle dans l'interpréteur linéaire inclusif.

La figure 2.4 reprend de manière imagée la sélection du chemin dans la méthode appelée et l'inclusion dans l'interpréteur linéaire inclusif. Les blocs super<X> représentent les blocs de base qui ont été converti en super-instructions. La région encerclée représente le chemin le plus fréquenté dans la méthode appelée. La figure nous montre le résultat final dans la méthode appelante, c'est-à-dire la construction d'une seule super-instruction composée de super1, super3, super4, super6 et super2.

2.2.4 Points de vérifications

Vérification de l'identité de la méthode incluse

S'il s'agit d'une méthode statique, il n'y a pas d'ambiguïté sur la validité de l'identité de la méthode qui a été incluse, puisqu'il ne peut en exister qu'une seule. Cependant pour les appels de méthodes d'instances non statiques, la méthode à invoquer est déterminée dynamiquement par le type de l'instance sur laquelle la méthode est appelée. Nous devons nous assurer, au sein même du `INLINED_INVOKE<X>`, que le code inclus est bien celui de la bonne méthode. Si ce n'est pas le cas, le flot d'exécution sort de la séquence courante pour continuer au début du code de la bonne méthode. Il est important de préciser que nous avons choisi la méthode à inclure grâce aux informations de profilage recueillies par notre système profilant. En théorie, la méthode appelée le plus souvent à ce site d'invocation est celle qui est incluse à cet endroit.

Vérification du chemin inclus

En effectuant l'inclusion partielle d'une méthode (en incluant seulement un de ses chemins d'exécutions), nous avons un problème si, dans certains cas, le chemin inclus n'est pas exact. Un chemin inclus n'est pas exact pour une exécution donnée quand celle-ci ne correspond plus au chemin qui a été inclus. Pour remédier à cette difficulté, nous introduisons des points de vérifications aux endroits (dans le code inclus) qui correspondent aux branchements qui se trouvaient dans le code original de la méthode appelée. Ces points de vérifications vérifient que les hypothèses que nous avons faites lors de la sélection du chemin sont toujours vraies au moment de l'exécution. Si le chemin inclus est valide, l'exécution se poursuit sans répartition et, sinon, le flot d'exécution saute au bon endroit dans le code original de la méthode appelée et l'exécution se poursuit de cet endroit sans frais additionnels. Si les choix que nous avons effectués grâce au profilage sont bons, nous diminuons le nombre de répartitions et, sinon, nous retournons dans le code original et l'exécution se poursuit normalement comme si nous n'avions jamais effectué l'inclusion de la méthode. Un des points forts de notre conception est l'absence de coût additionnel dans le cas où le chemin inclus est invalide pour un passage sur un site d'invocation donné. La

figure ?? reprend l'exemple de la figure 2.4, mais en illustrant la présence d'un point de vérification. Celui-ci valide, au moment de l'exécution, que nous sommes toujours sur le bon chemin et si ce n'est pas le cas, le flot d'exécution quitte le code inclus et continue son exécution au bon endroit dans le code original de la méthode appelée.

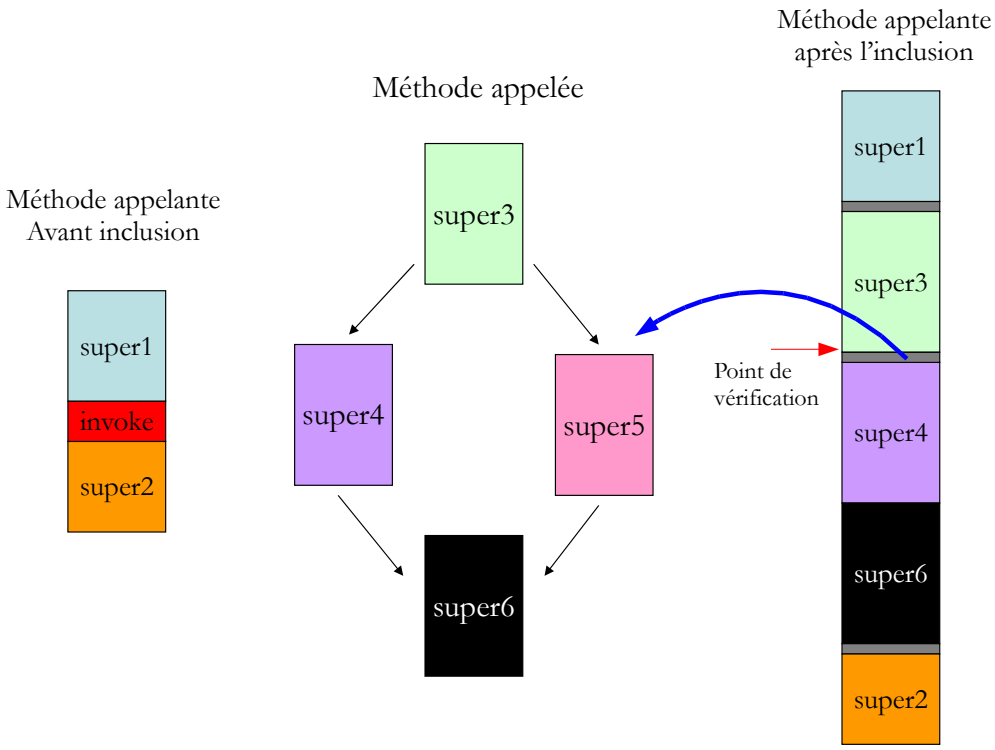

Figure 2.5 – Exemple imagé d'un point de vérification

Nous avons ajouté des codes d'opération au jeu d'instructions de SableVM qui sont utilisés comme points de vérifications (annexe B). Dans l'exemple B, un point de vérification (`INLINED_IF_ICMPLT`) a été inséré à l'adresse 87. Le chemin que nous avons inclus suppose que la condition associée au `IF_ICMPGE` de l'adresse 37 est toujours vraie, c'est-à-dire qu'à ce moment le premier entier sur le haut de la pile est toujours plus grand ou égal au second entier sur le haut de la pile. Lors de l'exécution du code inclus, le point de vérification vérifie la condition contraire, c'est-à-dire que le premier entier sur le haut de la pile est plus petit que le second entier sur le haut de la pile. Si c'est le cas, alors notre chemin est invalide et le point de vérification fait continuer le flot d'exécution à l'adresse

39 dans le code original de la méthode invoquée. Cette rectification du flot d'exécution est possible parce que l'implémentation du point de vérification contient un *goto*. Une fois l'exécution de la méthode invoquée terminée, dans les deux cas, le flot d'exécution se poursuit à l'adresse 96 de la méthode appelante. Dans la figure 2.3, le point de vérification correspondant se trouve au milieu de la super-instruction créée à l'adresse 225. Dans ce cas, au moment de l'exécution, si le chemin est valide, aucune répartition n'est effectuée lors de l'exécution du code du `INLINED_IF_ICMPLT` et l'exécution se poursuit au sein de la super-instruction. Ceci illustre le fait que nous éliminons des répartitions aussi aux points de vérifications si le chemin inclus est le bon.

2.2.5 Dédoublement de séquences

Dans l'exemple en mode linéaire inclusif de la figure 2.3, on remarque que la séquence `[IADD, ISTORE_1]` a été dédoublée. Elle se trouve à la fin de la super-instruction de l'adresse 225 et elle se trouve aussi à l'adresse 234. On remarque aussi l'ajout du code d'opération `INLINED_SKIP_NEXT_SEQ` à la fin de la super-instruction. Ce code d'opération, que nous avons ajouté au jeu d'instructions de SableVM, a comme opérande la valeur qui se trouve à l'adresse 232. Il s'agit d'une adresse qui sert de destination, car le `INLINED_SKIP_NEXT_SEQ` se comporte comme un `GOTO`. Cette instruction sert à sauter par-dessus la séquence dupliquée, celle qui suit la séquence courante. Il faut dupliquer la séquence qui suit le *invoke* dans le site d'invocation original si jamais celle-ci est incluse dans la super-instruction. En effet, lors de l'exécution du site d'invocation deux choses peuvent se produirent et elles sont décrites dans les paragraphes qui suivent.

A) L'invocation se termine par IRETURN (chemin inclus invalide)

Lors de l'exécution du point de vérification `INLINED_IF_ICMPLT` (adresse 225) dans la super-instruction, si nous détectons que le chemin est invalide, nous continuons l'exécution à l'adresse 218 dans le code original de la méthode appelée. La méthode se terminera par le `IRETURN` (à l'adresse 221) qui réajustera le `pc` à la valeur de retour qui est stockée dans le bloc d'activation de la méthode appelante. Dans ce cas-ci, le `pc` sera ajusté de manière à ce que le flot continue son exécution à l'adresse 234 (la séquence dupliquée). Il est impossible

de reprendre l'exécution de la méthode en plein milieu de l'implantation de la séquence qui se trouve à l'adresse 225. Lors d'une répartition, on est contraint de sauter au début d'une séquence. Pour cette raison, on plaçe une copie de la séquence (adresse 234) à la suite du code inclus.

B) L'invocation se termine par INLINED_IRETURN (chemin inclus valide)

Lors de l'exécution du point de vérification `INLINED_IF_ICMPLT` (adresse 225) dans la super-instruction, si on détecte que le chemin est valide, on continue l'exécution normalement dans le corps de la super-instruction de l'adresse 225. L'exécution de la méthode appelée se terminera par le `INLINED_IRETURN` (adresse 225,) qui ne fait aucune modification au `pc`, car celui-ci est déjà au bon endroit puisque le code est inclus. Le `INLINED_IRETURN` détruira le bloc d'activation de la méthode appelée et placera sur la pile des opérandes de la méthode appelante la valeur de retour. Ensuite, les codes d'opération `IADD` et `ISTORE_1` sont exécutés au sein même de la super-instruction. Ensuite, il faut sauter par-dessus la copie de la séquence qui se trouve à l'adresse 234 en utilisant le `INLINED_SKIP_NEXT_SEQ`. Ce code d'opération s'assure que lors de la répartition à la fin de la super-instruction, on continuera l'exécution à l'adresse 235. Nous aurons donc sauté par-dessus le doublon.

Nous effectuons la duplication d'une séquence seulement lorsque ces deux conditions sont respectées :

1. La séquence suit le site d'invocation
2. La séquence est incluse dans la séquence contenant le code inclus.

Cette duplication ajoute un minime coût additionnel à l'exécution de la super-instruction que nous créons, car nous devons exécuter le `INLINED_SKIP_NEXT_SEQ`. L'implantation du `INLINED_SKIP_NEXT_SEQ` ne comporte qu'une seule ligne de code (`pc = pc->addr`). Nous affectons au `pc` la valeur qui se trouve à l'emplacement référencé par le `pc`. C'est lors de la répartition que le véritable saut par-dessus la séquence dupliquée se fait.

2.3 Conclusion

Ce chapitre nous a permis d'établir clairement les objectifs de notre système d'inclusion des méthodes. Nous voulons diminuer le nombre de répartitions exécutées dans l'interpréteur linéaire inclusif de SableVM. Nous comptons effectuer cette diminution en augmentant la longueur moyenne des séquences appelées par l'interpréteur. Notre mécanisme d'inclusion partielle nous offre la possibilité d'augmenter la longueur moyenne des séquences appelées, en substituant les codes d'opération de type *invoke* par le chemin le plus fréquenté à l'intérieur de la méthode invoquée.

Nous déterminons ce chemin grâce au système de profilage que nous avons mis en place dans SableVM. Nous devons insérer des points de vérifications dans le chemin que nous incluons afin de détecter, lors de l'exécution, si le chemin inclus correspond bien au flot d'exécution. Dans le cas où le chemin est invalide, nous sautons au bon endroit dans le code original de la méthode appelée afin d'y poursuivre l'exécution et ceci à coût presque nul. Lors des prochains passages sur le site d'invocation optimisé, si l'identité de la méthode incluse est la bonne et que le chemin inclus est valide, nous diminuons le nombre de répartitions et sinon, nous n'en exécutons pas plus que dans le cas où le site n'aurait jamais été optimisé.

Chapitre 3

Système de profilage

Dans ce chapitre, nous allons nous concentrer sur le mécanisme de profilage que nous avons conçu et implanté dans SableVM. Nous mettrons en lumière la nécessité de profiler une méthode avant son inclusion et nous donnerons les raisons qui justifient ce profilage. Nous expliquerons, de manière détaillée, le fonctionnement interne du système de profilage. Nous verrons comment le système est initié et comment se fait la détermination des méthodes à profiler. Une section exposera comment nous recueillons les informations de profilage. Par la suite, nous discuterons des solutions que nous avons trouvées pour résoudre les problèmes reliés aux exceptions, aux sous-routines et à la synchronisation. Finalement, nous présenterons les limites du système et nous expliquerons les choix que nous avons faits quant à la précision et à la qualité du profilage mis en place.

3.1 Utilité du profilage

Comme mentionné au chapitre 3, notre objectif ultime est de pouvoir construire une seule séquence (super-instruction) avec tout le code de la méthode incluse. Pour ce faire, il est impératif que le code inclus ne représente qu'un seul bloc de base. C'est pourquoi nous choisisons un chemin d'exécution dans la méthode appelée lors de l'inclusion. De cette façon, les instructions que nous inclurons sont exemptes de sauts et de branchements. Le chemin choisi a obligatoirement été visité au moins une fois et ceci nous assure qu'il ne comporte pas de sauts vers une séquence de préparation.

Pour nous aider à déterminer le chemin le plus fréquenté, nous avons outillé SableVM d'un système de profilage. Nous avons besoin d'un minimum d'informations sur le comportement du code ciblé pour pouvoir sélectionner le chemin le mieux possible. Le système de profilage nous indique, pour chaque branchement, quelle est la destination la plus fréquente. Ces informations nous permettent de sélectionner un chemin assurément déjà fréquenté, éliminant ainsi la possiblité de rencontrer un saut vers une séquence de préparation. La figure 2.2 donne un exemple de chemin qui a été assujetti au processus d'inclusion. Le profilage sert à choisir le chemin qui sera inclus. Puisque nous n'incluons pas complètement la méthode appelée, nous considérons notre inclusion comme partielle (*partial inlining*). Notre système est donc un système d'inclusion partielle des méthodes.

3.2 Initiation et arrêt du profilage

Dans cette section, nous expliquons comment est initié le profilage et comment nous choisissons les méthodes à profiler. Nous ne profilons pas systématiquement toutes les méthodes, car nous ne voudrons pas nécessairement faire l'inclusion de toutes celles-ci. Il est inutile de profiler une méthode qui ne sera jamais incluse et il est inutile de faire l'inclusion d'une méthode qui sera rarement appelée.

3.2.1 Les méthodes *non-inlinable*

Comme nous l'avons déjà expliqué (section 1.3), le premier appel d'une méthode provoque sa préparation. Dès ce moment, nous pouvons choisir de marquer la méthode en question comme *inlinable*[1] ou pas. Par exemple, les initialiseurs de classes ou d'interfaces, qui ne s'exécutent qu'une seule fois, sont marqués comme étant *non-inlinable*[2] dès la préparation. Un initialisateur de classe est une méthode qui se nomme `<clinit>` et qui correspond à un bloc d'initialisation statique pour une classe Java. Puisque ce type de méthode n'est exécutée qu'une seule fois à l'initialisation de la classe, il est inutile de marquer celle-ci comme propice à l'inclusion.

[1] *inlinable :* Terme anglais qui précise qu'une méthode est autorisée à être incluse.
[2] *non-inlinable :* Terme anglais qui précise qu'une méthode n'est pas autorisée à être incluse.

Le système possède aussi un paramètre qui permet de spécifier la longueur maximale d'une méthode autorisée à être incluse. La taille d'une méthode constitue donc une contrainte qui peut la rendre *non-inlinable*. À partir de maintenant, quand nous parlerons d'une méthode *inlinable*, nous ferons référence à une méthode qui a été marquée comme *inlinable* au moment de sa préparation.

Seulement les initialisateurs de classes et les méthodes dont la taille du code est supérieure à la limite permise sont marqués comme étant *non-inlinable* par notre système au moment de la préparation des méthodes. L'ajout de contraintes supplémentaires aurait été possible, mais nous avons choisi de nous limiter à celles-ci.

3.2.2 Seuils d'initiation et d'arrêt du profilage

Nous voulons seulement faire le profilage et l'inclusion des méthodes chaudes. Seulement les méthodes *inlinable* exécutées un minimum de fois sont profilées. Une méthode devient chaude lorsque son nombre d'appels franchit un certain seuil. Le système possède trois paramètres qui servent de seuils de déclenchement et d'arrêt du profilage et d'autorisation d'inclusion de la méthode. Ces paramètres sont :

1. Seuil d'initiation du profilage
2. Seuil d'arrêt du profilage
3. Seuil d'autorisation à l'inclusion

Le premier paramètre est le seuil d'initiation du profilage (*method_profiling_start_threshold*), qui permet de spécifier le nombre minimum de fois qu'une méthode doit avoir été appelée avant que nous commencions à en faire le profilage. Il est important de spécifier que ce nombre d'appels est relatif à la méthode appelée et non pas au site d'invocation. Nous implantons donc ce seuil grâce à un compteur qui est associé à la méthode appelée et non pas au site d'invocation. Le second paramètre est le seuil d'arrêt du profilage (*method_profiling_stop_threshold*), qui permet de spécifier jusqu'à quand nous voulons effectuer le profilage. Le seuil d'arrêt est implanté par le même compteur utilisé pour le seuil d'initiation du profilage. Ce compteur sert à derterminer à la fois quand initier et arrêter le profilage. Le troisième paramètre (*method_invoke_start_threshold*) est relié à l'inclusion

d'une méthode dans un site d'invocation en particulier. Il s'agit du nombre de fois que le flot d'exécution doit passer sur un site d'invocation avant d'être autorisé à faire l'inclusion du `INVOKE` à cet endroit. Ce seuil est implanté grâce à un compteur associé au site d'invocation où on veut potentiellement faire l'inclusion d'une méthode.

Il est possible de passer des valeurs pour ces seuils en paramètre au système, au moment du démarrage de la machine virtuelle. Une fois la machine virtuelle démarrée, ces valeurs globales resteront les mêmes tout au long de l'exécution et seront les mêmes pour toutes les méthodes. Par exemple, si on prend respectivement les valeurs 5, 15 et 15, on commencera le profilage de toute méthode *inlinable* lors de son cinquième appel et le profilage de cette méthode se terminera lors de son quinzième appel. On sera autorisé à faire l'inclusion d'un `INVOKE` (qui appelle cette méthode) seulement lorsque on aura passé un minimum de 15 fois sur celui-ci. Le système ne permet pas l'inclusion d'une méthode avant que son profilage soit terminé. Dans cet exemple, les méthodes sont profilées pendant une vingtaine d'appels avant d'être sujettes à l'inclusion.

3.2.3 Ajout d'instructions spécialisées

Pour comprendre comment nous utilisons les valeurs de ces paramètres afin de mettre en marche le système, il faut se rappeler que les instructions existantes au sein de la machine virtuelle ne sont pas obligatoirement les mêmes que celles définies dans la spécification de la machine virtuelle Java [LY99] et que l'on retrouve dans le fichier *.class*. Une fois le chargement du fichier *.class* effectué, nous avons la liberté de modifier le code octet, sans toutefois modifier la sémantique du code en question. Il est de pratique courante, au sein des machines virtuelles Java, d'ajouter des nouvelles instructions au jeu d'instruction ou d'avoir plusieurs versions d'une même instruction pour accélérer l'exécution du code [ETKW06] [GH03].

Les instructions INITIAL_INVOKE<X>

Nous avons ajouté des instructions spécialisées[3] pour nous aider à articuler le mécanisme de profilage et d'inclusion. Reprenons l'exemple de la figure 1.3. Comme nous l'avons expliqué à la section 1.4, la première fois qu'on passe sur un site d'invocation d'un appel statique, nous rencontrons un `GOTO` vers un `PREPARE_INVOKESTATIC` plutôt qu'un `INVOKESTATIC`. Ce n'est qu'une fois que la classe qui contient la méthode est chargée et que la méthode est préparée et invoquée qu'on effectue le `REPLACE` qui va remplacer le saut vers la séquence de préparation par le `INVOKESTATIC`. Nous avons apporté une légère modification à cette façon de faire pour incorporer notre système.

La figure 3.1 reprend l'exemple de la figure 1.3 mais en incorporant notre système. Notre système se comporte de manière similaire à ce qui est présenté à la figure 1.3, sauf que le `REPLACE` de la ligne 4c va remplacer le `GOTO`, non pas par un `INVOKESTATIC`, mais par un `INITIAL_INVOKESTATIC`. Donc la deuxième fois que le flot d'exécution passera sur le site d'invocation, on exécutera le `INITIAL_INVOKESTATIC`. Il s'agit d'une instruction que nous avons ajoutée au jeu d'instructions déjà existant, qui se comporte exactement comme un `INVOKESTATIC`, mais qui vérifie avant d'exécuter la méthode si celle-ci est *inlinable*. Si ce n'est pas le cas, elle va se substituer elle-même par un `INVOKESTATIC`. Si la méthode appelée est *inlinable*, elle va se substituer elle-même par un `COUNTING_INVOKESTATIC`. Il s'agit ici d'une substitution atomique. Ces trois types de `INVOKE` (`INITIAL_INVOKE`, `COUNTING_INVOKE`, `INVOKE`) ont tous les mêmes opérandes dans le tableau de code. La seule chose qui doit être modifiée pour passer d'une à l'autre est l'instruction elle-même et cette instruction n'occupe qu'un mot mémoire dans le tableau de code. La figure 3.1 illustre le tableau de code aux différents passages sur le site d'invocation. Dans cette figure, le site d'invocation se trouve à l'adresse 18. On voit qu'après le premier appel, le `GOTO` a été remplacé par un `INITIAL_INVOKESTATIC`. Lors du deuxième appel, on remplacera le `INITIAL_INVOKESTATIC` par un `INVOKESTATIC` ou un `COUNTING_INVOKESTATIC`, selon que la méthode est *inlinable* ou non.

[3]Nous utilisons le terme `INITIAL_INVOKE<X>` pour faire référence aux quatre types de `INITIAL_INVOKE` (`INITIAL_INVOKESTATIC, INITIAL_INVOKESPECIAL, INITIAL_INVOKEVIRTUAL, INITIAL_INVOKEINTERFACE`)

Avant le 1er appel	**Après le 1er appel**
[...] --- **18: [GOTO]** **1c: (3c) <-- @sequence prep** **20: (method_nb_args)** **24: (@method_info)** **28: (@stack_gc_map)** --- 2c: [IADD] 30: [ISTORE_1] 34: [ILOAD_1] 38: [IRETURN] 3c: [PREPARE_INVOKESTATIC] 40: (20) 44: (@methodref_info) 48: (@stack_gc_map) 4c: [REPLACE] 50: (18) 54: (INITIAL_INVOKESTATIC) 58: [GOTO] 5c: (2c)	[...] --- **18: [INITIAL_INVOKESTATIC]** **1c: (3c)** **20: (method_nb_args)** **24: (@method_info)** **28: (@stack_gc_map)** --- 2c: [IADD] 30: [ISTORE_1] 34: [ILOAD_1] 38: [IRETURN] 3c: [PREPARE_INVOKESTATIC] 40: (20) 44: (@methodref_info) 48: (@stack_gc_map) 4c: [REPLACE] 50: (18) 54: (INITIAL_INVOKESTATIC) 58: [GOTO] 5c: (2c)
Après le 2ième appel (méthode *inlinable*)	**Après le 2ième appel** **(**méthode *non inlinable*)
[...] --- **18: [COUNTING_INVOKESTATIC]** **1c: (2) <-- compteur** **20: (method_nb_args)** **24: (@method_info)** **28: (@stack_gc_map)** --- 2c: [IADD] [...]	[...] --- **18: [INVOKESTATIC]** **1c: (3c)** **20: (method_nb_args)** **24: (@method_info)** **28: (@stack_gc_map)** --- 2c: [IADD] [...]

Figure 3.1 – Les trois premiers états d'un site d'appel (*méthode statique*).

Nous avons besoin de l'instruction `INITIAL_INVOKESTATIC` pour vérifier si la méthode est *inlinable* ou non au moment de l'appel de la méthode, car il nous est impossible de le faire avant. Dû au chargement paresseux des classes, il n'est pas possible, au moment de la préparation du tableau de code de la méthode appelante, de savoir si la méthode appelée est *inlinable* ou non. Au moment de la préparation, la classe contenant la méthode appelée n'a peut-être pas été chargée. Nous devons donc attendre au moment de l'exécution de celle-ci pour déterminer si la méthode est *inlinable* et si nous devons compter son nombre d'exécutions en vue d'un éventuel profilage.

Si la méthode appelée n'est pas *inlinable*, le site d'invocation se comportera normalement lors des prochains passages, car le `INITIAL_INVOKE` aura été remplacé par une

INVOKE. Dans le cas contraire, le COUNTING_INVOKE sera exécuté.

Les instructions COUNTING_INVOKE<X>

Le COUNTING_INVOKE[4] se comporte comme un INVOKE normal, sauf qu'on lui a ajouté quatre fonctions supplémentaires. Premièrement, il incrémente un compteur associé à la méthode appelée. Ce compteur indique le nombre de fois que la méthode a été invoquée de tous les sites d'invocations. C'est grâce à lui que nous pouvons savoir quand démarrer et arrêter le profilage de la méthode. Ce compteur est situé dans la structure relative à la méthode appelée (*method_info*).

Deuxièmement, l'instruction COUNTING_INVOKE incrémente aussi une de ses opérandes dans le tableau de code qui sert à savoir combien de fois le flot d'exécution a passé à cet endroit. Il est possible de voir à la figure 3.1 que le premier paramètre du COUNTING_-INVOKESTATIC est un entier qui sert de compteur au site d'invocation. Contrairement au compteur d'invocations de la méthode, celui-ci est situé à même le code, car chaque site possède son propre compteur. L'utilisation d'un entier est suffisante pour les appels de type INVOKESTATIC et INVOKESPECIAL puisque la méthode appelée ne dépend pas du type de l'objet courant. Les appels de type INVOKEVIRTUAL et INVOKEINTERFACE nécessitent une information supplémentaire pour que l'inclusion soit possible. Pour ces deux types d'appels, l'identité de la méthode appelée est déterminée par le type de l'objet sur lequel la méthode est appliquée. Il est donc nécessaire, pour effectuer l'inclusion, de savoir quelle méthode nous devrons inclure à cet endroit. Au lieu d'être un entier, le premier paramètre des instructions COUNTING_INVOKEVIRTUAL et COUNTING_INVOKEINTERFACE est un pointeur vers une mini structure qui contient le compteur du site (un entier) et une référence vers la méthode la plus souvent appelée à cet endroit. De cette façon, nous pourrons déterminer quelle méthode inclure à cet endroit lors de l'inclusion. L'utilisation de cette mini structure permet de substituer le INITIAL_INVOKE par un COUNTING_INVOKE sans avoir à recopier tout le tableau de code pour un simple paramètre supplémentaire. En plus d'incrémenter ces deux compteurs, le COUNTING_INVOKE vérifie si nous avons atteint

[4]Nous utilisons le terme COUNTING_INVOKE<X> pour faire référence aux quatre types de COUNTING_INVOKE (COUNTING_INVOKESTATIC, COUNTING_INVOKESPECIAL, COUNTING_INVOKEVIRTUAL, COUNTING_INVOKEINTERFACE)

le seuil de démarrage du profilage. Si jamais nous avons atteint le seuil de démarrage du profilage de la méthode appelée, l'instruction appelle la routine qui démarre le profilage de la méthode avant d'effectuer son appel. La quatrième fonctionnalité du `COUNTING_INVOKE` est que celui-ci vérifie si nous avons atteint le seuil qui permet l'inclusion de ce site. Si c'est le cas, l'instruction provoquera l'inclusion de la méthode appelée à cet endroit. Nous reviendrons sur ce point au chapitre 5. Évidemment, nous avons ajouté les nouvelles instructions pour chaque type d'invocations (voir annexe B).

3.2.4 Activation et arrêt du profilage

SableVM comprend deux routines très importantes pour notre système, *start_the_world()* et *stop_the_world()*. Lorsqu'un *thread* appelle *stop_the_world()*, ceci a pour effet de stopper tous les *threads* qui exécutent du code Java et empêche les autres de commencer à en exécuter. L'appel de la routine opposée, *start_the_world()*, a l'effet contraire c'est-à-dire que tous les *threads* arrêtés précédemment sont autorisés à repartir. Lorsque nous activons le profilage d'une méthode, la première chose à faire est d'arrêter le monde, car nous nous apprêtons à modifier le tableau de code d'une méthode. Il est donc primordial que tous les autres *threads* soient arrêtés. Nous voulons prévenir qu'un *thread* exécute du code obsolète ou inconsistant. Nous reviendrons sur l'aspect de la synchronisation plus loin dans ce chapitre.

Une fois le monde arrêté, nous construisons un second tableau de code pour la méthode à profiler. Il s'agit ici d'un tableau de code profilant. À tout moment, une méthode possède un pointeur *code* qui pointe soit vers le tableau de code normal, soit vers le tableau de code profilant. Une fois le tableau de code profilant construit, on doit s'assurer que le pointeur *code* de la méthode référence le code profilant. On met ensuite à jour toutes les invocations de la méthode pour qu'elles continuent leurs exécutions au bon endroit dans le code profilant. On réajuste donc le `pc` de toutes les invocations de la méthode vers le code profilant. En terminant, on redémarre le monde et le profilage de la méthode est activé.

Le processus d'arrêt du profilage est similaire. Lorsque nous détectons que nous avons atteint le seuil d'arrêt du profilage pour une méthode, nous arrêtons le monde, réajustons le pointeur *code* de la méthode vers le code normal et redémarrons le monde. Évidemment,

dans ce cas, nous devons aussi ajuster toutes les invocations de la méthode pour qu'elles continuent leur exécution dans le code normal (non profilant).

3.3 Construction du code profilant

Nous construisons le tableau de code profilant pour une méthode seulement au moment où nous démarrons son profilage. Chaque méthode dont nous faisons le profilage possède deux tableaux de code, le code normal et le code profilant. Ceci simplifie beaucoup l'arrêt du profilage. Pour stopper le profilage, il est seulement nécessaire de modifier le pointeur de code de la méthode pour qu'il référence de nouveau le code normal. Aucune manipulation sur le code n'est nécessaire. Nous conservons toujours le tableau de code profilant une fois le profilage terminé, car c'est lui qui permet de construire le chemin lors de l'inclusion.

Pour construire le code profilant, nous allouons un nouveau tableau de code et nous y copions toutes les instructions et leurs paramètres provenant du code normal. Les seules instructions que nous ne copions pas intégralement sont les branchements et les quatre types de `COUNTING_INVOKE`.

3.3.1 Ajout d'instructions profilantes

Nous recueillons les informations de profilage à même le tableau de code profilant. Pour nous permettre d'arriver à un tel résultat, nous avons ajouté au jeu d'instructions de SableVM des instructions profilantes (voir annexe B). Dans la construction du code profilant, nous remplaçons les instructions de branchement normal par des versions profilantes de celles-ci. Pour chaque instruction de branchement, nous avons créé une instruction de branchement équivalente, mais profilante. Les instructions qui possèdent le préfixe `PROFILING_` se retrouvent uniquement dans le code profilant.

La figure 3.2 liste les instructions de branchement normales et les instructions de branchement profilantes correspondantes.

L'implémentation des instructions `PROFILING_IF<X>` est identique à celle des instructions `IF<X>`, sauf qu'elle incrémente un compteur pour indiquer la destination la plus fréquente. Ce compteur est situé à même les opérandes de l'instruction dans le tableau de

Instructions normales	Instructions profilantes
COUNTING_INVOKEVIRTUAL	PROFILING_INVOKEVIRTUAL
COUNTING_INVOKESPECIAL	PROFILING_INVOKESPECIAL
COUNTING_INVOKESTATIC	PROFILING_INVOKESTATIC
COUNTING_INVOKEINTERFACE	PROFILING_INVOKEINTERFACE
IFEQ	PROFILING_IFEQ
IFNE	PROFILING_IFNE
IFLT	PROFILING_IFLT
IFGE	PROFILING_IFGE
IFGT	PROFILING_IFGT
IFLE	PROFILING_IFLE
IF_ICMPEQ	PROFILING_IF_ICMPEQ
IF_ICMPNE	PROFILING_IF_ICMPNE
IF_ICMPLT	PROFILING_IF_ICMPLT
IF_ICMPGE	PROFILING_IF_ICMPGE
IF_ICMPGT	PROFILING_IF_ICMPGT
IF_ICMPLE	PROFILING_IF_ICMPLE
IF_ACMPEQ	PROFILING_IF_ACMPEQ
IF_ACMPNE	PROFILING_IF_ACMPNE
IFNULL	PROFILING_IFNULL
IFNONNULL	PROFILING_IFNONNULL
TABLESWITCH	PROFILING_TABLESWITCH
LOOKUPSWITCH	PROFILING_LOOKUPSWITCH

Figure 3.2 – Les instructions profilantes.

code. Les instructions profilantes possèdent donc un ou des paramètres supplémentaires. Toutes les instructions `IF<X>` possèdent deux destinations possibles. Si la condition est vraie nous effectuons le branchement et sinon nous continuons l'exécution à l'instruction suivante. Nous avons choisi d'utiliser un seul compteur, au lieu de deux, afin de conserver l'information sur le choix le plus fréquent. Si le branchement est effectué, le compteur est incrémenté et sinon il est décrémenté. Ceci permet d'utiliser moins d'espace pour le tableau de code, mais cela permet surtout de sauver un `pc++`, qui serait nécessaire pour sauter par-dessus le deuxième compteur à chaque passage sur un branchement profilant. De cette façon, si le compteur est positif, nous savons que c'est le branchement qui est le plus fréquent. Si le compteur est négatif, nous savons que la plupart du temps le branchement n'est pas effectué.

Les instructions `TABLESWITCH` et `LOOKUPSWITCH` possèdent, quant à elles, un compteur pour chaque destination possible c'est-à-dire un pour chaque *case* et un pour le *default case*. De cette façon, il sera aisé de déterminer par la suite la destination la plus fréquente. La figure 3.3 montre la différence dans le tableau de code entre un `IF`, un `TABLESWITCH`, et un `LOOKUPSWITCH` normal et leur équivalent profilant. Les versions profilantes possèdent

Version normale	Version profilante
IF<X>: ... [IF<X>] (@destination) ...	**PROFILING_IF<X>:** ... [IF<X>] (@destination) (compteur) ...
TABLESWITCH[1] (3 cases: 22,23,24): ... [TABLESWITCH] (@default_destination) (22) <- valeur min des cases (24) <- valeur max des cases (@destination pour case=22) (@destination pour case=23) (@destination pour case=24) ...	**PROFILING_TABLESWITCH:** ... [PROFILING_TABLESWITCH] (@default_destination) (22) <- valeur min des cases (24) <- valeur max des cases (compteur pour default) (@destination pour case=22) (compteur) (@destination pour case=23) (compteur) (@destination pour case=24) (compteur) ...
LOOKUPSWITCH[2] (3 cases: 2,31,81): ... [LOOKUPSWITCH] (@default_destination) (3) <- nombre de cases (2) (@destination pour case=2) (31) (@destination pour case=31) (81) (@destination pour case=81) ...	**PROFILING_LOOKUPSWITCH:** ... [PROFILING_LOOKUPSWITCH] (@default_destination) (3) <- nombre de cases (compteur_pour_default) (2) (@destination pour case=2) (compteur) (31) (@destination pour case=31) (compteur) (81) (@destination pour case=81) (compteur) ...

1 Ce *tableswitch* correspond en Java à un switch avec 3 cases dont les valeurs sont: 22, 23, 24.
2 Ce *lookupswitch* correspond en Java à un switch avec 3 cases. Dont les valeurs sont: 2, 31, 81.

Figure 3.3 – Exemple des trois types d'instructions de profilage.

des compteurs en plus parmi leurs opérandes. La figure 3.4 illustre les différences, au point de vue de l'implémentation, entre le `TABLESWITCH` et le `PROFILING_TABLESWITCH`. La seule différence est que dans la version profilante, on y incrémente les compteurs de profilage.

Comme expliqué précédemment, les `COUNTING_INVOKE` incrémentent un compteur relatif au nombre de fois que le flot d'exécution est passé sur ce *invoke*. Puisque nous voulons conserver trace, dans le code normal, des passages qui seront fait sur ce `INVOKE` lors du profilage et que le profilage se fait par l'entremise d'un autre tableau de code, nous avons créé le `PROFILING_INVOKE` qui remplace les `COUNTING_INVOKE` dans le code profilant. Les

```
case TABLESWITCH:
{
  _svmt_code *default_dest = (pc++)->addr;
  jint low = (pc++)->jint;
  jint high = (pc++)->jint;
  _svmt_code *table = pc;
  jint index = stack[--stack_size].jint;

  if (index < low || index > high)
  {
    pc = default_dest;
  }
  else
  {
    pc = table[index - low].addr;
  }
}
```

```
case PROFILING_TABLESWITCH:
{
  _svmt_code *default_dest = (pc++)->addr;
  jint low = (pc++)->jint;
  jint high = (pc++)->jint;
  jint index = stack[--stack_size].jint;

  if (index < low || index > high)
  {
    /* incrémentation compteur default */
    (pc->jint)++;
    pc = default_dest;
  }
  else
  {
    jint ind = (index - low) * 2;
    pc++;

    /* incrémentation compteur case */
    (pc[ind + 1].jint)++;
    pc = pc[ind].addr;
  }
}
```

Figure 3.4 – Implémentations du `TABLESWITCH` et du `PROFILING_TABLESWITCH`.

`COUNTING_INVOKE` existent seulement dans les tableaux de code normaux. Comme toutes les instructions préfixées par `PROFILING_`, le `PROFILING_INVOKE` existe seulement au sein de code profilant. Il se comporte comme le `COUNTING_INVOKE`, sauf que le compteur qu'il incrémente n'est pas un de ses paramètres, mais bien le paramètre du `COUNTING_INVOKE` qui lui correspond dans le code normal (voir figure 3.5). De cette façon, quand nous reviendrons au code normal, nous aurons un compte plus exact du nombre de passage sur l'invocation en question. Nous n'aurons pas perdu d'informations pendant le profilage du code.

3.3.2 Correction des adresses

Une fois que nous avons copié le code normal dans le nouveau tableau de code profilant en substituant chaque code d'opération de branchement par sa version profilante, il est nécessaire d'ajuster tous les opérandes qui sont des adresses absolues. Puisque nous avons effectué un copiage presque systématique du code original, certaines instructions possèdent des opérandes qui doivent être modifiés. Le `GOTO`, par exemple, possède comme opérande une adresse absolue qui lui sert de destination. Nous ne voulons pas qu'en cours d'exécution du code profilant, le flot d'exécution puisse sauter dans le code normal. Si on regarde

COUNTING_INVOKESTATIC
Dans le code normal:

```
[…]
18: [COUNTING_INVOKESTATIC]
1c: (2) ← compteur
20: (method_nb_args)
24: (@method_info)
28: (@stack_gc_map)
[…]
```

PROFILING_INVOKESTATIC
correspondant dans le code profilant:

```
[…]
50: [PROFILING_INVOKESTATIC]
54: (1c) ← adresse du compteur
58: (method_nb_args)
5c: (@method_info)
60: (@stack_gc_map)
[…]
```

Figure 3.5 – Mise à jour du compteur dans le tableau de code normal par le `PROFILING_INVOKESTATIC`.

l'exemple réel de code normal et de code profilant, qui se trouve en annexe C, on constate que les adresses qui sont des opérandes ont été ajustées dans le code profilant pour que le flot d'exécution reste toujours dans le code profilant.

3.4 Exceptions vs profilage

3.4.1 Table d'exceptions

Chaque méthode Java qui possède un ou des blocs *try*{} *catch*{} dans son code possède une table d'exceptions qui lui est associée [LY99]. La table d'exceptions sert à implanter les blocs *try/catch* au niveau du code octet. Une table d'exceptions est constituée des champs *pc_start*, *pc_end*, *handler* et *type*. Les champs *pc_start* et *pc_end* délimitent le début et la fin du bloc *try* en utilisant les adresses des bornes dans le tableau de code. Le champ *handler* est l'adresse de début du code qui représente le bloc *catch* dans le code octet. Le champ *type* représente le type de l'exception qui concerne la ligne courante dans la table. La figure 3.6 illustre le code Java, le code octet normal et la table d'exceptions du code octet normal pour une méthode donnée.

3.4.2 Gestion d'une exception

En cours d'exécution, si une exception se produit, on prend la valeur courante du `pc` et on parcourt en ordre la table d'exceptions à la recherche d'une ligne où le `pc` sera compris

```
public static int m1(String val) {
  int result ;
  try {
    result = Integer.parseInt(val);
    if(result > 0) {
      result *=10 ;
    }
    else {
      result *= -10 ;
    }
  }
  catch(NumberFormatException e) {
    return -1 ;
  }
  return result ;
}
```

Normal :

```
be4: [ALOAD_0]
be8: [INVOKESTATIC]
bec: (c58)
bf0: (method_nb_args)
bf4: (@method_info)
bf8: (@stack_gc_map)
bfc: [ISTORE_2]
c00: [ILOAD_2]
c04: [IFLE]
c08: (c28)

c0c: [ILOAD_2]
c10: [LDC_INTEGER]
c14: (10)
c18: [IMUL]
c1c: [ISTORE_2]
c20: [GOTO]
c24: (c3c)
c28: [ILOAD_2]
c2c: [LDC_INTEGER]
c30: (-10)
c34: [IMUL]
c38: [ISTORE_2]
c3c: [GOTO]
c40: (c50)
c44: [ASTORE_1]
c48: [ICONST_M1]
c4c: [IRETURN]
c50: [ILOAD_2]
c54: [IRETURN]
```

Profilant :

```
d14: [ALOAD_0]
d18: [INVOKESTATIC]
d1c: (c58)
d20: (method_nb_args)
d24: (@method_info)
d28: (@stack_gc_map)
d2c: [ISTORE_2]
d30: [ILOAD_2]
d34: [PROFILING_IFLE]
d38: (d5c)
d3c: (0) <-- compteur
d40: [ILOAD_2]
d44: [LDC_INTEGER]
d48: (10)
d4c: [IMUL]
d50: [ISTORE_2]
d54: [GOTO]
d58: (d70)
d5c: [ILOAD_2]
d60: [LDC_INTEGER]
d64: (-10)
d68: [IMUL]
d6c: [ISTORE_2]
d70: [GOTO]
d74: (d84)
d78: [ASTORE_1]
d7c: [ICONST_M1]
d80: [IRETURN]
d84: [ILOAD_2]
d88: [IRETURN]
```

Table des exceptions du code normal:

pc_start	pc_end	handler	type
be4	c3c	c44	java/lang/NumberFormatException

Table des exceptions du code profilant:

pc_start	pc_end	handler	type
d14	d70	d78	java/lang/NumberFormatException

Figure 3.6 – Code normal, code profilant et tables d'exceptions.

dans l'intervalle représenté par *pc_start* et *pc_end*. Si on trouve une ligne qui satisfait cette contrainte, il faut vérifier que le type de l'exception qui a été lancée est compatible avec le champ type de la ligne trouvée. Le champ type de cette ligne doit être le même que celui de l'exception lancée. Si ce n'est pas le cas, le champ type doit être une super classe de l'exception lancée. Si cette contrainte est satisfaite, on vide la pile des opérandes de la méthode courante, on place à son sommet la référence vers l'instance de l'exception courante et on continue l'exécution à l'adresse qui se trouve dans le champ *handler*.

Si aucune entrée dans la table ne satisfait ces exigences, nous dépilons le bloc d'activation de la méthode courante et recommençons la même procédure avec la table d'exception de la méthode associée au bloc d'activation suivant. Nous répétons l'opération jusqu'à ce que nous trouvions la méthode, dans la pile d'appels de méthode, qui pourra gérer l'exception. Si aucune méthode pouvant gérer l'exception n'est trouvée, le système arrête l'exécution en signalant une erreur.

3.4.3 Exceptions dans l'exécution du code profilant

À la lumière des explications ci-haut, il devient évident qu'un problème se pose si une exception se produit dans l'exécution du code profilant. Nous ne pouvons utiliser la table d'exceptions déjà présente, puisque les adresses qu'elle contient font référence au code normal. Ceci aurait pour effet qu'une exception survenant dans le code profilant ne serait jamais attrapée.

Nous pallions à ce problème en créant une deuxième table d'exceptions que nous associons au code profilant (voir figure 3.6). C'est après avoir complété la construction du tableau de code profilant que nous construisons la table d'exceptions qui lui est associée. Par la suite, nous n'avons qu'à modifier le pointeur *exception_table* de la méthode pour qu'il pointe vers la bonne table d'exception. De manière similaire, quand on arrête le profilage, on n'a qu'à rétablir le pointeur vers la table d'exceptions du code normal. Cette opération est simple et a un coût presque nul.

3.5 Mise à jour des invocations

3.5.1 Le *pc* dans les blocs d'activation

À tout moment, pour une méthode donnée, il n'existe qu'un seul tableau de code valide. Nous avons choisi d'agir ainsi afin de simplifier le système et de prévenir les inconsistances au sein de la machine virtuelle. Il est impossible, par exemple, qu'un *thread* exécute une méthode `A` normale, pendant qu'un autre *thread* exécute la version profilante de cette même méthode `A`. Lorsque le profilage d'une méthode est activé, toutes les invocations de cette méthode participent au profilage. Lorsque le profilage est terminé, toutes les invocations reviennent au code normal de cette méthode.

Afin de mettre en place la propriété d'unicité du code actif pour une méthode donnée, il est nécessaire, avant de redémarrer le monde (*start_the_world*), de mettre à jour toutes les invocations de la méthode que nous venons d'altérer. Nous avons vu au chapitre 2 que chaque *thread* possède sa propre pile de blocs d'activation. Chaque bloc d'activation contient une sauvegarde du contexte de la méthode auquel il est associé. Grâce à ces informations, il est facile de mettre à jour toutes les invocations pour une méthode donnée.

L'algorithme de mise à jour consiste en partie à parcourir la pile des blocs d'activation pour chacun des *threads*. Chaque fois que nous rencontrons un bloc qui est associé à la méthode ciblée, nous modifions le `pc` sauvegardé dans le bloc pour qu'il pointe vers le nouveau tableau de code. Cet algorithme permet de mettre à jour les invocations de la méthode qui sont empilées dans la pile des blocs d'activation de chaque *thread*. L'algorithme ne permet pas de mettre à jour le `pc` de la méthode qui était en cours d'exécution dans chacun des *threads* avant l'arrêt du monde.

3.5.2 Rafraîchissement des *pc* dans les méthodes actives

Lorsqu'un *thread* s'arrête, suite à un appel par un autre *thread* de la procédure *stop_the_world*, celui-ci va d'abord sauvegarder son `pc` dans son bloc d'activation qui est au sommet de la pile. Même si nous mettons à jour la sauvegarde du `pc` qui est dans ce bloc d'activation, cette action n'est pas suffisante pour que la méthode sur le haut de la pile des appels reprenne son exécution dans le bon tableau de code. Nous devons forcer le

rafraîchissement du `pc` courant, lors de la réanimation du *thread*, avec la valeur que nous avons modifié dans le bloc d'activation. De cette façon, lors de la reprise de l'exécution du *thread*, l'invocation en question utilisera le bon tableau de code.

Nous avons donc systématiquement ajouté, dans l'implantation des instructions Java de la machine virtuelle, un rafraîchissement obligatoire du `pc` après tous les endroits susceptibles de provoquer l'arrêt du *thread* courant. Si un *thread* fait appel à *stop_the_world*, les autres ne s'arrêtent pas immédiatement. Ils le feront lorsqu'ils croiseront un point de vérification et nous n'effectuons pas cette vérification à toutes les instructions Java en cours d'exécution. Cette manière de faire serait trop coûteuse. Cette vérification se fait seulement lorsque le flot d'exécution retourne en arrière (sauts arrière dans les boucles) ou lorsqu'on entre ou quitte une méthode [Gag02]. Grâce à ceci, il devient facile de repérer dans l'implantation des instructions Java les endroits où il est nécessaire de rafraîchir la valeur du `pc`.

3.5.3 Le problème des variables locales d'adresses

Les variables locales d'une méthode peuvent contenir des valeurs de type de base (*int*, *boolean*, *float*, *double*, ...) ou de type référence. SableVM effectue, lors de la préparation d'une méthode, la séparation des variables locales de manière à regrouper séparément (dans le tableau des variables locales) les variables qui sont de type référence et de type non-référence. Ceci est utile au nettoyeur de mémoire pour lui permettre de déterminer les objets qui sont encore actifs en mémoire. Il doit pouvoir repérer facilement et sans ambiguïté les références vers les objets. Cette séparation en deux catégories de variables locales (référence et non-référence) n'est plus suffisante pour notre système. Nous avons conséquemment ajouté une troisième catégorie de variable locale qui est essentielle à la mise à jour des activations d'une méthode. Nous appelons cette catégorie les variables d'adresses. Il est possible qu'une méthode possède, dans un cas exceptionnel que nous expliquons ici, dans ses variables locales, une adresse qui servira de destination de retour à un appel de sous-routine. Nous devons détecter la présence de telles valeurs dans les variables locales et les mettre à jour pour prévenir un saut dans un tableau de code périmé.

3.5.4 Les sous-routines dans le code octet

La sous-routine dans le code octet est utilisée pour implémenter le bloc *finally*{}, qui peut optionnellement suivre un *try*{} *catch*{}. Le fonctionnement des sous-routines est bien documenté [LY99, Gag02]. Une sous-routine est implémentée par les codes d'opération `JSR` et `RET`. La figure 3.7 illustre l'utilisation d'une sous-routine pour implanter le bloc *finally* {} d'une méthode. La figure utilise du code octet non transformé par SableVM, c'est-à-dire celui que nous retrouvons dans un fichier *.class*. Voici ce qui se produit lors de l'exécution d'un `JSR`. Premièrement, le `JSR` met sur la pile des opérandes l'adresse de l'instruction qui le suit. Ensuite, son paramètre lui indique l'adresse de la sous-routine où il faut continuer le flot d'exécution. Habituellement, la sous-routine commence par un `ASTORE` qui sauvegarde l'adresse sur le haut de la pile (l'adresse qui suit le `JSR`) dans une variable locale. Ensuite, la sous-routine exécute son code et se termine par un `RET`. L'instruction `RET` possède en paramètre l'index dans le tableau des variables locales, où se trouve l'adresse de retour de la sous-routine. Le `RET` saute à cette adresse et le flot d'exécution continue de cet endroit.

3.5.5 Séparation des variables locales

En réutilisant l'algorithme d'analyse de flot [Gag02] déjà en place dans SableVM, on réussis à savoir quels types de variables sont stockées pour chaque emplacement dans le tableau des variables locales. Nous avons apporté quelques modifications à l'algorithme pour qu'il supporte l'identification d'un troisième type de variable : les variables d'adresses. Avec ces informations, nous pouvons séparer et regrouper les variables qui sont utilisées pour emmagasiner les adresses de retour des sous-routines. Nous plaçons ces variables à la fin du tableau des variables locales. Nous conservons, dans la structure de la méthode (*method_info*), l'index du début de ces variables. Nous modifions aussi, dans le code octet, les instructions qui référençaient ces variables pour qu'ils utilisent le nouvel index. Si une variable était utilisée pour stocker plusieurs types de valeurs, cette variable est dédoublée pour que chaque espace ne stocke plus qu'un seul type de valeur. De cette façon, nous retirons l'ambiguïté qui existerait sur le type du contenu de cette variable locale. La figure 3.7 illustre le tableau des variables locales avant et après la séparation.

Puisque ce travail est effectué lors de la préparation de la méthode, il devient facile

Code Java:

```
public static int m1(int a) {
 int b;
 int result = -1;
 try {
  result = m2();
 }
 catch(Exception e) {
  result = 0;
 }
 finally {
  result = Math.abs(result);
 }
 B = 12;
 return result;
}
```

Code octet (avant séparation):

Locals:

index	0	1	2	3	4
type	int	int	ref	addr int	ref

```
   0:   iconst_m1
   1:   istore_1
   2:   invokestatic
   5:   istore_1
   6:   goto    31
--- /* catch */
   9:   astore  4
   11:  iconst_0
   12:  istore_1
   13:  goto    31
--- /* handler */
   16:  astore_2
   17:  jsr     22
   20:  aload_2
   21:  athrow
--- /* sous-routine */
   22:  astore  3
   24:  iload_1
   25:  invokestatic
   28:  istore_1
   29:  ret 3
---
   31:  jsr     22
   34:  bipush  12
   36:  istore_3
   37:  iload_1
   38:  iload_3
   39:  iadd
   40:  ireturn
```

Code octet (après séparation):

Locals:

index	0	1	2	3	4	5
type	int	int	int	ref	ref	addr

```
   0:   iconst_m1
   1:   istore_1
   2:   invokestatic
   5:   istore_1
   6:   goto    31
--- /* catch */
   9:   astore  4
   11:  iconst_0
   12:  istore_1
   13:  goto    31
--- /* handler */
   16:  astore_3
   17:  jsr     22
   20:  aload_3
   21:  athrow
--- /* sous-routine */
   22:  astore  5
   24:  iload_1
   25:  invokestatic
   28:  istore_1
   29:  ret 5
---
   31:  jsr     22
   34:  bipush  12
   36:  istore_2
   37:  iload_1
   38:  iload_2
   39:  iadd
   40:  ireturn
```

Figure 3.7 – Sous-routines avec (`jsr/ret`).

de mettre à jour les variables d'adresses lorsque nous actualisons les invocations d'une méthode. Nous effectuons cette mise à jour, si nécessaire, lors du parcours de la pile des blocs d'activation de chaque *thread* au moment où nous actualisons la sauvegarde du `pc` dans le bloc. Au même moment, nous actualisons les valeurs des variables locales d'adresses, si elles existent.

Résumons le processus de mise à jour des invocations. Nous parcourons tous les blocs d'activation dans la pile associée à chaque *thread*. Si nous croisons un bloc qui est associé à la méthode ciblée, nous ajustons le `pc` et le contenu des variables d'adresses (variables locales) dans le bloc d'activation. Lorsque les *threads* endormis se réveilleront, ils vont rafraîchir leur `pc` avec la valeur qui a été sauvegardée dans leur bloc d'activation.

3.6 Synchronisation

3.6.1 Modification des attributs d'une méthode

Puisque SableVM est un environnement potentiellement habité par plusieurs *threads*, nous devons prendre des précautions supplémentaires afin d'assurer l'intégrité du système de profilage. Le sujet a déjà été abordé en partie à la section 3.2.4. Nous avions introduit le fait que nous utilisons les procédures *stop_the_world* et *start_the_world* pour s'assurer qu'aucun *thread* n'exécute du code Java pendant qu'on modifie une méthode en particulier. Sachant ceci, voici comment nous procédons pour nous assurer un accès exclusif aux structures à modifier dans la méthode. Nous ne voulons pas que deux *threads* différents puissent, en même temps, modifier les structures associées à une méthode donnée.

Si un *thread* veut démarrer ou stopper le profilage d'une méthode, il doit d'abord stopper tous les *threads* successibles d'exécuter du code Java avec la procédure *stop_the_world*. Une fois le monde immobilisé, il peut modifier la méthode et ses attributs à volonté. Les modifications étant complétées, celui-ci doit s'assurer que tous les *threads*, qui utilisaient la méthode ciblée, utilisent maintenant la nouvelle version de la méthode. Finalement, il peut redémarrer tous les *threads* par un appel à la procédure *start_the_world*.

Cette manière de faire simplifie beaucoup la modification du tableau de code et de la table d'exceptions de la méthode. Nous n'avons pas besoin de nous soucier des autres

threads qui pourraient venir utiliser ou modifier le tableau de code pendant qu'il se trouve dans un état incohérent.

Nous aurions pu essayer de trouver une façon de faire le même travail sans toutefois arrêter le monde. Toutefois, puisque nous avions plusieurs variables à modifier dans la structure de la méthode, il aurait été difficile d'effectuer leur mise à jour de manière atomique. Nous avons donc favorisé un arrêt complet du monde pendant que nous transformons une méthode.

3.6.2 Incrémentation des compteurs

L'incrémentation des compteurs de profilage pendant l'exécution s'effectue sans synchronisation. Il est possible que deux *threads*, qui exécutent une même instruction de profilage au sein de la même méthode en même temps, puissent causer une perte d'information. Par exemple, si les événements suivants se produisent :

1. le *thread* 1 lit la valeur du compteur (valeur=12)
2. le *thread* 2 lit la valeur du compteur (valeur=12)
3. le *thread* 1 incrémente la valeur lue (condition du branchement vraie)
4. le *thread* 2 décrémente la valeur lue (condition du branchement fausse)
5. le *thread* 1 écrit le résultat dans le tableau de code (valeur=13)
6. le *thread* 2 écrit le résultat dans le tableau de code (valeur=11)

Le résultat qui sera inscrit dans le tableau de code sera 11, alors qu'il devrait plutôt être 12. Cette situation est possible, mais peu fréquente.

Nous n'avons pas besoin d'un profilage exact à 100%, car tout ce que nous voulons est d'avoir une idée du chemin qui est le plus fréquemment emprunté. Nous voulons obtenir cette information sans toutefois détériorer les performances de l'interpréteur. Pour prévenir la situation qui est présentée dans l'exemple précédent, il faudrait protéger la modification des compteurs par des verrous. Cette protection supplémentaire aurait des conséquences importantes sur la performance. Dans ce cas, l'utilisation d'un verrou pourrait représenter beaucoup plus d'instructions-machine que l'instruction Java que nous voulons profiler. Alors, nous avons choisi de permettre un accès non synchronisé aux compteurs de profilage et nous considérons négligeables les effets de ce choix sur les résultats que nous visons à obtenir.

3.7 Précision et limites du système

Nous avons dû faire des compromis au niveau de la précision et de la qualité du profilage afin de maintenir des performances acceptables au sein de l'interpréteur. Dans le contexte d'une *vm* outillée d'un compilateur dynamique optimisant, où les gains causés par les optimisations sont beaucoup plus substantiels, il est possible d'effectuer un profilage plus agressif. Mais dans notre cas, les gains que nous souhaitons obtenir ne sont reliés qu'à la diminution des répartitions entre les instructions exécutées. Les gains sont donc plus modestes et moins susceptibles de compenser un profilage plus complexe. Regardons quelques-unes des limites du système de profilage présenté dans ce chapitre.

Premièrement, nous avons déjà discuté dans la section 3.6 de l'imprécision des compteurs de profilage situés à même les instructions de branchement dans le code profilant. Une possible imprécision pourrait résulter d'un accès non exclusif aux compteurs de profilage.

Deuxièmement, un manque de contexte relié au profilage vient réduire la précision de celui-ci. Pour une méthode donnée, nous stockons les informations de profilage de manière globale (dans le code de celle-ci). Nous n'utilisons pas des informations de profilage relatives à chaque site d'invocation. Il est possible qu'il existe une corrélation entre le chemin emprunté dans une méthode et l'identité de l'appelant de cette méthode. Il est aussi possible qu'il existe une corrélation entre le chemin emprunté et l'emplacement du site d'invocation. Nous ne tenons pas compte non plus des paramètres qui sont passés à la méthode. Il est fort probable qu'il existe une relation entre la valeur des paramètres passés à une méthode et le chemin d'exécution à l'intérieur de celle-ci. En conclusion, pour le profilage d'une méthode donnée, nous ne tenons pas compte de qui fait l'appel, ni du lieu d'où se fait l'appel, ni de la valeur des paramètres passés à la méthode.

Tenir compte d'un plus grand contexte dans notre profilage aurait augmenté de beaucoup la complexité et le coût du système et nous ne pouvons nous permettre un tel luxe. Le profilage doit rester simple et léger, car les gains que nous ferons par l'inclusion d'une méthode devront compenser les coûts additionnels du profilage et du mécanisme d'inclusion.

L'argument qui nous permet de nous satisfaire d'un profilage moins exact est le suivant : si jamais nous faisons l'inclusion d'un mauvais chemin, le coût pour sortir du code inclus

et continuer l'exécution au bon endroit dans le code original est presque nul. Même si le flot lors de l'exécution ne correspond pas toujours au code inclus (chemin sélectionné), nous pourrons le détecter et continuer l'exécution dans le code original, à prix modique.

3.8 Conclusion

Dans ce chapitre, nous avons expliqué plus en détails comment est articulé le mécanisme de profilage. Nous avons vu pourquoi le profilage est nécessaire et comment nous sélectionnons les méthodes à inclure. Nous avons exposé les nouvelles instructions que nous avons ajoutées au jeu d'instructions de SableVM, pour permettre d'initier et d'arrêter le profilage d'une méthode ainsi que pour effectuer le profilage en tant que tel. Nous avons présenté les solutions mises en place pour régler les problèmes reliés aux exceptions, aux sous-routines et à la synchronisation. Nous avons également présenté les limites du système et nous avons justifié nos choix quant à la précision et à la qualité du profilage mis en place.

Chapitre 4

Mécanisme d'inclusion

Dans ce chapitre, nous allons détailler et approfondir plus spécifiquement le mécanisme d'inclusion. Nous commençons par expliquer comment se prend la décision de faire ou non l'inclusion et comment est démarrée l'inclusion d'une méthode. Ensuite, nous expliquons comment se déroule l'inclusion d'une méthode (sections 4.2 à 4.7). Ces sections représentent, en ordre, les différentes étapes du processus. Finalement, nous discutons des contraintes d'implémentation et des limites du système.

4.1 Démarrage du mécanisme d'inclusion

Comme nous l'avons expliqué à la section 3.2.3, les instructions `COUNTING_INVOKE<X>` se chargent de compter le nombre de fois qu'un site d'invocation a été visité. Par la suite, l'instruction utilise ce même compteur pour déterminer si l'inclusion du site est autorisée. Pour que l'inclusion soit autorisée à cet endroit, la valeur du compteur doit dépasser le seuil d'autorisation à l'inclusion qui est un paramètre global du système (*method_invoke_start_threshold*). Si la condition est respectée, tous les *threads* sont arrêtés par l'appel à la routine *stop_the_world()* et l'inclusion du site est effectuée.

Si l'opération se termine avec succès, les *threads* au sein de la *vm* sont autorisés à redémarrer par l'appel à la routine *start_the_world()*. La méthode appelante possède alors un nouveau tableau de code dans lequel la méthode appelée a été incluse. Si au contraire l'opération échoue, le tableau de code de la méthode appelante reste inchangé, sauf que le

`COUNTING_INVOKE<X>` est remplacé atomiquement par un `INVOKE<X>` normal. Cette modification a pour effet d'empêcher d'incrémenter le compteur du site lors des prochains passages et d'empêcher de tenter l'inclusion de nouveau alors que nous savons que l'inclusion y est impossible. Nous aurions affaire à cette situation dans le cas d'un appel récursif. Nous ne supportons pas pour le moment l'inclusion des appels de méthodes récursives afin de prévenir des séquences de longueurs non-bornées. Nous conservons l'ajout de cette fonctionnalité pour des travaux futurs.

4.2 Construction du code à inclure

Une fois l'inclusion d'une méthode autorisée, la première étape du processus est de sélectionner le chemin à inclure en le stockant dans un tableau de code que nous allouons dynamiquement au besoin. Nous illustrons ici, la construction d'un tel tableau avec un exemple simple, avant de présenter l'algorithme plus général.

4.2.1 Exemple

La figure 4.1 illustre, en se basant sur l'exemple de la figure 2.2, le tableau de code une fois qu'il a été rempli. Nous utilisons la version profilante du code de la méthode appelée pour construire le code à inclure. L'algorithme pour remplir le tableau du code est assez simple. La première chose à faire est d'y copier la première instruction, c'est-à-dire le bon type de `INLINED_INVOKE`. Dans notre exemple, nous effectuons l'inclusion d'une méthode statique alors nous copions un `INLINED_INVOKESTATIC` ainsi que ses paramètres. Ensuite, nous nous positionnons au début du tableau de code profilant et nous copions dans le tableau de code résultat toutes les instructions et leurs paramètres jusqu'à ce que nous rencontrions une instruction profilante associée à un branchement. Dans notre exemple, cela correspond aux instructions comprises entre les adresses 130 et 139. À ce moment, nous vérifions la valeur du compteur de profilage associée à l'instruction de branchement pour déterminer ce qui se produit le plus souvent à cet endroit (compteur adresse 139). Ici, le compteur est positif, ce qui indique que le branchement est emprunté plus souvent qu'autrement. Nous ne copions pas l'instruction de branchement, mais plutôt

une instruction qui sert de point de vérification (`INLINED_IF_ICMPLT`) et qui vérifiera, lors de l'exécution, si l'hypothèse que nous avons faite à cet endroit est vraie. Cette instruction vérifie donc la condition contraire à celle du branchement original. Nous construisons ici le chemin en supposant que la condition est vraie et le point de vérification sert à vérifier si la condition est fausse lors de l'exécution. Une fois que le point de vérification a été inséré dans le tableau de code résultat, nous sautons à l'endroit où sautait le branchement dans le code profilant (adresse 147). Nous continuons de copier les instructions de cet endroit, car nous avons supposé que la condition du branchement (adresse 137) est toujours vraie. Si le compteur du branchement profilant avait été négatif, nous aurions continué la copie des instructions à l'adresse 140, c'est-à-dire de l'instruction qui suit le branchement, puisque le compteur nous aurait indiqué que le choix le plus fréquent à cet endroit est un non-branchement. Ensuite, nous copions les instructions comprises entre 147 et 152 dans le tableau de code résultat. Lorsque nous rencontrons le `IRETURN` (adresse 152) nous le substituons par un `INLINED_IRETURN` et ceci marque la fin du chemin à inclure. Tout chemin à inclure se termine par un *stack_gc_map* que nous devons ajouter pour le nettoyeur de mémoire (adresse 195).

4.2.2 Le *stack_gc_map* de terminaison

Nous avons déjà expliqué que chaque bloc d'activation dans la pile des appels de méthode contient la valeur du `pc` de retour qui sert à indiquer où doit reprendre le flot d'exécution une fois la méthode appelée terminée (section 1.2). Quand l'exécution du nettoyeur de mémoire est déclenchée, celui-ci doit inspecter chaque bloc d'activation de la pile des appels de méthodes pour y trouver les références et ainsi pouvoir identifier les objets qui sont toujours vivants en mémoire. Ces références se trouvent dans le tableau des variables locales du bloc d'activation ou dans la pile des opérandes. Pour trouver les éléments qui sont des références dans la pile des opérandes, le nettoyeur de mémoire utilise le *stack_gc_map* (section 1.3.5), qui doit toujours, dans SableVM, être stocké dans le tableau de code à l'emplacement identifié par le `pc` de retour moins 1. Dans l'exemple de la figure 2.2, lors de l'invocation de la méthode appelée dans le cas normal à l'adresse 05, le `pc` de retour de la méthode appelante est modifié pour qu'il pointe sur l'instruction

Méthode appelée (version profilante)	**Tableau du code à inclure**	**Table de conversion**
130: [ILOAD_0] 131: [ISTORE_1] 132: [LDC] 133: (11) 134: [ISTORE_0] 135: [ILOAD_0] 136: [ICONST_4] 137: [IF_ICMPGE] 138: (@147) 139: (5) <- cpt profilage 140: [ILOAD_1] 141: [ILOAD_0] 142: [IADD] 143: [ISTORE_1] 144: [GOTO] 145: (@151) 147: [ILOAD_1] 148: [ILOAD_0] 149: [IADD] 150: [ISTORE_1] 151: [ILOAD_1] 152: [IRETURN]	175: [INLINED_INVOKESTATIC] 176: (16) <- taille code inclus 177: (args_count) 178: (@method_info) 179: (@stack_gc_map) 180: [ILOAD_0] 181: [ISTORE_1] 182: [LDC] 183: (11) 184: [ISTORE_0] 185: [ILOAD_0] 186: [ICONST_4] 187: [INLINED_IF_ICMPLT] 188: (@319) <- @ dans table conv. 189: [ILOAD_1] 190: [ILOAD_0] 191: [IADD] 192: [ISTORE_1] 193: [ILOAD_1] 194: [INLINED_IRETURN] 195: (@stack_gc_map)	310: @30 311: @31 312: @32 313: @33 314: @34 315: @35 316: @36 317: @37 318: @38 319: @39 320: @40 321: @41 322: @42 323: @43 324: @44 325: @45 326: @46 327: @47 328: @48 329: @49 330: @50

Figure 4.1 – Tableau contenant le code à inclure (interpréteur *switch*).

de l'adresse 10. Lors du retour de la méthode appelée, c'est à cet endroit que reprendra l'exécution de la méthode appelante. On note qu'à l'adresse 09 (`pc`-1) se trouve un pointeur vers un *stack_gc_map*. Pour que le nettoyeur de mémoire puisse faire son travail, s'il est déclenché pendant l'exécution de code inclus, il faut que le tableau de code contienne l'adresse d'un *stack_gc_map* à l'emplacement qui précède le `pc` de retour de la méthode appelante. Dans notre exemple (figure 2.2), le `pc` de retour pointerait vers l'adresse 96 et l'adresse du *stack_gc_map* doit être placée à l'adresse 95. Nous ne nous étendrons pas plus sur le sujet, mais il faut savoir que tout code inclus doit se terminer par l'adresse d'un *stack_gc_map* afin d'assurer l'intégrité de la *vm* si le nettoyeur de mémoire est déclenché pendant l'exécution de code inclus.

4.2.3 Le problème des sauts arrière

Un problème se pose lors de la sélection du chemin à inclure si la méthode appelée possède des sauts arrière au sein de son code. Les sauts arrière sont surtout utilisés pour

implanter les boucles. Comme nous voulons nous assurer que le chemin que nous construisons possède une longueur finie, nous devons traiter de manière spéciale les sauts arrière. Plutôt que d'avoir à effectuer une analyse de flot du code pour déterminer si un saut arrière nous emmène sur un chemin de longueur infinie, nous avons décidé de simplifier les choses et de poser que la rencontre d'un saut arrière détermine la fin du chemin en question. Lorsque nous rencontrons un saut arrière, nous ajoutons une instruction de type `INLINED_GOTO` pour terminer le chemin et faire sauter le flot d'exécution dans le code original de la méthode appelée.

4.2.4 Algorithme de construction du code à inclure

L'algorithme est assez simple, il suffit de parcourir le tableau de code profilant et de vérifier pour chaque instruction lequel des huit cas suivants doit s'appliquer.

1. Si l'instruction courante est un `GOTO`, alors :
   - Si ce `GOTO` effectue un saut vers l'avant, nous sautons à cette destination et nous continuons à copier les instructions de cet endroit. Nous ne copions pas le `GOTO` dans le code à inclure.
   - Si le `GOTO` effectue un saut vers l'arrière, nous insérons un `INLINED_GOTO` dans le code à inclure. Cette instruction a pour objectif de faire sauter le flot d'exécution au bon endroit dans le code original de la méthode appelée. Ceci marque la fin du chemin sélectionné, car nous ne supportons pas les sauts arrière dans le code à inclure.
2. Si l'instruction courante est une instruction de branchement profilante de type `PROFILING_IF<X>`, alors :
   - Si le compteur de profilage est positif et que le branchement est vers l'avant, nous ne copions pas l'instruction de branchement, mais une instruction point de vérification qui vérifie la condition contraire (voir figure 4.1 adresse 187). Nous continuons ensuite à copier les instructions à partir de la destination du branchement. Le point de vérification vérifira si le chemin est invalide et si c'est le cas il nous fera sauter dans le code original de la méthode appelée.
   - Si le compteur de profilage est positif et que le branchement est vers l'arrière, nous

devrions normalement sauter à la destination arrière et poursuivre la construction du chemin de cet endroit, mais nous ne supportons pas les sauts arrière. Dans ce cas, nous pourrions terminer le chemin en ajoutant un `INLINED_GOTO` qui nous ferait retomber dans le code original. Plutôt que de mettre fin au chemin de cette façon et faire sauter le flot d'exécution dans le code original dans tous les cas, nous allons inclure le chemin comme si c'était le non branchement qui était le plus fréquent. De cette façon, dans les cas où le branchement n'est pas pris, nous pourrons sauver des répartitions et dans le cas contraire, nous brancherons dans le code original de la méthode appelée. Donc, nous ne copions pas l'instruction de branchement, mais une instruction point de vérification qui vérifie la même condition que celle du branchement courant. Nous continuons ensuite à copier les instructions à partir de l'instruction qui suit le branchement dans le code profilant. Nous supposons ici qu'il n'y a pas de branchement. Le point de vérification vérifira, au moment de l'exécution, si nous devrions brancher et si c'est le cas nous brancherons dans le code original de la méthode appelée.

- Si le compteur de profilage est négatif, nous ne copions pas l'instruction de branchement, mais une instruction point de vérification qui vérifie la même condition que celle associée au branchement courrant. Nous continuons ensuite à copier les instructions à la prochaine instruction. Nous supposons ici qu'il n'y a pas de branchement. Le point de vérification vérifira, au moment de l'exécution, si nous devrions brancher et si c'est le cas, nous brancherons dans le code original de la méthode appelée.

3. Si l'instruction courante est une instruction de branchement de type `PROFILING_TABLESWITCH`, alors nous déterminons grâce aux compteurs de profilage la destination vers l'avant la plus fréquente.
   - Si la destination vers l'avant la plus fréquente est le *default case*, nous copions un point de vérification du type `INLINED_TABLESWITCH_DEFAULT` qui sert à vérifier, lors de l'exécution, que nous sommes vraiment dans le cas par défaut. Nous sautons ensuite à la destination associée au *default case* et nous continuons à copier les instructions de cet endroit.

- Si la destination vers l'avant la plus fréquente est un *case* normal, nous copions un point de vérification du type `INLINED_TABLESWITCH_CASE` qui servira à vérifier, lors de l'exécution, que nous avons vraiment choisi le bon *case* lors de l'inclusion. Ensuite, nous sautons à la destination associée au *case* que nous avons choisi et nous continuons à copier les instructions de cet endroit.

4. Si l'instruction courante est une instruction de branchement de type `PROFILING_LOOKUPSWITCH` alors, nous agissons exactement comme dans le cas d'un `PROFILING_TABLESWITCH` expliqué plus haut, sauf qu'au lieu d'utiliser un `INLINED_TABLESWITCH_DEFAULT` nous utilisons un `INLINED_LOOKUPSWITCH_DEFAULT` et au lieu d'utiliser un `INLINED_TABLESWITCH_CASE` nous utilisons un `INLINED_LOOKUPSWITCH_CASE`.
5. Si l'instruction courante est un `<X>RETURN` alors nous ne copions pas cette instruction dans le code à inclure, mais nous la remplaçons par le `INLINED_<X>RETURN` correspondant et ceci marque la fin du code à inclure.
6. Si l'instruction courante est l'instruction `JSR` (saut vers une sous-routine) alors nous ne copions pas l'instruction `JSR` dans le code à inclure, mais plutôt l'instruction `INLINED_JSR`. Le `INLINED_JSR` met sur la pile non pas l'adresse de l'instruction qui le suit comme un `JSR` normal, mais plutôt l'adresse de retour de la sous routine dans le code original de la méthode appelée. Nous sautons ensuite à la sous-routine et continuons le copiage des instructions de cet endroit.
7. Si l'instruction courante est l'instruction `RET` (retour de sous-routine) alors nous ne copions pas l'instruction `RET` dans le code à inclure, mais plutôt l'instruction `INLINED_RET` qui se comporte exactement comme le `RET` sauf que la destination du saut se trouve dans le code original de la méthode appelée.
8. Si l'instruction courante ne correspond à aucun des cas énumérés ci-haut, on copie l'instruction et ses opérandes intégralement dans le tableau du code à inclure et nous passons à l'instruction suivante.

L'annexe B contient la liste exhaustive de tous les nouveaux codes d'opération que nous avons ajoutés au jeu d'instructions de SableVM pour permettre l'inclusion d'une

méthode. Tous les codes d'opération préfixés de `INLINED_` se retrouvent uniquement au sein de code inclus.

4.2.5 Table de conversion

Dans l'exemple de la figure 2.2 nous avions mis, à des fins de simplification, comme paramètre du point de vérification l'adresse dans le code original où on doit sauter si le chemin inclus est invalide (voir ligne 88). En réalité, le paramètre d'un point de vérification est l'adresse d'un élément dans une table de conversion associée à la méthode appelée. Cet élément de la table donne l'adresse de la destination où nous devons sauter pour continuer l'exécution dans le code original. La figure 4.1 donne la table de conversion pour notre exemple. La colonne de gauche représente les adresses des éléments du tableau et la colonne de droite donne les adresses qui sont stockées dans la table. On peut constater, à l'adresse 188 dans le tableau du code à inclure, que le paramètre est l'adresse dans la table de conversion (319) qui indique où sauter dans le code original (dans ce cas l'adresse de destination serait 39). Cette table agit comme une indirection supplémentaire pour permettre de retourner le flot d'exécution dans le code original.

Nous ne pouvons pas utiliser directement les adresses absolues dans le code original pour les points de vérification, car il est possible que le code original de la méthode appelée change avec le temps. Par exemple, supposons qu'une méthode appelante A appelle une méthode B et que nous faisions l'inclusion de B dans A. Si nous utilisons des adresses absolues référant vers le code de B dans les points de vérifications utilisés lors de l'inclusion de B dans A et que plus tard nous effectuions l'inclusion d'une méthode C dans B, alors les points de vérification référeront vers du code obsolète ou une zone mémoire invalide, car B possèdera un nouveau tableau de code.

Pour palier à ce problème, chaque méthode possède une table de conversion. Il s'agit d'un tableau unidimensionnel d'adresses. Sa taille est fixe et toujours égale à la taille originale du tableau de code de la méthode en question. Ce tableau nous permet en tout temps de déterminer l'adresse mémoire d'une position dans le code original de la méthode. Par exemple, si le code original de la méthode possède une instruction quelconque à l'index 10 de son tableau de code, pour obtenir l'adresse mémoire de cette instruction dans le

tableau courant, il suffit de consulter la table à l'adresse (`@table + 10`). Les instructions utilisées comme points de vérification ont toutes comme opérande une adresse dans la table de conversion plutôt qu'une adresse absolue qui leur sert à référencer une destination dans le code non inclus de la méthode appelée. Les instructions `INLINED_JSR` et `INLINED_GOTO` utilisent aussi ce type d'indirection supplémentaire pour référencer le code non inclus de la méthode appelée.

Notre conception impose en tout temps l'unicité du code pour une méthode donnée. Il n'existe en tout temps qu'un seul tableau de code valide pour une méthode. Puisque nous ne voulons pas conserver plusieurs versions d'une même méthode, la table de conversion est nécessaire pour que le code inclus antérieurement à divers endroits puisse en tout temps sauter dans le code actualisé de sa méthode mère.

4.3 Table des exceptions du code à inclure

Une fois que nous avons construit le tableau de code à inclure (le chemin sélectionné), nous devons gérer les exceptions pour le code sélectionné. Si le code profilant de la méthode appelée possède une table d'exceptions, nous construisons une table d'exceptions qui est associée au tableau de code à inclure. En fonction du chemin que nous avons sélectionné, cette table sera un sous-ensemble de la table du code profilant. Il s'agit d'une table intermédiaire utilisée pour ajuster la table d'exceptions de la méthode appelante. Cet ajustement est nécessaire pour gérer les exceptions qui pourraient se produire dans le code inclus. Nous allons ajouter à la table d'exceptions de la méthode appelante (section 4.5) des entrées pour gérer les exceptions qui pourraient se produire dans le code inclus.

Quand une exception se produit lors de l'exécution de code inclus, nous sautons vers le *handler* original dans le code non inclus de la méthode appelée. Puisque les exceptions sont supposées être des événements exceptionnels, lorsqu'ils se produisent nous retournons le flot d'exécution dans le code original de la méthode appelée. Si jamais une exception se produit pendant l'exécution du code inclus et qu'il n'existe pas de *handler* dans le corps de la méthode originale et que celle-ci est synchronisée, nous libérons le verrou associé à cette méthode pour rechercher un *handler* dans la table d'exception de la méthode appelante (englobante). Il est possible que nous devions parcourir toute la pile des appels

de méthodes avant de résussir à trouver un *handler* pour l'exception en question. Si aucun *handler* n'est trouvé pour cette exception, le programme termine son exécution et signale une erreur.

```
/* Adresses originales des instructions du code à inclure */
adresses;
chemin;    /* Tableau du code à inclure */
tableProf; /* Table d'exceptions du code profilant */
tableTmp;  /* Table intermédaire à construire */

i = 0; j = 0;

/* pour chaque entrée de la table d'execeptions */
tant que (i < tailleTableCodeProfilant)
{
  p = 0;

  /* parcours du chemin sélectionné */
  tant que (p < tailleChemin)
  {
    /* si l'entrée courante de tableProf concerne
       une instruction du chemin sélectionné */
    si (chemin[p] est une instruction) et
       (adresses[p] >= tableProf[i].pc_start) et
       (adresses[p] <= tableProf[i].pc_end)
    {
      tableTmp[j].pc_start = @chemin + p;
      tableTmp[j].type = tableProf[i].type;
      tableTmp[j].handler =
      profiling2convTable(tableProf[i].handler);

   /* avancer dans le chemin tant que l'instruction
      courante doit être gérée par l'entrée tableProf[i] */
      faire
      {
        pc_end = @chemin + p++;
      }
      tant que (adresses[p] >= tableProf[i].pc_start) et
               (adresses[p] <= tableProf[i].pc_end) et
               (p < tailleChemin);

      tableTmp[j++].pc_end = pc_end;

      /* sauter à la prochaine entrée de la table */
      break;
    }
  }
  i++;
}
```

Figure 4.2 – Algorithme de construction de la table d'exceptions du code à inclure.

Un fait important, que nous avons choisi de ne pas aborder dans la section 4.2, est qu'à

mesure que nous construisons le tableau du chemin sélectionné, nous remplissons aussi en parallèle un tableau d'adresses qui nous permet de conserver, pour chaque élément du chemin sélectionné, son adresse originale dans le tableau du code profilant. C'est ce tableau d'adresses, en combinaison avec la table des exceptions du code profilant, qui nous permettent de construire la table d'exceptions intermédiaire. La figure 4.2 montre une version simplifiée en pseudo-code de l'algorithme utilisé pour construire la table. Puisque nous ne supportons pas les sauts arrière, l'algorithme fait l'hypothèse que les instructions dans le tableau du chemin sélectionné sont dans le même ordre que ceux du code profilant. La fonction *profiling2convTable()* dans l'algorithme sert à convertir l'adresse du *handler* dans le code profilant en l'adresse de l'emplacement dans la table de conversion où se trouve l'adresse de ce même *handler* dans le non inclus et non profilant de la méthode appelée.

Mémoire du travail accompli

Il est important de noter que le tableau du chemin sélectionné et la table d'exceptions qui lui est associée seront toujours les mêmes pour une méthode donnée et ce peu importe le site d'invocation où nous en ferons l'inclusion. Une fois le profilage d'une méthode terminée, la construction du chemin et de sa table d'exceptions donneront toujours le même résultat. Une fois le travail accompli pour un site d'invocation, nous conservons ces deux structures au-delà de l'inclusion courante. Lors d'une prochaine inclusion de la méthode, l'algorithme réutilisera les deux structures, sans avoir à refaire le travail déjà accompli.

4.4 Reconstruction du code de la méthode appelante

4.4.1 Nouveau tableau de code

Une fois que le code du chemin à inclure et sa table d'exceptions ont été construits, nous sommes prêts à reconstruire le code de la méthode appelante. Nous commençons par allouer un nouveau tableau de code. Ensuite, nous copions intégralement tout le code qui précède le site d'invocation, le code du tableau contenant le chemin sélectionné et le code

qui suit le site d'invocation. Nous copions ces trois parties de code les unes à la suite de l'autre dans le nouveau tableau de code (voir le résultat figure 2.2).

4.4.2 Fusionnement des séquences (interpréteur linéaire inclusif)

Toutes les étapes du mécanisme d'inclusion sont identiques pour les trois types d'interpréteurs de SableVM, sauf pour l'interpréteur linéaire inclusif où nous avons ajouté une étape entre la reconstruction du code de la méthode appelante et l'ajustement des adresses à l'intérieur de celle-ci. Cette étape supplémentaire consiste à fusionner les séquences qui peuvent l'être. Puisque la construction de la nouvelle version du tableau de code consiste en la concaténation de trois parties de code, les séquences qui se touchent aux frontières de ces trois parties peuvent potentiellement être fusionnées, sans compter qu'il est aussi possible que nous puissions fusionner des séquences qui se trouvent dans le chemin sélectionné. C'est cette fusion qui occasionnera une réduction des répartitions lors de l'exécution de la zone de code optimisée. Dans l'exemple de la figure 2.3, nous voyons que les séquences qui précèdent et qui suivent le site d'invocation ont été fusionnées avec les séquences qui composent le code inclus. L'étape qui consiste en la fusion des séquences s'assure aussi de dupliquer la séquence qui suit le site d'invocation si celle-ci a été incluse dans la séquence du code inclus (section 2.2.5).

4.4.3 Ajustement des adresses

Comme dans le cas de la construction du code profilant (section 3.3.2), nous devons aussi ajuster les adresses qui sont les opérandes des instructions de notre nouveau tableau de code. Ceci est nécessaire pour prévenir que le flot d'exécution puisse sauter dans l'ancien tableau de code qui aura été libéré de la mémoire. Nous utilisons une formule simple pour effectuer l'ajustement des adresses. Chaque fois qu'on rencontre une adresse qui est l'opérande d'une instruction qui n'est pas dans le code inclus, on agit comme suit : supposons que `A` est l'ancien tableau de code de la méthode appelante, que `N` est son nouveau tableau de code et que `diff` est la différence de taille entre les deux tableaux.

– Si la destination est avant le site d'invocation dans `A`, la nouvelle adresse est : `@N + (destination - @A)`

– Si la destination est après le site d'invocation dans `A`, la nouvelle adresse est : `@N + ((destination - @A) + diff)`

Lorsque l'ajustement des adresses a été effectué, nous détruisons l'ancien tableau de code et ceci marque la fin de la reconstruction de la méthode appelante.

4.5 Ajustement de la table d'exceptions

Nous devons modifier la table d'exceptions de la méthode appelante, car les entrées qu'elle contient se réfèrent à du code qui n'existe plus. Si nous avons construit une table d'exceptions pour le chemin sélectionné (section 4.3), nous devons aussi ajouter les entrées qu'elle contient dans la table pour gérer les exceptions potentielles dans le code inclus. L'ajustement de la table se fait en deux étapes.

4.5.1 Ajout des nouvelles entrées

Si nous devons ajouter des nouvelles entrées, nous créons une nouvelle table d'exceptions, sinon nous conservons la même table et sautons à l'étape de la modification des entrées déjà existantes (section 4.5.2). Puisque nous voulons que les nouvelles entrées (relatives au code inclus) que nous ajoutons à la table aient priorité sur celles déjà présentes, nous devons les ajouter au début de la table. Cette priorisation des entrées s'explique de la façon suivante. Quand une exception se produit, nous devons parcourir la table des exceptions en ordre à la recherche d'une entrée chargée de gérer l'exception (section 3.4.2). Si plusieurs entrées peuvent gérer l'exception, nous utilisons la première entrée rencontrée dans la table. Puisqu'en Java il est possible d'avoir plusieurs *try/catch* imbriqués, il peut exister plusieurs entrées dans la table qui s'appliquent à une même zone de code. Nous copions donc les entrées relatives au code inclus, qui proviennent de la table d'exception du code à inclure, au début de la nouvelle table. À chaque entrée que nous ajoutons, il est nécessaire d'ajuster les champs *pc_start* et *pc_end* pour qu'ils correspondent aux adresses dans la méthode appelante au lieu de correspondre à des adresses dans le tableau du code à inclure. Le champ *handler* reste inchangé puisqu'il se réfère à la table de conversion de la méthode appelée. Le champ *type* reste aussi inchangé.

4.5.2 Modification des entrées déjà existantes

Si nous avons créé une nouvelle table à l'étape précédente, nous copions les entrées de l'ancienne table dans la nouvelle à la suite des entrées relatives au code inclus et sinon nous ne faisons que modifier les champs déjà existants. Dans les deux cas, nous devons modifier les champs *pc_start*, *pc_end* et *handler* pour qu'ils se réfèrent à la nouvelle version de la méthode. La conversion des adresses contenues dans ces champs s'effectue exactement comme la conversion des adresses dans le nouveau tableau de code (section 4.4.3).

4.6 Mise à jour de la table de conversion

Comme nous l'avons expliqué à la section 4.2.5, chaque méthode possède une table de conversion qu'il est nécessaire de mettre à jour quand nous modifions son code. L'algorithme pour effectuer cette tâche est simple. Nous parcourons le nouveau tableau de code de la méthode modifiée et nous utilisons deux index pour nous permettre d'avancer simultanément dans les deux tableaux. L'index `i` est utilisé pour le tableau de code et `j` pour la table de conversion. Donc, pour chaque élément du tableau code :

- Si la position courante est dans une zone de code incluse, on passe au prochain élément sans rien faire (`i++`).
- Si la position courante n'est pas dans une zone de code incluse, on ajoute l'adresse de cet élément dans la table de conversion (`table_conv[j++] = @code + i++`).

4.7 Mise à jour des invocations

Après avoir modifié le code et la table d'exceptions d'une méthode, il est impératif de s'assurer que toutes les invocations de cette méthode soient mises à jour afin que tous utilisent la nouvelle version du code (car l'ancienne n'existe plus). Nous devons effectuer cette mise à jour avant de donner l'autorisation aux autres *threads* de redémarrer. Nous effectuons cette tâche de la même façon que lorsque nous activons ou désactivons le profilage d'une méthode. Nous avons déjà expliqué dans la section 3.5 comment se fait la mise à jour des `pc` dans les blocs d'activation pour tous les *threads* et comment pour chacun de

ces blocs nous mettons à jour les variables locales d'adresses utilisées pour les retours de sous-routines. Une fois que toutes les invocations de la méthode ont été mises à jour, le processus d'inclusion est terminé et le flot d'exécution du code Java pour le *thread* courant reprend son cours au début du code de la méthode incluse (sur le `INLINED_INVOKE<X>`). Les autres *threads* sont aussitôt autorisés à redémarrer et le travail est terminé.

4.8 Contraintes d'implémentation

Cette section concerne l'interpréteur linéaire inclusif de SableVM et les difficultés supplémentaires que nous avons dû affronter pour mettre en place le mécanisme d'inclusion. Comme nous l'avons expliqué dans la section 1.5.3, pour créer dynamiquement l'implémentation d'une super-instruction, nous devons copier en mémoire les implémentations des instructions élémentaires qui la composent. La super-instruction est donc constituée de la concaténation des implémentations des instructions qui la composent. Or, un problème important peut se poser lors du copiage en mémoire. Il faut se rappeler que le code que nous copions en mémoire est le code machine généré par le compilateur C (dans notre cas *gcc*).

Il n'est pas toujours possible de copier en mémoire le corps des instructions et de maintenir leur intégrité [Gag02]. Certaines instructions, lorsque copiées en mémoire, deviennent inutilisables. L'implantation d'un code d'opération n'est pas copiable en mémoire si elle contient un des éléments suivants :

- Un appel de fonction C, si le compilateur implémente les appels de fonctions par un déplacement relatif en mémoire. Une fois copié, ce déplacement relatif devient invalide.
- Toute instruction C qui provoque un appel interne caché d'une fonction C et qui serait implantée par un déplacement relatif. Par exemple, les divisions sur les *long* sur les plateformes x86 compilées avec *gcc* font un appel de fonction cachée interne.
- Toute instruction située entre l'étiquette `START` et l'étiquette `END` du code d'opération courant et qui consiste en un saut vers une adresse non absolue. Par exemple, un saut relatif à la valeur du `pc`.

Ceci implique que plusieurs codes d'opération contenus dans le jeu d'instructions de

SableVM ne sont pas copiables en mémoire et qu'ils ne peuvent pas se retrouver au sein d'une super-instruction. L'algorithme présenté à la figure 1.7 ne peut donc pas être utilisé tel quel pour déterminer les blocs de base. Il faut le modifier pour qu'il s'assure que les instructions non copiables en mémoire se retrouvent seules dans leur bloc de base.

Cette problématique est venue compliquer l'implémentation de notre mécanisme d'inclusion. Toutes les instructions préfixées de `INLINED_` (annexe B) ont dû être rendues copiables en mémoire, de manière à pouvoir être insérées dans les super-instructions. Nous avons dû tenir compte de cette contrainte dans la mise en place de l'algorithme de fusion des séquences (section 4.4.2).

4.9 Limites du système

Dans cette section, nous voulons donner un aperçu des limites de notre mécanisme d'inclusion et des fonctionnalités qui pourraient être implantées dans des travaux futurs.

Récursion

Pour l'instant, nous ne supportons pas l'inclusion de méthodes récursives. Sans toutefois supporter l'inclusion récursive non-bornée, il serait intéressant de pouvoir supporter la récursivité d'une méthode, au moins pour quelques niveaux.

Sauts arrière et boucles

Dans la situation actuelle, quand nous rencontrons un saut arrière dans le code de la méthode à inclure, cela marque la fin du chemin que nous incluons. Ce fait constitue une limite pour la construction des séquences que nous voulons les plus longues possible. Il serait intéressant d'essayer de trouver une analyse peu coûteuse et qui nous permettrait d'inclure des chemins constitués à partir de sauts arrière, sans toutefois occasionner la création d'une séquence de longueur non-bornée. Quand nous rencontrons un saut arrière, si nous pouvions détecter s'il s'agit d'une boucle, nous pourrions implanter le déroulement des boucles. Le déroulement des boucles (même partiel), nous permettrait de construire des chemins encore plus longs et réduirait davantage le nombre de répartitions.

Instructions non copiables en mémoire

Évidemment, si tous les codes d'opération de SableVM étaient copiables en mémoire, nous pourrions construire des séquences plus longues et ainsi diminuer davantage le nombre de répartitions. Nous pourrions trouver des astuces qui nous permettraient de rendre toutes les instructions copiables en mémoire. Nous utilisons déjà quelques-unes de ces astuces, comme l'appel des fonctions C par pointeur, qui nous permet d'augmenter le nombre de codes d'opération copiables en mémoire.

4.10 Conclusion

Dans ce chapitre, nous avons expliqué le mécanisme d'inclusion plus en profondeur. Nous avons expliqué à quel moment et comment se fait l'inclusion à un site d'invocation donné. Nous avons détaillé chacune des étapes du processus d'inclusion d'une méthode. Nous commençons par choisir le chemin à inclure et nous construisons la table d'exceptions temporaire qui lui est associée. Ensuite, nous reconstruisons le code de la méthode appelante en y incluant le code du chemin sélectionné. Si nous sommes dans le cas de l'interpréteur linéaire inclusif, nous fusionnons les séquences aux frontières du code inclus. Par la suite, il ne reste qu'à ajuster les adresses dans le code ainsi que la table d'exceptions de la méthode appelante et l'inclusion est terminée. Nous avons ensuite expliqué comment nous mettons à jour les autres invocations de la méthode que nous venons de modifier. Nous avons aussi exposé les contraintes d'implémentation et les limites du système mis en place.

Chapitre 5

Résultats expérimentaux

Dans ce chapitre, nous présentons les résultats de nos expérimentations. Dans la première section, nous présentons notre plateforme de tests. Par la suite, nous démontrons la fonctionnalité de notre système par un micro test de performance (*micro benchmark*). Tous nos résultats sont des mesures dynamiques effectuées au cours de l'exécution de la machine virtuelle SableVM. Nous mesurons le temps d'exécution, le nombre de répartitions exécutées, la longueur moyenne des super-instructions appelées et la qualité de nos informations de profilage. Nous exposons les résultats des tests effectués avec des applications réelles d'envergures comme SableCC (générateur de compilateur) et Soot (*Java Optimisation Framework*). À ces résultats, nous ajoutons ceux obtenus avec la suite de tests SPECjvm98. Finalement, nous discutons des résultats obtenus.

5.1 Plateforme de tests

Toutes nos expériences ont été effectuées sur un seul ordinateur équipé d'un processeur Pentium 4 cadencé à 3Ghz, 1Mo de cache, 1Go de mémoire vive et d'un disque dur 7200RPM de 120Go. Nous avons utilisé le système d'exploitation Debian/GNU Linux avec le noyau version 2.6.16-2-686.

Tous les temps d'exécutions ont été mesurés avec la commande `GNU time` *(system + user)*. Toutes les mesures de temps de ce chapitre ont été calculées en faisant la moyenne de trois exécutions du programme.

5.2 Preuve de concept

Avant de nous lancer dans des tests impliquant des applications plus complexes, nous avons voulu tester la fonctionnalité de notre système sur un plus petit test, nous permettant ainsi de vérifier nos hypothèses. Nous avons écrit un petit programme Java (annexe D) de manière à exploiter au maximum notre système de profilage et d'inclusion partielle. Notre objectif était de vérifier si le fait de construire une très longue séquence et de la ré-exécuter à l'extrême génère effectivement une diminution du temps d'exécution, une diminution du nombre de répartitions ainsi qu'une augmentation de la longueur moyenne des séquences appelées. Ce test ne prétend aucunement être réaliste, mais constitue plutôt une preuve que notre système fonctionne tel qu'attendu et que les gains sont possibles tels que suggérés par notre démarche théorique. Le programme Java exécuté est constitué de la méthode `A` qui appelle la méthode `B` qui contient plusieurs instructions conditionnelles et quelques *switch*. La méthode `B` est appelée à répétition dans le corps de la méthode `A`. La méthode `B` est profilée en cours d'exécution et est plus tard incluse dans `A`. Dans ce test, le chemin inclus dans `A` est toujours le bon et a permis de construire une séquence de 74 instructions à l'intérieur du corps de `A`. Puisque la méthode de `B` est appelée en boucle dans la méthode `A`, cette séquence sera réutilisée intensivement.

	Temps d'exécution (en secondes)	**Nombre de répartitions**	**Longueur moyenne des séquences appelées**
Normal	102,93	3 300 205 485	6,86
Inclusion	74,07	300 191 327	71,96
Gain (%)	-28,04%	-90,90%	1048,98%

Tableau 5.1 – Résultats du micro test de performance.

Comme le montre le tableau 5.1, on obteint une diminution du temps d'exécution de 28,04%, le nombre de répartitions diminue de 90,9% et la longueur moyenne des séquences appelées passe de 6,86 à 71,96 instructions, ce qui constitue une augmentation d'un facteur de 10,49. Évidemment, puisque notre programme Java a été conçu sur mesure pour exploiter les forces de notre système, nous ne pouvons nous attendre à ce genre de résultats avec

des applications réelles. Par contre, ce test démontre que le potentiel de gain au niveau de l'augmentation de la longueur moyenne des séquences appelées, de la réduction du nombre de répartitions et de la diminution du temps d'exécution est bien réel. Ceci constitue notre preuve de concept (*proof of concept*).

5.3 Tests et mesures

5.3.1 La suite de tests

Nous avons utilisé la suite de tests SPECjvm98 [SPE] afin de réaliser nos tests et nos mesures. Cette suite de tests est largement utilisée dans les publications sur les machines virtuelles pour effectuer des mesures empiriques. Nous devons préciser qu'aucun des résultats présentés dans ce mémoire ne représente des mesures de performances SPEC officielles telles que définies dans les règles de SPEC pour la mesure des performances. Nous avons utilisé la suite de tests de SPEC (*compress, jess, db, javac, raytrace, mpegaudio, rtmt*), mais nous avons utilisé nos propres scripts de tests pour lancer les tests et recueillir les résultats.

Nous avons aussi ajouté à cette suite de tests deux autres applications, *SableCC* [Saba] et *Soot* [Soo]. *SableCC* est un générateur de compilateur. Pour nos tests, nous avons donné en entrée à *SableCC* la grammaire du langage Java 1.4. *Soot* est un environnement d'optimisation de Java (*Java Optimisation Framework*). Pour nos tests, nous avons donné en entrée à *Soot* la classe *java.lang.String* à optimiser.

Après avoir effectué plusieurs expérimentations avec différentes valeurs pour les paramètres de notre système, nous avons constaté que la modification de ceux-ci changeait très peu la tendance globale de nos résultats. Tous nos tests ont donc été effectués avec les valeurs ci-bas.

- $method\_profiling\_start\_threshold = 50$
- $method\_profiling\_stop\_threshold = 55$
- $method\_inlining\_start\_threshold = 55$
- $max\_callee\_method\_length = 500$
- $max\_calling\_method\_length = 500$

5.3.2 Temps d'exécution

Les premiers tests que nous avons effectués concernent le temps d'exécution. Notre objectif est de voir si notre système permet d'aller chercher des gains de performance avec des applications réelles. Nous avons implanté une version de notre système qui fait tout le travail du système d'inclusion, mais sans jamais mettre à jour les tableaux de code. De cette façon, cette version nous permet d'approximer le coût du travail de l'inclusion sans jamais pouvoir bénéficier des gains potentiels occasionnés par celui-ci. Notre but est de comparer les résultats obtenus avec cette version que nous appelons *simulation* et la version normale (sans inclusion), afin de tenter de mesurer le coût du mécanisme de profilage et d'inclusion.

Test	Sans inclusion (sec.)	Simulation de l'inclusion (sec.)	Gains de performance (%)	Avec inclusion (sec.)	Gains de performance (%)
compress	63,45	66,28	-4,27%	67,66	-6,23%
jess	30,57	31,81	-3,91%	31,43	-2,75%
db	39,17	42,05	-6,85%	40,88	-4,17%
javac	44,26	47,42	-6,66%	47,46	-6,75%
raytrace	32,22	34,20	-5,79%	36,01	-10,52%
mpegaudio	62,73	63,53	-1,25%	65,01	-3,50%
mtrt	33,20	36,05	-7,91%	36,76	-9,69%
SableCC	600,03	499,10	20,22%	483,29	24,16%
Soot	107,57	116,51	-7,68%	118,12	-8,94%

Tableau 5.2 – Mesures du temps d'exécution et des gains de performance.

Le tableau 5.2 présente les résultats obtenus. Voici dans l'ordre ce que représente les différentes colonnes du tableau : 1) le nom du test, 2) le temps d'exécution sans notre système d'inclusion, 3) le temps d'exécution avec le système d'inclusion, mais sans la mise à jour des tableaux de code, 4) le gain de performance en pourcentage obtenu avec la simulation par rapport à l'exécution normale, 5) le temps d'exécution avec notre système d'inclusion, 6) le gain de performance en pourcentage obtenu avec notre système d'inclusion par rapport à l'exécution normale.

Dans tous les tests, sauf *SableCC*, le temps d'exécution est augmenté de 2,75% à 10,52% par rapport à l'exécution normale sans inclusion. Contrairement à nos attentes, la version

simulation obtient de meilleurs résultats (sauf dans *jess, db et SableCC*) que la version avec inclusion dans la plupart des tests. C'est un phénomène assez curieux considérant que la version simulation ne bénéficie pas de la réduction des répartitions de la version avec inclusion.

Le cas de *SableCC* est assez particulier. On note des gains de performance de 24,16% dans le cas de la version avec inclusion. Il est facile de constater que ce gain n'est pas entièrement généré par le système d'inclusion puisque la version simulation montre un gain de 20,22% sans même que nous réduisions le nombre de répartitions pendant l'exécution.

Sur les architectures modernes, une variance importante du temps d'exécution peut être occasionnée par le comportement difficilement prévisible des mémoires caches au niveau du processeur et des prédicteurs de branchements. En effet, Gu, Clark et Gagnon [GVG05, GVG06] ont montré qu'une variation allant jusqu'à 10% dans le temps d'exécution des machines virtuelles peut être occasionné par des modifications simples du code source, sans toutefois en changer la sémantique. Par exemple, le simple fait d'intervertir des blocs de code, de changer le nom de quelques variables ou d'ajouter un `printf` dans le code peut modifier le comportement de la cache des instructions et peut engendrer des variations dans le temps d'exécution qui sont non négligeables. La disposition du code en mémoire peut agir comme une source de bruit dans les résultats obtenus [GVG04]. Le simple fait de modifier la disposition du code en mémoire, sans même modifier son contenu, engendre des variations importantes dans le temps d'exécution. Il faut donc être prudent avant d'affirmer qu'une technique engendre des gains de performance lorsque ces gains sont inférieurs à 10%. Le gain de 24,16% observé dans notre test avec *SableCC* s'explique certainement par le comportement peu prévisible des caches du processeur Pentium, puisque les mêmes tests exécutés sur un processeur AMD Athlon XP 2500+ nous donnent une dégradation des performances de -6,4%.

Malgré le potentiel noté de notre système (section 5.2), nos mesures sur le temps d'exécution nous mènent à conclure que la diminution des répartitions ne parvient pas à amortir le coût de l'inclusion pour engendrer une diminution du temps d'exécution sur des applications réelles.

5.3.3 Longueurs des super-instructions et nombre de répartitions

Nous avons aussi entrepris de mesurer dynamiquement, pendant l'exécution, la longueur moyenne des super-instructions invoquées, ainsi que le nombre de répartitions. Le tableau 5.3 montre les résultats obtenus. Voici dans l'ordre la signification des colonnes du tableau : 1) le nom du test, 2) la longueur moyenne des super-instructions pour l'exécution normale (sans inclusion), 3) la longueur moyenne des super-instructions pour l'exécution avec inclusion, 4) l'augmentation de la longueur moyenne en % des super-instructions causées par le système d'inclusion, 5) le nombre de répartitions pour l'exécution normale, 6) le nombre de répartition pour l'exécution avec l'inclusion, 7) la diminution en % du nombre de répartitions engendré par le système d'inclusion.

	Longueur moyenne des super-instructions			**Nombre moyen de répartitions**		
Test	**Normal**	**Avec inclusion**	**Augmentation**	**Normal**	**Avec inclusion**	**Diminution (%)**
compress	6,06	11,67	92,57%	2 066 609 818	1 178 658 823	42,97%
jess	2,38	4,62	94,12%	805 748 762	500 238 717	37,92%
db	3,06	4,88	59,48%	1 205 297 509	832 393 235	30,94%
javac	2,86	4,13	44,41%	919 029 850	678 106 604	26,21%
raytrace	1,86	3,50	88,17%	1 175 576 708	708 551 521	39,73%
mpegaudio	9,50	12,51	31,68%	1 216 852 828	947 730 021	22,12%
mtrt	1,87	3,48	86,10%	1 198 886 171	729 806 157	39,13%
SableCC	3,06	4,25	38,89%	3 564 115 855	3 062 356 978	14,08%
Soot	2,95	6,41	117,29%	2 773 476 895	1 363 985 935	50,82%

Tableau 5.3 – Mesures dynamiques de la longueur moyenne des super-instructions et du nombre de répartitions.

Il faut se rappeler que notre objectif de départ était d'augmenter la longueur moyenne des super-instructions et, par le fait même, de réduire le nombre de répartitions entre les instructions interprétées. À la lumière des résultats obtenus dans ces tests, nous pouvons affirmer que nous avons atteint notre objectif. Nous réussissons à augmenter la longueur moyenne des super-instructions dans tous les tests d'un pourcentage variant entre 32% et 117%. Nous avons également réussi à diminuer le nombre de répartitions dans tous les tests et cette diminution varie entre 14,08% et 50,82%.

Il est important de noter que nous n'observons pas, pour un test donné, de corrélation entre le taux de diminution des répartitions et le temps d'exécution. Nous nous attendions à obtenir des gains de performance proportionnels à la diminution des répartitions, mais ce n'est pas le cas. Prenons le cas *Soot*. Ce test obtient la plus grande diminution des répartitions (50,82%), mais lorsque nous observons son temps d'exécution, c'est un des tests avec la plus grande détérioration des performances (-8,94%). Même en diminuant le nombre de répartitions de moitié, nous ne réussissons pas à diminuer le temps d'exécution.

Afin de nous aider à mieux comprendre ces résultats, nous avons entrepris de mesurer la précision de notre système de profilage pour mesurer la fréquence de l'inclusion de mauvais chemins ou de mauvaises méthodes dans le cas des *invokevirtual* et *invokeinterface*.

5.3.4 Précision du profilage

Nous avons mesuré dynamiquement le nombre de super-instructions qui contiennent des points de vérifications et qui sont exécutés au sein de code inclus. Nous avons aussi mesuré le nombre de fois où nous devons quitter le chemin inclus aux points de vérifications pour continuer l'exécution dans le code original de la méthode appelée. Notre objectif est de mesurer le nombre de fois où le chemin inclus ne correspond pas au flot d'exécution. Pour rentabiliser notre système d'inclusion, il est important que les chemins que nous sélectionnons pour l'inclusion correspondent le plus souvent possible au flot d'exécution.

Nous avons aussi mesuré dynamiquement le nombre de fois où nous exécutons des méthodes virtuelles incluses et le nombre de fois où méthode incluse pour un site d'invocation donné est invalide. Nous savons que l'identité de la méthode virtuelle à invoquer est déterminée dynamiquement à l'exécution en fonction du type de l'objet sur lequel l'invocation est effectuée. Or, il est possible que parfois la méthode que nous avons incluse ne corresponde pas à celle qui devrait être invoquée.

Le tableau 5.4 illustre nos résultats. Voici dans l'ordre le contenu de chaque colonne : 1) le nom du test, 2) le nombre total de super-instructions contenant au moins un point de vérification exécuté dans du code inclus, 3) le nombre de fois où le flot d'exécution a dû quitter une super-instruction à un point de vérification (mauvais chemin inclus), 4) le pourcentage de fois où le chemin inclus ne correspond pas au flot d'exécution, 5) le

nombre total de méthodes virtuelles incluses appelées, 6) le nombre de fois où l'identité de la méthode virtuelle incluse n'était pas la bonne, 7) le pourcentage de fois où l'identité de la méthode virtuelle incluse n'était pas la bonne.

	Validité des chemins inclus			Validité des méthodes virtuelles incluses		
Test	**Exécutions de chemins inclus**	**Exécutions de mauvais chemin inclus**	**Mauvais chemin (%)**	**Total des appels virtuels inclus**	**Mauvaise méthode incluse**	**Mauvaise méthode (%)**
compress	67 922 983	18 067 430	26,60%	187 812 834	61	0,00%
jess	41 663 122	19 700 495	47,29%	77 163 164	1 411 353	1,83%
db	88 487 610	24 168 769	27,31%	43 430 061	52	0,00%
javac	45 017 534	9 837 027	21,85%	45 067 837	2 743 155	6,09%
raytrace	4 941 004	650 596	13,17%	144 021 672	788 187	0,55%
mpegaudio	48 738 357	8 505 664	17,45%	32 482 608	2 979 832	9,17%
mtrt	3 387 458	535 809	15,82%	144 262 929	806 018	0,56%
SableCC	713 937 094	241 392 92	33,81%	669 626 692	345 497	0,05%
Soot	210 250 048	15 248 190	7,25%	250 477 749	4 909 221	1,96%

Tableau 5.4 – Mesures dynamiques de la validité des chemins inclus et de l'identité des méthodes virtuelles incluses.

Nous constatons que malgré la naïveté de notre système de profilage, nous obtenons quand même d'assez bons résultats. En moyenne, dans 77% des cas, le flot d'exécution correspond au chemin que nous avons sélectionné pour l'inclusion. En moyenne, dans 98% des exécutions de méthodes virtuelles incluses, l'identité de la méthode incluse correspond à celle qui doit être appelée.

Ces résultats indiquent que la précision de notre système de profilage ne semble pas être la cause des résultats mitigés obtenus dans les tests de temps d'exécution.

5.4 Conclusion

Nous avons démontré dans la section 5.2 que notre système offre un potentiel de gains de performance avec l'interpréteur linéaire inclusif de SableVM. Nous réussissons à réduire le temps d'exécution d'un mini programme Java de 28%. Nous réussissons à diminuer le nombre de répartition de 90% et nous augmentons la longueur moyenne des super-instructions invoquées d'un facteur de 10. Évidemment, ce mini programme Java écrit sur

mesure pour bénéficier de notre système d'inclusion ne représente en rien la réalité d'une véritable application, mais il illustre le potentiel de notre système.

Dans la section 5.3.2, nous avons présenté nos mesures de performances. Nous avons noté que malgré le potentiel de gains de notre système, nos mesures sur le temps d'exécution nous emmènent à conclure que la diminution des répartitions ne parvient pas à amortir le coût de l'inclusion pour engendrer une diminution du temps d'exécution sur des applications réelles.

Par contre, dans la section 5.3.3, nous avons montré, par nos mesures dynamiques sur la longueur moyenne des super-instructions invoquées et sur le nombre de répartitions au cours de l'exécution, que nous avons atteint notre objectif. Nous avions comme objectif d'augmenter la longueur moyenne des super-instructions appelées au sein de l'interpréteur linéaire inclusif de SableVM et de réduire, par le fait même, le nombre de répartitions au cours de l'exécution. Nous réussissons à augmenter la taille moyenne des super-instructions invoquées d'un pourcentage variant entre 32% à 117% et nos réduisons le nombre de répartitions pour tous les tests d'un pourcentage variant entre 14% et 51%.

Dans la dernière section, nous exposons les mesures dynamiques que nous avons effectuées sur l'exactitude des chemins que nous choisissons au moment de l'inclusion d'une méthode. Nous avons aussi quantifié la précision de nos choix quant à l'identité d'une méthode virtuelle à inclure pour un site d'invocation donné. Nous avons mesuré que dans 77% des cas où nous exécutons un chemin dans une zone de code incluse, le chemin correspond au flot d'exécution. Ce qui veut dire que le chemin n'est invalide que dans 23% des cas. Nous avons aussi évalué que dans 98% des cas où nous exécutons une méthode virtuelle incluse, il s'agit de la bonne méthode à exécuter. Ceci signifie que malgré la légèreté et la relative naïveté de notre système de profilage, nos informations de profilage sont de qualité.

Chapitre 6

Travaux reliés

Ce chapitre vise à exposer les travaux réalisés et publiés en rapport avec notre travail de recherche. Le chapitre est divisé en deux parties. Dans la première partie, nous exposons les travaux reliés à l'inclusion partielle. Ces travaux ont tous été faits dans un cadre de compilation, puisqu'à notre connaissance, nous sommes les premiers à intégrer ce genre de mécanisme dans un interpréteur. Dans la deuxième partie, nous allons discuter des principales approches modernes d'optimisation des interpréteurs. Nous abordons principalement l'utilisation des super-instructions, la spécialisation des codes d'opération, la spécialisation des interpréteurs, la prédiction des branchements indirects au niveau matériel et finalement la répartition avec contexte.

6.1 Inclusion partielle

6.1.1 Inclusion partielle et compilation par régions

L'inclusion partielle est un concept relativement nouveau qui existe depuis peu dans l'univers de la compilation. Les premières références qu'il est possible de trouver sur le sujet [HmWHR97, WP02, SYN03] sont reliées au domaine de la compilation par régions (*region based compiling*). Ce type de compilation provient d'un domaine de recherche appelé le parallélisme au niveau instruction (*Instruction-Level Parallelism ILP*). Pour atteindre un plus haut niveau de parallélisme sur ce type d'architecture, il est nécessaire de recourir à des techniques d'analyse et d'optimisation interprocédurales au cours de

la compilation. Une des approches adoptées pour minimiser les coûts élevés associés à l'analyse interprocédurale est la compilation par régions.

Normalement, dans un compilateur, l'unité de compilation est la fonction, mais dans le contexte d'*ILP*, les frontières entre les fonctions cachent souvent des opportunités d'optimisation. Dans ce cas, l'approche habituelle d'inclusion complète des fonctions pour éliminer la barrière entre les fonctions occasionne la génération de trop grandes procédures. La compilation par régions est utilisée pour résoudre ce problème, car elle permet au compilateur de contrôler la taille, tout en mettant en évidence les opportunités d'optimisations et de déplacement du code. On partitionne un programme en unités plus propices aux optimisations et à un meilleur ordonnancement (*scheduling*). Les régions sont créées afin de permettre une meilleure analyse du programme et ainsi mieux refléter son comportement dynamique. La compilation par régions repose sur la collection de traces, car la partition en régions est basée sur des informations de profilage.

Les fondateurs de la compilation par région [HmWHR97] ont proposé les premiers d'inclure un mécanisme d'inclusion partielle pour ce type de compilation. Ce n'est que plus tard que Way et Pollock [WP02] ont réussi à mettre en place dans leur système un algorithme d'inclusion partielle au sein d'une région qui permet de bénéficier des gains occasionnés par une inclusion complète, mais sans l'augmentation habituelle de la taille du code qui vient normalement avec ce type d'inclusion. C'est au moment de la création d'une région que le système effectue l'inclusion partielle des fonctions comprises dans cette région afin d'éliminer la barrière qui existe entre la fonction appelante et la fonction appelée. Pour limiter l'expansion du code, le système effectue seulement l'inclusion du code chaud de la méthode appelée. Le code froid de la méthode appelée est retiré par clonage. Le clonage consiste à copier une zone de code froid dans une nouvelle fonction pour ensuite remplacer la zone de code froide originale par un appel à la méthode nouvellement créée (figure 6.1).

Way et Pollock utilisent comme nous le profilage pour détecter les zones chaudes à inclure. Par contre, leur mécanisme d'inclusion a pour but d'élargir le potentiel d'optimisation en éliminant les frontières entre les méthodes, tandis que le nôtre vise à diminuer le nombre de répartitions et ne permet pas d'effectuer des optimisations supplémentaires sur le code une fois qu'il a été inclus. La façon dont ils effectuent l'inclusion partielle est

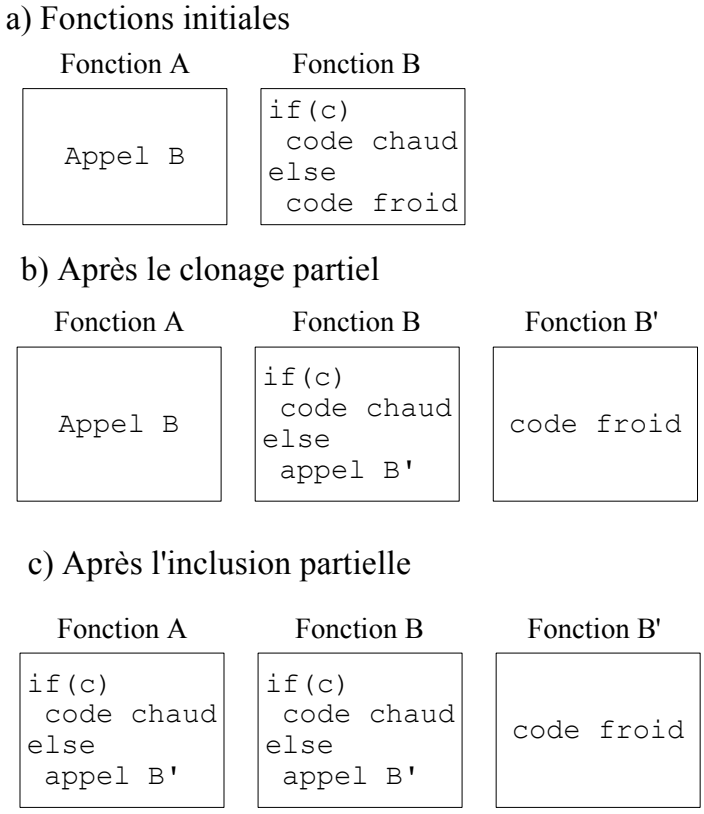

Figure 6.1 – Inclusion partielle par clonage.

différente de la nôtre. Plutôt que d'insérer des points de vérifications dans le code inclu qui permettront d'ajuster le flot d'exécution dans le code original de la méthode appelée si le mauvais chemin a été inclu, ils utilisent la technique du clonage. Notre technique n'inclut pas du tout le code froid dans la méthode appelée tandis que leur technique l'inclut, mais celui-ci est encapsulé dans un appel de méthode.

6.1.2 Inclusion partielle et compilateurs dynamiques adaptatifs

Les compilateurs dynamiques adaptatifs sont au coeur de presque toutes les machines virtuelles à hautes performances. Il s'agit de compilateurs qui sont reliés à un système de profilage servant à leur indiquer les zones de code chaudes à compiler et à optimiser. Par défaut, le code est interprété ou compilé naïvement sans optimisations. À mesure que des parties de code se réchauffent et franchissement un certain seuil, le compilateur dy-

namique entre en action et effectue des optimisions plus agressives sur ces régions. Dans ce type d'environnement dynamique, le temps nécessaire à la compilation est important et doit être réduit au maximum tout en obtenant les meilleures optimisations possible. L'inclusion de méthodes et l'analyse de flot de données sont deux éléments des optimisations possibles qui souffrent énormément de la présence de code froid au sein du code à optimiser. Puisque la majorité des algorithmes d'optimisation ont un temps d'exécution proportionnel à la taille du code à optimiser, il importe de restreindre ce code uniquement au code chaud. Des techniques de compilation dynamique partielle des méthodes ont déjà été développées [Wha01]. Dans ce genre de techniques, les optimisations agressives sont effectuées seulement sur le code chaud.

La technique de compilation par régions (section 6.1.1) et les idées d'inclusion partielle qui lui sont associées ont été intégrées dans un JIT Java [SYN03]. La motivation première est de permettre une meilleure granularité d'optimisation que celle offerte par un type de compilation traditionnelle qui se base sur la méthode comme unité de compilation. L'inclusion complète se fait habituellement avec un certain seuil qui sert de limite maximum pour la taille des fonction qui sont admissibles pour l'inclusion. Ce seuil vient limiter les possibilités d'optimisation pour les grandes fonctions. L'inclusion partielle permet d'inclure des grandes méthodes qui contiennent beaucoup de code froid, puisque seul le code chaud est inclus. L'inclusion partielle permet la construction de régions de code constituées uniquement de code chaud et sur lesquelles on peut lancer des optimisations plus coûteuses. Le JIT utilisant la compilation par régions et l'inclusion partielle [SYN03] va chercher des gains de performance variant en moyenne entre 5% et 7% supérieurs au JIT utilisant la compilation par fonctions traditionnelle.

Dans ce cas, l'inclusion partielle plutôt que complète vise principalement à diminuer la taille du code inclus, de manière à éviter d'effectuer les optimisations coûteuses sur le code froid peu utilisé. Dans notre cas, nous utilisons l'inclusion partielle afin d'éliminer le surcoût associé aux répartitions qui se trouvent aux frontières des blocs de base en incluant le chemin le plus fréquemment emprunté. Le concept d'inclusion partielle est similaire, mais nous avons des objectifs distincts et nous utilisons des moyens différents pour y arriver.

6.1.3 Inclusion partielle par exclusion du code froid

Des travaux plus récents [ZA03, ZA04, ZA05] ont été faits sur l'inclusion partielle des méthodes par l'entremise de l'exclusion de fonctions (*function outlining*). Il est fréquent que lors de l'inclusion d'une méthode, on introduit du code froid provenant de la méthode appelée dans le corps de la méthode appelante (méthode chaude). Il est aussi fréquent qu'une méthode ne puisse être sujette à l'inclusion à cause de sa trop grande taille. Pour pallier à ces problèmes, les auteurs ont développé un système d'inclusion partielle reposant lui aussi sur des informations de profilage qui permettent de distinguer les zones froides et chaudes du code à optimiser.

L'exclusion de fonctions (*function outlining*) n'est qu'une appellation différente pour la technique de clonage illustrée à la figure 6.1. Elle consiste à remplacer une zone de code froid par un appel à une fonction qui contient l'équivalent du code substitué. Ceci permet d'extraire le code froid de la méthode courante. Il existe deux approches pour accomplir cette tâche, l'exclusion indépendante (*independent outlining*) et l'exclusion collective (*collective outlining*). L'approche de l'exclusion indépendante replace chaque zone de code froid par un appel à une fonction, spécifique à chaque zone de code substituée. L'approche de l'exclusion collective regroupe toutes les zones de code froid ensemble et utilise un seul appel de fonction qui contient le regroupement de toutes les zones de code froid retirées de la méthode. Évidemment, l'approche d'exclusion collective nécessite l'ajout de code supplémentaire pour sauter à l'unique appel de fonction qui gère le code froid. On devra aussi, dans ce cas, passer en paramètre à la fonction un drapeau permettant de savoir quelle est la partie de code froid dans la fonction qui devra être exécutée. Leurs travaux indiquent que c'est l'exclusion indépendante qui est la plus efficace et la plus performante.

Pour distinguer les zones de code froides et chaudes, le code est divisé en blocs de base et à chacun de ces blocs sont associées les informations ($size_R$, $freq_R$). Ces valeurs sont respectivement la taille de la région (bloc de base) et la fréquence à laquelle elle a été exécutée. Ils utilisent l'heuristique $benefit_R$ pour déterminer si une région doit être soumise à l'exclusion.

$$size\_ratio_R = \frac{size_R}{size_F} \tag{6.1}$$

$$freq\_ratio_R = \frac{freq_R}{freq_F} \tag{6.2}$$

$$benefit_R = \frac{size\_ratio_R}{freq\_ratio_R} \tag{6.3}$$

Le $size\_ratio_R$ exprime le ratio entre la taille de la région froide par rapport à celle de la fonction dans laquelle elle se trouve. Le $freq\_ratio_R$ est le ratio entre la fréquence d'exécution de la région froide relativement à celle de la fonction dans laquelle elle se trouve. L'heuristique $benefit_R$ favorisera les zones de code de grande taille qui ont une fréquence d'exécution faible. Plus le $benefit_R$ est grand, plus il est impératif d'exclure par *outlining* la région de code. Plusieurs autres heuristiques de températures ont été investiguées dans [ZA03].

Une fois les zones de code froides exclues, l'inclusion normale peut procéder normalement. Le code résultant sera plus petit et ne contiendra que du code chaud et les appels de fonctions qui remplaçent le code froid. La mise en place de l'inclusion partielle permet aux auteurs d'aller chercher un gain variant entre -0,85% et 5,75% par rapport à l'inclusion complète normale.

Dans ce cas, le profilage est utilisé pour déterminer quelles zones de code exclure. Dans nos travaux, nous utilisons le profilage pour déterminer quelles zones de code à inclure. Les auteurs éliminent le code froid en le cachant dans un appel de fonction. Nous avons plutôt choisi de ne pas inclure le code froid et de ne laisser qu'un point de vérification dans le code inclus (section 2.2.4). Évidemment, la principale différence entre notre technique et toutes les autres techniques présentées ici est que nous ne pouvons pas bénéficier d'optimisations supplémentaires dans la méthode appelante une fois la méthode appelée incluse parce que nous ne sommes pas dans un contexte de compilation, mais d'interprétation.

6.2 Optimisation des interpréteurs

6.2.1 Super-instructions et spécialisation des codes d'opération

L'utilisation des super-instructions pour augmenter la performance des interpréteurs est un sujet qui a été abordé par plusieurs auteurs [Pro95, PR98, GEK01, VMK02, Gag02, CEG05, ETKW06]. Les super-instructions sont des nouveaux codes d'opération ajoutés au jeu d'instructions déjà existant et qui effectuent un travail équivalent aux séquences d'instructions les plus communes. Certains construisent et déterminent les super-instructions statiquement à la génération de l'interpréteur [VMK02] et d'autres effectuent comme nous le travail dynamiquement [Gag02].

Tous les interpréteurs performants spécialisent les codes d'opération de leur jeu d'instructions. Les codes d'opération spécialisés sont des nouvelles versions de codes d'opération qui existent déjà, mais qui effectuent un travail spécifique à un opérande quelconque ou à un type précis. Par exemple, SableVM effectue ce genre d'optimisation en spécialisant un sous-groupe d'instructions pour des types précis (section 1.3.3, annexe A). Un autre exemple de ce type de spécialisation est la création de codes d'opération à deux versions, une version lente avec vérifications et une version rapide sans vérifications (section 1.3.6). La spécialisation des codes d'opération a été étudiée et utilisée par plusieurs auteurs [GEK01, CEG05, ETKW06]. Puisque nous avons déjà abordé les super-instructions et la spécialisation au chapitre 2, nous ne nous étendrons pas davantage sur le sujet.

6.2.2 Interpréteurs et systèmes embarqués

Une vaste gamme de produits de consommation comprenant des systèmes embarqués est apparue sur le marché dans les dernières années (ordinateurs de poche, téléphones cellulaires sophistiqués, lecteurs de musique portables, etc.). Les fabricants utilisent souvent des environnements d'exécution Java sur ce type de produits pour pouvoir bénéficier de la portabilité des composants logiciels développés en Java. Nous allons maintenant aborder une technique portable qui permet d'optimiser de manière agressive un interpréteur Java pour un système embarqué. Cette technique permet d'aller chercher des gains pouvant aller jusqu'à 300% ce qui représente des résultats comparables à l'utilisation d'un compilateur

dynamique adaptatif.

Il s'agit d'un procédé connu sous le nom de *sEc*[1] [VMK02]. Cette technique adapte la machine virtuelle embarquée aux propriétés et aux exigences de l'application qu'elle devra exécuter, en générant une *vm* spécialisée et différente pour chaque application. Sur un système embarqué, l'application est spécifique, dédiée et connue d'avance. Le jeu d'instruction de la *vm* est augmenté avec des codes d'opération spécifiques à l'application et une phase statique et agressive d'optimisation qui assure un meilleur couplage entre l'application Java, la *vm* et la plateforme matérielle. *sEc* permet donc de générer une *vm* équipée d'un interpréteur spécialisé en fonction de l'application. Il est important de spécifier que cette technique n'engendre pas de surcoût pendant l'exécution, contrairement à un compilateur dynamique. Un fait à noter est que le compilateur dynamique est beaucoup plus gourmand en ressources (temps *cpu* et mémoire), ce qui est moins convenable dans un environnement embarqué. La génération d'une *vm* sur mesure offre une polyvalence dont ne bénéficient pas les machines virtuelles Java matérielles (*hardware JVM*).

La technique repose sur une analyse statique de l'application à être exécutée sur la *vm* et la collecte d'informations de profilage recueillies dynamiquement pendant l'exécution. Ces informations permettent de construire les codes d'opération qui seront ajoutés au jeu d'instructions de la *vm*. L'analyse détecte les régions de code qui se répètent souvent pour ensuite construire des super-instructions à partir de ces régions, de manière similaire à ce qui est fait dans l'interpréteur linéaire inclusif de SableVM. La différence est que le travail est accompli statiquement. Une fois l'analyse effectuée, il y génération d'une *vm* spécifique dont l'interpréteur supportera les nouveaux codes d'opération. Le procédé génère le code C de la *vm* qui est ensuite optimisé par le compilateur (*gcc*), permettant ainsi aux nouveaux codes d'opération créés d'être optimisés aussi intensivement que désiré. Contrairement à la technique de Piumarta et Riccardi [PR98] qui construit des super-instructions dans le but d'éliminer les répartitions entre les instructions, cette technique élimine non seulement les répartitions, mais optimise aussi les supers instructions.

Une fois le code de la *vm* généré, il faut modifier le code des méthodes de l'application de manière à ce qu'elle utilise les nouveaux codes d'opération qui ont été ajoutés

[1] sEc : *Semantically Enriched Code.*

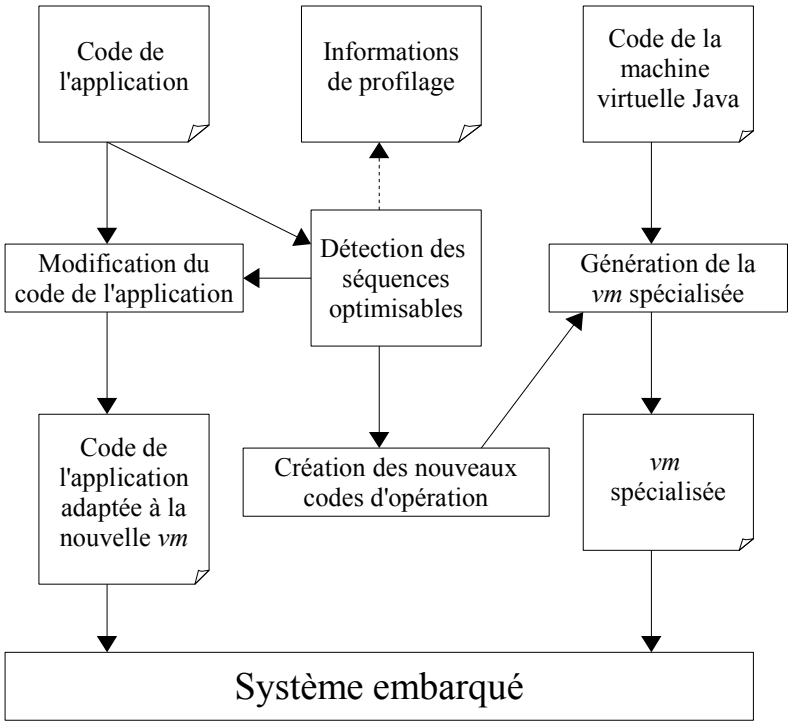

Figure 6.2 – *sEc* : Optimisation d'interpréteurs pour systèmes embarqués.

au jeu d'instruction de la *vm*. La technique prévoit deux façons d'effectuer cette tâche, soit statiquement, en modifiant l'attribut code des méthodes dans les fichiers *.class*, soit dynamiquement, en utilisant un chargeur de classes spécial. La figure 6.2 donne une vue globale de tout le processus.

Contrairement à notre approche, celle-ci effectue tout son travail d'optimisation statiquement, éliminant ainsi le surcoût au moment de l'exécution. Cette méthode est possible dans ce cas, parce qu'ils connaissent à l'avance l'application qui sera exécutée et qu'ils possèdent déjà les informations de profilage avant l'exécution. Comme nous voulions mettre au point un système générique et adaptatif, nous avons favorisé l'approche dynamique. Si nous avions effectué notre travail d'inclusion statiquement, nous aurions par contre pu éliminer le coût de l'inclusion à l'exécution et ainsi bénéficier seulement des gains de l'inclusion.

6.2.3 Prédiction des branchements indirects dans les interpréteurs

Les interpréteurs performants exécutent une grande quantité de branchements indirects[2] et peuvent passer plus de la moitié du temps d'exécution dans de mauvaises prédictions de branchements. Une autre approche à l'optimisation des interpréteurs est celle qui tente d'améliorer la prédiction des branchements lors de l'interprétation. Le meilleur prédicteur de branchements disponible sur les architectures modernes est sans doute les BTB (*Branch Target Buffers*). Ertl et Gregg ont étudié plusieurs interpréteurs sur plusieurs *vm* et avec différents prédicteurs de branchements [EG01]. Ils ont démontré que les BTB font des erreurs de prédiction pour les branchements indirects dans 81%-98% des cas avec un interpréteur par aiguillage et dans 57%-63% des cas avec un interpréteur linéaire direct. Une mauvaise prédiction coûte approximativement 10 cycles sur le Pentium 3 et approximativement 20 cycles sur le Pentium 4. Sachant ceci, il apparaît évident qu'une amélioration des prédictions pourrait engendrer un gain de performance substantiel.

Les interpréteurs par aiguillage et linéaire direct exécutent un branchement indirect entre chaque code d'opération exécuté. Un BTB idéal contient une entrée pour chaque branchement indirect et garde en mémoire la dernière destination pour chacun des branchements. Lorsque le branchement est rencontré de nouveau, le système suppose (prédit) que la destination sera la même que la dernière fois que ce branchement s'est exécuté. Ce procédé fonctionne assez bien à condition que chaque code d'opération d'un tableau de code possède son propre code de répartition à la fin de son implantation.

Dans un interpréteur par aiguillage, tous les codes d'opération partagent le même branchement indirect pour passez au code d'opération suivant. Ce branchement est implicite à la boucle d'interprétation. Le fait que la destination change presque à toutes les fois explique la piètre prédiction des branchements avec ce type d'interpréteur. La situation est par contre moins dramatique dans le cas de l'interpréteur linéaire direct. Chaque implémentation d'un code d'opération se termine par son propre branchement indirect utilisé pour la répartition. Le problème survient quand on rencontre plusieurs fois le même

[2]Contrairement à un branchement direct, un branchement indirect ne contient pas l'adresse de la destination du saut, mais l'adresse de l'adresse de la destination du saut. Le branchement indirect contient donc une indirection supplémentaire.

code d'opération lors de l'exécution. Pour un code d'opération qui se répète, la destination de son branchement indirect changera évidemment en fonction de la position de celui-ci dans le tableau de code. La figure 6.3 illustre le problème lors de la deuxième exécution du tableau de code pour les deux types d'interpréteurs.

Tableau de code		**Prochaine instruction** (interpréteur par aiguillage)		**Prochaine instruction** (interpréteur linéaire direct)	
		Prédiction BTB	Destination réelle	Prédiction BTB	Destination réelle
`lbl:`	`A`	`A`	`B`	`GOTO`	`B`
	`B`	`B`	`A`	`A`	`A`
	`A`	`A`	`GOTO`	`B`	`GOTO`
	`GOTO lbl`	`GOTO`	`A`	`A`	`A`

Figure 6.3 – Comportement du BTB lors de la 2e exécution du tableau de code.

On voit dans la colonne de la prédiction du BTB pour l'interpréteur par aiguillage qu'après l'exécution du code d'opération `A`, le BTB prédit que la prochaine instruction sera `A` et qu'après l'exécution de `B`, la prochaine instruction sera `B`, etc. Le BTB se trompe à tout coup, car le branchement de répartition est partagé par tous les codes d'opération. La figure 6.3 illustre dans le cas de l'interpréteur linéaire direct, la problématique provoquée par la présence de deux codes d'opération `A`. La première ligne montre que le BTB prédit que la prochaine instruction sera `GOTO` alors que c'est `B`. Le BTB effectue cette prédiction parce que la dernière fois que le `A` a été exécuté, la prochaine instruction était effectivement un `GOTO`.

Une solution proposée par Ertl et Gregg [EG03] à ce problème est la réplication des codes d'opération au sein de la *vm*. Dans l'exemple précédent, si la *vm* avait disposé de plusieurs codes d'opération identiques à A, ces répliques auraient pu être utilisées pour améliorer les prédictions du BTB, car chaque code d'opération aurait eu un branchement de répartition distinct (voir figure 6.4). Les auteurs proposent deux types de réplications, statique et dynamique. Dans le cas de la réplication statique, le code des répliques est généré statiquement au moment de la génération de l'interpréteur. Lors de la création des tableaux de codes en cours d'exécution, on utilise toujours la version d'un code d'opération

Tableau de code		**Prochaine instruction** (interpréteur linéaire direct)	
		Prédiction BTB	Destination réelle
`lbl:`	`A1`	`B`	`B`
	`B`	`A2`	`A2`
	`A2`	`GOTO`	`GOTO`
	`GOTO lbl`	`A1`	`A1`

Figure 6.4 – Comportement du BTB avec réplication des codes d'opération.

qui a été le moins utilisé récemment (*round-robin*). Avec l'approche dynamique, c'est en cours d'exécution que l'interpréteur génère dynamiquement au besoin les répliques de manière à assurer que la majorité des codes d'opération soit distincts. Les auteurs utilisent aussi cette technique de réplication pour les interpréteurs avec super-instructions. Ils utilisent plusieurs répliques de celles-ci pour augmenter l'efficacité des prédictions. Une amélioration de performance de 208% a pu être obtenue avec un interpréteur qui utilise un mélange d'interprétation linéaire directe et de super-instructions avec réplications statiques par rapport à sans réplications. Une amélioration de 317% a pu être obtenue avec un interpréteur avec super-instructions et duplications statiques par rapport à sans duplications. Des travaux réalisés tout récemment par Casey, Ertl et Gregg [CEG05] indiquent que la réplication ne donne pas toujours des résultats positifs. Pour obtenir un effet positif sur la prédiction des branchements, il faut utiliser un grand nombre de répliques.

Cette technique vise, comme nos travaux, à optimiser le rendement d'un interpréteur, mais en utilisant une approche complètement différente. Ces travaux démontrent l'importance des prédicteurs des branchements indirects durant l'interprétation. Puisque notre mécanisme d'inclusion partielle réussit dans presque tous les cas à diminuer le nombre de répartitions durant l'exécution, nous pouvons supposer que nous éliminons aussi les inconvénients associés aux mauvaises prédictions associées aux répartitions éliminées. Nous diminuons donc indirectement les mauvaises prédictions effectuées par les prédicteurs de branchements indirects.

6.2.4 Répartition avec contexte

Malgré que la réplication augmente la qualité des prédictions [EG03], elle augmente aussi la taille du corps des codes d'opération, ce qui a pour conséquence de dégrader la performance des mémoires caches [VA04]. Une autre approche existe pour faire face aux problèmes des prédictions des branchements indirects pour les interpréteurs linéaire direct. Il s'agit d'une technique connue sous le nom de répartition avec contexte (*context threading*) [BVZB05]. Selon cette approche, la principale cause des mauvaises prédictions pour les branchements indirects au niveau matériel est le manque de corrélation entre le flot d'exécution virtuel de la *vm* et le `pc` au niveau matériel. Ce problème est connu sous le nom de problème de contexte (*context problem*).

La répartition avec contexte a pour but d'effectuer la répartition entre les codes d'opération, tout en conservant un alignement maximal entre l'état de la *vm* et celui du matériel. Pour les séquences de code octet sans branchements ou sauts (*linear virtual instructions*), les répartitions sont effectuées en utilisant des appels et des retours natifs de manière à aligner le `pc` virtuel et matériel. De cette façon, le flot d'exécution peut mieux être anticipé par la pile des appels au niveau matériel (*hardware return stack*). Les instructions de branchements virtuels sont converties en branchements natifs de manière à bénéficier des ressources matérielles de prédiction des branchements. Cette technique a été implantée dans dans l'interpréteur linéaire direct de SableVM par les auteurs et la répartition avec contexte s'exécute en moyenne 26% plus rapidement sur un Pentium 4 que la répartition sans contexte. Par contre, l'interpréteur linéaire inclusif de SableVM demeure 5% plus rapide en moyenne que le système de répartition avec contexte linéaire direct. La répartition avec contexte réussit à resserrer la cohésion entre l'état virtuel de la *vm* et celui du matériel, engendrant de ce fait des meilleures prédictions de branchements sans toutefois hypothéquer la performance des caches. Le surcoût associé aux appels et retours natifs utilisés pour la répartition dans les blocs de base vient par contre limiter les gains de performance.

Ces travaux montrent l'impact d'une meilleure cohésion entre l'état de la machine virtuelle et celui du matériel pour améliorer la performance des prédicteurs de branchements indirects. Ceci laisse entrevoir les bienfaits que nous pourrions obtenir si nous pouvions

combiner un tel système à nos travaux actuels, constituant ainsi une piste de recherche intéressante pour des travaux futurs.

6.3 Conclusion

Dans ce chapitre, nous avons présenté les travaux qui abordent le thème de l'inclusion partielle. Les similitudes entre ces travaux et les nôtres sont limitées puisque nous sommes les seuls à avoir mis en place un tel système dans un interpréteur. Nous utilisons tous des informations de profilage pour détecter les zones chaudes et motiver nos décisions d'inclusions. Certains utilisent, contrairement à nous, des heuristiques plus complexes que des simples compteurs pour faire leurs choix. La majorité de ces travaux favorisent l'exclusion du code froid par une substitution de celui-ci par un appel de fonction. Nous sommes les seuls à complètement éliminer de l'inclusion le code froid et à utiliser des points de vérifications permettant de sauter au bon endroit dans le code original en cas de mauvais chemin inclus (sortir du code inclus).

Nous avons aussi présenté dans ce chapitre, d'autres approches d'optimisation des interpréteurs. Nous avons présenté une approche qui effectue statiquement un travail similaire à ce que fait SableVM dynamiquement en mode linéaire inclusif. Cette méthode destinée aux systèmes embarqués se base, contrairement à SableVM, sur des informations de profilage pour la construction des super-instructions. Nous avons aussi présenté des travaux qui montrent l'importance de favoriser des bonnes prédictions de branchements indirects.

Ces travaux nous mènent à penser que puisque notre système d'inclusion partielle réduit le nombre de répartitions (donc de branchements indirects) pendant l'exécution, il réduira aussi le taux d'erreurs relié aux prédictions de branchements indirects. Ceci pourrait constituer un facteur contribuant aux gains potentiels de notre approche.

Chapitre 7

Conclusion et travaux futurs

Ce chapitre est divisé en deux parties. Dans la première partie, nous récapitulons et tirons les conclusions finales de notre étude. Dans la deuxième partie, nous exposons les travaux futurs et nous suggérons quelques avenues à explorer pour améliorer notre système.

7.1 Conclusion

Notre intention au cours de cette étude était de vérifier jusqu'où il est possible de pousser l'optimisation des interpréteurs. Cette intention nous a conduite à la conception et l'implantation de notre système d'inclusion partielle des méthodes, ainsi qu'à l'étude de son impact sur des applications réelles.

Dans ce mémoire, nous avons commencé par expliquer quelques notions de base reliées au code octet Java et à son interprétation. Nous avons présenté les astuces les plus communes contenues dans la préparation des méthodes qui permettent d'accélérer l'interprétation. Nous avons aussi présenté l'utilisation des codes d'opération à deux versions (séquences de préparation) qui permet d'accélérer l'interprétation des instructions synchronisées. Nous avons décrit le fonctionnement des trois types d'interpréteurs de base qui ont servi de fondation à notre étude (par aiguillage, linéaire direct, linéaire inclusif).

Avec comme objectif de permettre à un interpréteur linaire inclusif d'augmenter la longueur moyenne de ses super-instructions et ainsi pouvoir diminuer son nombre de répartitions pendant l'exécution, nous avons conçu et implanté un mécanisme d'inclu-

sion partielle des méthodes. Nous avons expliqué en quoi consiste l'inclusion partielle et comment notre mécanisme arrive à augmenter la longueur des super-instructions. Nous avons aussi discuté des difficultés rencontrées lors de l'implémentation du système dans SableVM. Nous avons solutionné plusieurs problèmes reliés au nettoyeur de mémoire, à la synchronisation, aux exceptions, aux sous-routines dans le code octet et à l'impossibilité d'effectuer la copie en mémoire de l'implémentation de certaines instructions du code octet Java. Nous avons résolu tous ces problèmes, ce qui rend possible l'exécution de n'importe quelle application Java sur notre système.

Nous avons effectué une étude de l'impact de notre mécanisme d'inclusion partielle des méthodes avec *SableCC*, *Soot* et la suite de tests *SPECjvm98*.

Malheureusement, nos mesures sur le temps d'exécution nous mènent à conclure que la diminution des répartitions ne parvient pas à amortir le coût de l'inclusion pour engendrer une diminution du temps d'exécution sur des applications réelles.

Par contre, nous atteignons nos objectifs initiaux d'augmenter la longueur moyenne des super-instructions invoquées pour réduire le nombre de répartitions pendant l'exécution. Nous parvenons à augmenter la longueur moyenne des super-instructions invoquées dans tous les tests d'un pourcentage variant entre 32% et 117%. Nous parvenons aussi à réduire le nombre de répartitions pendant l'exécution dans tous les tests d'une valeur variant entre 14% à 50%.

Notre étude nous a emmenés à concevoir et à implanter un système de profilage léger capable de détecter les sites d'invocations les plus chauds et les chemins les plus souvent fréquentés dans les méthodes chaudes. Le système nous permet aussi d'identifier l'identité de la méthode la plus souvent appelée sur un site d'invocation d'une méthode virtuelle ou d'interface. Nous avons illustré et commenté en détail le fonctionnement du système de profilage dans le chapitre 4. Nous y avons aussi présenté les forces et les faiblesses du système ainsi que les principaux problèmes rencontrés (exceptions, synchronisation). Nous avons mesuré la précision de notre système de profilage et nous avons constaté qu'en moyenne, dans 77% des cas, le chemin sélectionné pour l'inclusion correspond au bon choix. Lorsque nous devons choisir parmi plusieurs méthodes pour l'inclusion d'une méthode virtuelle ou d'interface, notre système permet de choisir la bonne méthode en

moyenne dans 98% des cas.

Nous parvenons à atteindre nos objectifs initiaux en augmentant, dans tous les cas étudiés, la longueur moyenne des super-instructions et en diminuant le nombre de répartitions à l'exécution dans un interpréteur linéaire inclusif.

7.2 Travaux futurs

Récursivité

Notre système ne supporte pas l'inclusion des méthodes récursives. Nous pourrions offrir l'inclusion pour ce type de méthode. Si nous pouvions supporter, ne serait-ce qu'un nombre limité d'appels récursif, cela pourrait constituer une valeur ajoutée.

Sauts arrières

Nous avons déjà expliqué, dans la section 4.9, que notre système ne supporte pas la présence de boucles ou de cycles dans le chemin à inclure. Un saut arrière marque donc la fin du chemin. Il serait intéressant lors de travaux futurs de trouver une solution plus élégante à ce problème. Nous pourrions effectuer le déroulement des boucles qui possèdent un nombre constant d'itérations. Ceci permettrait de construire des séquences encore plus longues.

Synchronisation

Il serait préférable de pouvoir effectuer l'inclusion d'une méthode et de pouvoir propager les modifications sans avoir à arrêter l'exécution de tous les autres *threads*. Nous avons choisi une solution simple et fonctionnelle (section 3.5 et 4.7), mais nous pourrions lors de travaux futurs améliorer cette manière de faire.

Instructions non copiables en mémoire

Nous avons déjà commencé à étudier des pistes de solutions pour diminuer le nombre d'instructions non copiables en mémoire. Comme nous l'avons expliqué au chapitre 5 (section 4.9), cette limitation diminue la longueur moyenne des super-instructions que

nous construisons. Il serait primordial de continuer ces investigations et de trouver une solution à ce problème si nous voulons vraiment nous attaquer à l'amélioration de la performance.

Performance

Lors de la conception et de l'implantation de notre système, nous avons concentré nos efforts sur le bon fonctionnement, la simplicité et la facilité d'entretien du code plutôt que sur son optimisation. Nous désirions avant tout obtenir un système fonctionnel et bien conçu. L'optimisation de nos algorithmes et du code est laissé pour des travaux futurs. Évidemment, avant de se lancer dans une optimisation systématique de tout le code, il serait essentiel d'effectuer une étude plus approfondie des coûts de l'inclusion. La première étape serait de détecter les parties de notre système qui engendre les plus grands coûts pour pouvoir ensuite les optimiser.

Annexe A

Codes d'opération typés additionnels

Cette annexe énumère les codes d'opération qui ont été typés par SableVM. Les codes d'opération marqués comme normaux dans la figure A.1 sont ceux qui sont contenus dans le code octet du fichier *.class*. Ceux-ci sont remplacés, lors de la préparation de la méthode, par des versions typées telles qu'illustrées dans la figure A.1.

Normal: LDC **SableVM:** LDC_INTEGER LDC_FLOAT LDC_STRING **Normal:** LDC2_W **SableVM:** LDC_LONG LDC_DOUBLE	**Normal:** PUTFIELD **SableVM:** PUTFIELD_BOOLEAN PUTFIELD_BYTE PUTFIELD_SHORT PUTFIELD_CHAR PUTFIELD_INT PUTFIELD_LONG PUTFIELD_FLOAT PUTFIELD_DOUBLE PUTFIELD_REFERENCE	**Normal:** PUTSTATIC **SableVM:** PUTSTATIC_BOOLEAN PUTSTATIC_BYTE PUTSTATIC_SHORT PUTSTATIC_CHAR PUTSTATIC_INT PUTSTATIC_LONG PUTSTATIC_FLOAT PUTSTATIC_DOUBLE PUTSTATIC_REFERENCE
Normal: GETFIELD **SableVM:** GETFIELD_BOOLEAN GETFIELD_BYTE GETFIELD_SHORT GETFIELD_CHAR GETFIELD_INT GETFIELD_LONG GETFIELD_FLOAT GETFIELD_DOUBLE GETFIELD_REFERENCE	**Normal:** GETSTATIC **SableVM:** GETSTATIC_BOOLEAN GETSTATIC_BYTE GETSTATIC_SHORT GETSTATIC_CHAR GETSTATIC_INT GETSTATIC_LONG GETSTATIC_FLOAT GETSTATIC_DOUBLE GETSTATIC_REFERENCE	**Normal:** NEWARRAY **SableVM:** NEWARRAY_BOOLEAN NEWARRAY_CHAR NEWARRAY_FLOAT NEWARRAY_DOUBLE NEWARRAY_BYTE NEWARRAY_SHOT NEWARRAY_INT NEWARRAY_LONG

Figure A.1 – Typage additionnel des codes d'opération fait par SableVM.

Annexe B

Codes d'opération ajoutés à SableVM

Le tableau B.1 comprend la liste des codes d'opération qui nous avons ajoutés au jeu d'instructions de SableVM pour mettre en place notre système. Le tableau est séparé en deux parties. La première illustre les codes d'opération utilisés pour mettre en place le système de profilage et la deuxième partie liste les codes d'opération ajoutés pour mettre en place le système d'inclusion.

Mécanisme de profilage	Mécanisme d'inclusion
Initiation du mécanisme: INITIAL_INVOKESTATIC INITIAL_INVOKESPECIAL INITIAL_INVOKEVIRTUAL INITIAL_INVOKEINTERFACE **Invocations profilantes:** COUNTING_INVOKESTATIC COUNTING_INVOKESPECIAL COUNTING_INVOKEVIRTUAL COUNTING_INVOKEINTERFACE PROFILING_INVOKESTATIC PROFILING_INVOKESPECIAL PROFILING_INVOKEVIRTUAL PROFILING_INVOKEINTERFACE **Branchements profilants:** PROFILING_IFEQ PROFILING_IFNE PROFILING_IFLT PROFILING_IFGE PROFILING_IFGT PROFILING_IFLE PROFILING_IF_ICMPEQ PROFILING_IF_ICMPNE PROFILING_IF_ICMPLT PROFILING_IF_ICMPGE PROFILING_IF_ICMPGT PROFILING_IF_ICMPLE PROFILING_IF_ACMPEQ PROFILING_IF_ACMPNE PROFILING_IFNULL PROFILING_IFNONNULL PROFILING_TABLESWITCH PROFILING_LOOKUPSWITCH	**Début du code inclus:** INLINED_INVOKESTATIC INLINED_INVOKESPECIAL INLINED_INVOKEVIRTUAL INLINED_INVOKEINTERFACE **Fin du code inclus:** INLINED_RET INLINED_GOTO INLINED_RETURN INLINED_ARETURN INLINED_IRETURN INLINED_LRETURN INLINED_FRETURN INLINED_DRETURN **Points de vérifications:** INLINED_IFEQ INLINED_IFNE INLINED_IFLT INLINED_IFGE INLINED_IFGT INLINED_IFLE INLINED_IF_ICMPEQ INLINED_IF_ICMPNE INLINED_IF_ICMPLT INLINED_IF_ICMPGE INLINED_IF_ICMPGT INLINED_IF_ICMPLE INLINED_IF_ACMPEQ INLINED_IF_ACMPNE INLINED_IFNULL INLINED_IFNONNULL INLINED_TABLESWITCH_CASE INLINED_TABLESWITCH_DEFAULT INLINED_LOOKUPSWITCH_CASE INLINED_LOOKUPSWITCH_DEFAULT INLINED_JSR INLINED_SKIP_NEXT_SEQ

Tableau B.1 – Codes d'opération ajoutés à SableVM pour mettre en place le profilage et l'inclusion.

Annexe C

Exemple réel : code normal vs. profilant

Cette annexe montre les tableaux de code octet normal et profilant pour la méthode `m1`. Il s'agit de code réel exécutable par SableVM dans l'interpréteur par aiguillage (*switch*). Dans la figure C.2, la colonne de gauche représente le code normal qui correspond à la méthode `m1` et la colonne de droite le code profilant qui lui correspond.

```
public static int m1(int a) {

   int result = -12;

   if(a < result) {
      a++;
   }
   else if(a >= result) {
      a--;
   }
   else { a = result + a; }

   switch(a) {
      case 0: { a += 1; break; }
      case 1: { a += 2; break; }
      default: { a += 3; break; }
   }
   result = result + m2(result);

   switch(result) {
      case -1: { a -= 4; break; }
      case 14: { a -= 5; break; }
      default: { a -= 6; break; }
   }
   return result;
}
```

Figure C.1 – Code Java pour la méthode m1.

Code normal:	**Code profilant:**
0x41ec6c08: [LDC_INTEGER]	0x41ec6ee0: [LDC_INTEGER]
0x41ec6c0c: [-12]	0x41ec6ee4: [-12]
0x41ec6c10: [ISTORE_1]	0x41ec6ee8: [ISTORE_1]
0x41ec6c14: [ILOAD_0]	0x41ec6eec: [ILOAD_0]
0x41ec6c18: [ILOAD_1]	0x41ec6ef0: [ILOAD_1]
0x41ec6c1c: [IF_ICMPGE]	0x41ec6ef4: [PROFILING_IF_ICMPGE]
0x41ec6c20: [0x41ec6c38]	0x41ec6ef8: [0x41ec6f14]
	0x41ec6efc: [0]
0x41ec6c24: [IINC]	0x41ec6f00: [IINC]
0x41ec6c28: [0]	0x41ec6f04: [0]
0x41ec6c2c: [1]	0x41ec6f08: [1]
0x41ec6c30: [GOTO]	0x41ec6f0c: [GOTO]
0x41ec6c34: [0x41ec6c6c]	0x41ec6f10: [0x41ec6f4c]
0x41ec6c38: [ILOAD_0]	0x41ec6f14: [ILOAD_0]
0x41ec6c3c: [ILOAD_1]	0x41ec6f18: [ILOAD_1]
0x41ec6c40: [IF_ICMPLT]	0x41ec6f1c: [PROFILING_IF_ICMPLT]
0x41ec6c44: [0x41ec6c5c]	0x41ec6f20: [0x41ec6f3c]
	0x41ec6f24: [0]
0x41ec6c48: [IINC]	0x41ec6f28: [IINC]
0x41ec6c4c: [0]	0x41ec6f2c: [0]
0x41ec6c50: [-1]	0x41ec6f30: [-1]
0x41ec6c54: [GOTO]	0x41ec6f34: [GOTO]
0x41ec6c58: [0x41ec6c6c]	0x41ec6f38: [0x41ec6f4c]
0x41ec6c5c: [ILOAD_1]	0x41ec6f3c: [ILOAD_1]
0x41ec6c60: [ILOAD_0]	0x41ec6f40: [ILOAD_0]
0x41ec6c64: [IADD]	0x41ec6f44: [IADD]
0x41ec6c68: [ISTORE_0]	0x41ec6f48: [ISTORE_0]
0x41ec6c6c: [ILOAD_0]	0x41ec6f4c: [ILOAD_0]
0x41ec6c70: [TABLESWITCH]	0x41ec6f50: [PROFILING_TABLESWITCH]
0x41ec6c74: [0x41ec6cb0]	0x41ec6f54: [0x41ec6f9c]
0x41ec6c78: [0]	0x41ec6f58: [0]
0x41ec6c7c: [1]	0x41ec6f5c: [1]
0x41ec6c80: [0x41ec6c88]	0x41ec6f60: [0]
0x41ec6c84: [0x41ec6c9c]	0x41ec6f64: [0x41ec6f74]
	0x41ec6f68: [0]
	0x41ec6f6c: [0x41ec6f88]
	0x41ec6f70: [0]
0x41ec6c88: [IINC]	0x41ec6f74: [IINC]
0x41ec6c8c: [0]	0x41ec6f78: [0]
0x41ec6c90: [1]	0x41ec6f7c: [1]
0x41ec6c94: [GOTO]	0x41ec6f80: [GOTO]
0x41ec6c98: [0x41ec6cbc]	0x41ec6f84: [0x41ec6fa8]
0x41ec6c9c: [IINC]	0x41ec6f88: [IINC]
0x41ec6ca0: [0]	0x41ec6f8c: [0]
0x41ec6ca4: [2]	0x41ec6f90: [2]
0x41ec6ca8: [GOTO]	0x41ec6f94: [GOTO]
0x41ec6cac: [0x41ec6cbc]	0x41ec6f98: [0x41ec6fa8]
0x41ec6cb0: [IINC]	0x41ec6f9c: [IINC]
0x41ec6cb4: [0]	0x41ec6fa0: [0]
0x41ec6cb8: [3]	0x41ec6fa4: [3]
0x41ec6cbc: [ILOAD_1]	0x41ec6fa8: [ILOAD_1]
0x41ec6cc0: [ILOAD_1]	0x41ec6fac: [ILOAD_1]

Figure C.2 – Code normal et code profilant correspondant.

Code normal (suite):	**Code profilant (suite):**
0x41ec6cc4: [COUNTING_INVOKESTATIC]	0x41ec6fb0: [PROFILING_INVOKESTATIC]
0x41ec6cc8: [15]	0x41ec6fb4: [0x41ec6cc8]
0x41ec6ccc: [1]	0x41ec6fb8: [1]
0x41ec6cd0: [0x41ec63ac]	0x41ec6fbc: [0x41ec63ac]
0x41ec6cd4: [0x80ac2e0]	0x41ec6fc0: [0x80ac2e0]
0x41ec6cd8: [IADD]	0x41ec6fc4: [IADD]
0x41ec6cdc: [ISTORE_1]	0x41ec6fc8: [ISTORE_1]
0x41ec6ce0: [ILOAD_1]	0x41ec6fcc: [ILOAD_1]
0x41ec6ce4: [LOOKUPSWITCH]	0x41ec6fd0: [PROFILING_LOOKUPSWITCH]
0x41ec6ce8: [0x41ec6d28]	0x41ec6fd4: [0x41ec7020]
0x41ec6cec: [2]	0x41ec6fd8: [2]
0x41ec6cf0: [-1]	0x41ec6fdc: [0]
0x41ec6cf4: [0x41ec6d00]	0x41ec6fe0: [-1]
0x41ec6cf8: [14]	0x41ec6fe4: [0x41ec6ff8]
0x41ec6cfc: [0x41ec6d14]	0x41ec6fe8: [0]
	0x41ec6fec: [14]
	0x41ec6ff0: [0x41ec700c]
	0x41ec6ff4: [0]
0x41ec6d00: [IINC]	0x41ec6ff8: [IINC]
0x41ec6d04: [0]	0x41ec6ffc: [0]
0x41ec6d08: [-4]	0x41ec7000: [-4]
0x41ec6d0c: [GOTO]	0x41ec7004: [GOTO]
0x41ec6d10: [0x41ec6d34]	0x41ec7008: [0x41ec702c]
0x41ec6d14: [IINC]	0x41ec700c: [IINC]
0x41ec6d18: [0]	0x41ec7010: [0]
0x41ec6d1c: [-5]	0x41ec7014: [-5]
0x41ec6d20: [GOTO]	0x41ec7018: [GOTO]
0x41ec6d24: [0x41ec6d34]	0x41ec701c: [0x41ec702c]
0x41ec6d28: [IINC]	0x41ec7020: [IINC]
0x41ec6d2c: [0]	0x41ec7024: [0]
0x41ec6d30: [-6]	0x41ec7028: [-6]
0x41ec6d34: [ILOAD_1]	0x41ec702c: [ILOAD_1]
0x41ec6d38: [IRETURN]	0x41ec7030: [IRETURN]
0x41ec6d3c: [PREPARE_INVOKESTATIC]	0x41ec7034: [PREPARE_INVOKESTATIC]
0x41ec6d40: [0x41ec6ccc]	0x41ec7038: [0x41ec6fb8]
0x41ec6d44: [0x41ec61cc]	0x41ec703c: [0x41ec61cc]
0x41ec6d48: [0x80ac2e0]	0x41ec7040: [0x80ac2e0]
0x41ec6d4c: [REPLACE]	0x41ec7044: [REPLACE]
0x41ec6d50: [0x41ec6cc4]	0x41ec7048: [0x41ec6fb0]
0x41ec6d54: [INITIAL_INVOKESTATIC]	0x41ec704c: [INITIAL_INVOKESTATIC]
0x41ec6d58: [GOTO]	0x41ec7050: [GOTO]
0x41ec6d5c: [0x41ec6cd8]	0x41ec7054: [0x41ec6fc4]
Des espaces blancs ont été insérés dans le code normal pour qu'il soit aligné avec le code profilant, assurant ainsi une meilleure lisibilité.	

Figure C.3 – Code normal et code profilant correspondant (suite).

Annexe D

Code Java du micro test de performance

Cette annexe contient le code Java du micro test de performance utilisé pour obtenir les résultats du tableau 5.1 de la section 5.2.

```
public class Micro {

    public static void main(String[] args) {

      int res = 0;
      int val1 = 10;
      int val2 = 20;
      int max = 100000000;

      for(int k = 0 ; k < 3 ; k++) {
        for(int i = 0 ; i < max ; i++) {
          res = m1(val1, val2);
          res += 3;
        }
      }
    }

    public static int m1(int a, int b) {

      switch(a) {
        case 1: b++; break;
        case 30: b += 3; break;
        case 77: b += 7; break;
        case 900: b += 9; break;
```

```
    case 10: b += 10; break;
    default:
      b = 1;
  }

  b += b - 20;
  b += b - 21;
  b += b - 22;

  switch(a) {
    case 1: b++; break;
    case 30: b += 3; break;
    case 77: b += 7; break;
    case 900: b += 9; break;
    case 10: b += 10; break;
    default:
      b = 1;
  }

  b += b - 20;
  b += b - 21;
  b += b - 22;

  switch(a) {
    case 1: b++; break;
    case 30: b += 3; break;
    case 77: b += 7; break;
    case 900: b += 9; break;
    case 10: b += 10; break;
    default:
      b = 1;
  }

  b += b - 20;
  b += b - 21;
  b += b - 22;

  switch(a) {
    case 1: b++; break;
    case 30: b += 3; break;
    case 77: b += 7; break;
    case 900: b += 9; break;
    case 10: b += 10; break;
    default:
      b = 1;
  }

  b += b - 20;
  b += b - 21;
  b += b - 22;
```

```
switch(a) {
  case 1: b++; break;
  case 30: b += 3; break;
  case 77: b += 7; break;
  case 900: b += 9; break;
  case 10: b += 10; break;
  default:
    b = 1;
}

b += b - 20;
b += b - 21;
b += b - 22;

switch(a) {
  case 1: b++; break;
  case 30: b += 3; break;
  case 77: b += 7; break;
  case 900: b += 9; break;
  case 10: b += 10; break;
  default:
    b = 1;
}

b += b - 20;
b += b - 21;
b += b - 22;

switch(a) {
  case 1: b++; break;
  case 30: b += 3; break;
  case 77: b += 7; break;
  case 900: b += 9; break;
  case 10: b += 10; break;
  default:
    b = 1;
}

b += b - 20;
b += b - 21;
b += b - 22;

switch(a) {
  case 1: b++; break;
  case 30: b += 3; break;
  case 77: b += 7; break;
  case 900: b += 9; break;
  case 10: b += 10; break;
  default:
```

```
            b = 1;
        }

        return a;
    } // Fin de la méthode m1

} // Fin de la classe Micro
```

Annexe E

Les codes d'opération du code octet Java

Cette annexe comprend la liste des codes d'opération contenus dans le code octet Java. Les instructions sont ici présentées avec une numérotation qui associe le code d'opération à l'entier qui le représente dans la spécification du fichier *.class*. Nous présentons une brève description de la fonctionnalité de chaque instruction. La spécification complète de ces instructions est disponible dans le chapitre six de la spécification de la machine virtuelle Java [LY99].

0. `nop` : Ne fait rien.
1. `aconst_null` : Ajoute la constante `null` sur la pile.
2. `iconst_m1` : Ajoute la constante `-1` de type *integer* sur la pile.
3. `iconst_0` : Ajoute la constante `0` de type *integer* sur la pile.
4. `iconst_1` : Ajoute la constante `1` de type *integer* sur la pile.
5. `iconst_2` : Ajoute la constante `2` de type *integer* sur la pile.
6. `iconst_3` : Ajoute la constante `3` de type *integer* sur la pile.
7. `iconst_4` : Ajoute la constante `4` de type *integer* sur la pile.
8. `iconst_5` : Ajoute la constante `5` de type *integer* sur la pile.
9. `lconst_0` : Ajoute la constante `0` de type *long* sur la pile.
10. `lconst_1` : Ajoute la constante `1.0` de type *long* sur la pile.
11. `fconst_0` : Ajoute la constante `0` de type *float* sur la pile.
12. `fconst_1` : Ajoute la constante `1.0` de type *float* sur la pile.

13. `fconst_2` : Ajoute la constante `2.0` de type *float* sur la pile.
14. `dconst_0` : Ajoute la constante `0` de type *double* sur la pile.
15. `dconst_1` : Ajoute la constante `1.0` de type *double* sur la pile.
16. `bipush` : Ajoute une valeur de type *byte* sur la pile.
17. `sipush` : Ajoute une valeur de type *short* sur la pile.
18. `ldc` : Ajoute une valeur (index simple) du bassin des constantes sur la pile.
19. `ldc_w` : Ajoute une valeur (index double) du bassin des constantes sur la pile.
20. `ldc2_w` : Ajoute une valeur de type *long* ou *double* (index double) du bassin des constantes sur la pile.
21. `iload` : Ajoute le contenu d'une variable locale de type *integer* sur la pile.
22. `lload` : Ajoute le contenu d'une variable locale de type *long* sur la pile.
23. `fload` : Ajoute le contenu d'une variable locale de type *float* sur la pile.
24. `dload` : Ajoute le contenu d'une variable locale de type *double* sur la pile.
25. `aload` : Ajoute le contenu d'une variable locale de type *reference* sur la pile.
26. `iload_0` : Ajoute le contenu de la variable locale `0` de type *integer* sur la pile.
27. `iload_1` : Ajoute le contenu de la variable locale `1` de type *integer* sur la pile.
28. `iload_2` : Ajoute le contenu de la variable locale `2` de type *integer* sur la pile.
29. `iload_3` : Ajoute le contenu de la variable locale `3` de type *integer* sur la pile.
30. `lload_0` : Ajoute le contenu de la variable locale `0` de type *long* sur la pile.
31. `lload_1` : Ajoute le contenu de la variable locale `1` de type *long* sur la pile.
32. `lload_2` : Ajoute le contenu de la variable locale `2` de type *long* sur la pile.
33. `lload_3` : Ajoute le contenu de la variable locale `3` de type *long* sur la pile.
34. `fload_0` : Ajoute le contenu de la variable locale `0` de type *float* sur la pile.
35. `fload_1` : Ajoute le contenu de la variable locale `1` de type *float* sur la pile.
36. `fload_2` : Ajoute le contenu de la variable locale `2` de type *float* sur la pile.
37. `fload_3` : Ajoute le contenu de la variable locale `3` de type *float* sur la pile.
38. `dload_0` : Ajoute le contenu de la variable locale `0` de type *double* sur la pile.
39. `dload_1` : Ajoute le contenu de la variable locale `1` de type *double* sur la pile.
40. `dload_2` : Ajoute le contenu de la variable locale `2` de type *double* sur la pile.
41. `dload_3` : Ajoute le contenu de la variable locale `3` de type *double* sur la pile.
42. `aload_0` : Ajoute le contenu de la variable locale `0` de type *reference* sur la pile.
43. `aload_1` : Ajoute le contenu de la variable locale `1` de type *reference* sur la pile.
44. `aload_2` : Ajoute le contenu de la variable locale `2` de type *reference* sur la pile.
45. `aload_3` : Ajoute le contenu de la variable locale `3` de type *reference* sur la pile.
46. `iaload` : Ajoute un *integer* provenant d'un tableau sur la pile.
47. `laload` : Ajoute un *long* provenant d'un tableau sur la pile.

48. `faload` : Ajoute un *float* provenant d'un tableau sur la pile.
49. `daload` : Ajoute un *double* provenant d'un tableau sur la pile.
50. `aaload` : Ajoute une *référence* provenant d'un tableau sur la pile.
51. `baload` : Ajoute un *byte* provenant d'un tableau sur la pile.
52. `caload` : Ajoute un *char* provenant d'un tableau sur la pile.
53. `saload` : Ajoute un *short* provenant d'un tableau sur la pile.
54. `istore` : Stocke la valeur de type *integer* sur le haut de la pile dans une variable locale.
55. `lstore` : Stocke la valeur de type *long* sur le haut de la pile dans une variable locale.
56. `fstore` : Stocke la valeur de type *float* sur le haut de la pile dans une variable locale.
57. `dstore` : Stocke la valeur de type *double* sur le haut de la pile dans une variable locale.
58. `astore` : Stocke la valeur de type *reference* sur la pile dans une variable locale.
59. `istore_0` : Stocke la valeur de type *integer* sur la pile dans la variable locale `0`.
60. `istore_1` : Stocke la valeur de type *integer* sur la pile dans la variable locale `1`.
61. `istore_2` : Stocke la valeur de type *integer* sur la pile dans la variable locale `2`.
62. `istore_3` : Stocke la valeur de type *integer* sur la pile dans la variable locale `3`.
63. `lstore_0` : Stocke la valeur de type *long* sur la pile dans la variable locale `0`.
64. `lstore_1` : Stocke la valeur de type *long* sur la pile dans la variable locale `1`.
65. `lstore_2` : Stocke la valeur de type *long* sur la pile dans la variable locale `2`.
66. `lstore_3` : Stocke la valeur de type *long* sur la pile dans la variable locale `3`.
67. `fstore_0` : Stocke la valeur de type *float* sur la pile dans la variable locale `0`.
68. `fstore_1` : Stocke la valeur de type *float* sur la pile dans la variable locale `1`.
69. `fstore_2` : Stocke la valeur de type *float* sur la pile dans la variable locale `2`.
70. `fstore_3` : Stocke la valeur de type *float* sur la pile dans la variable locale `3`.
71. `dstore_0` : Stocke la valeur de type *double* sur la pile dans la variable locale `0`.
72. `dstore_1` : Stocke la valeur de type *double* sur la pile dans la variable locale `1`.
73. `dstore_2` : Stocke la valeur de type *double* sur la pile dans la variable locale `2`.
74. `dstore_3` : Stocke la valeur de type *double* sur la pile dans la variable locale `3`.
75. `astore_0` : Stocke la valeur de type *reference* sur la pile dans la variable locale `0`.
76. `astore_1` : Stocke la valeur de type *reference* sur la pile dans la variable locale `1`.
77. `astore_2` : Stocke la valeur de type *reference* sur la pile dans la variable locale `2`.
78. `astore_3` : Stocke la valeur de type *reference* sur la pile dans la variable locale `3`.
79. `iastore` : Stocke la valeur de type *integer* sur la pile dans un tableau.
80. `lastore` : Stocke la valeur de type *long* sur la pile dans un tableau.
81. `fastore` : Stocke la valeur de type *float* sur la pile dans un tableau.

82. `dastore` : Stocke la valeur de type *double* sur la pile dans un tableau.
83. `aastore` : Stocke la valeur de type *reference* sur la pile dans un tableau.
84. `bastore` : Stocke la valeur de type *byte* sur la pile dans un tableau.
85. `castore` : Stocke la valeur de type *char* sur la pile dans un tableau.
86. `sastore` : Stocke la valeur de type *short* sur la pile dans un tableau.
87. `pop` : Supprime l'opérande sur le haut de la pile.
88. `pop2` : Supprime les deux premières opérandes sur le haut de la pile.
89. `dup` : Duplique l'opérande sur le haut de la pile.
90. `dup_x1` : Duplique l'opérande sur le haut de la pile et l'insère 2 éléments plus bas.
91. `dup_x2` : Duplique l'opérande sur le haut de la pile et mais l'insère 2 ou 3 éléments plus bas
92. `dup2` : Duplique les 2 premières opérande sur le haut de la pile.
93. `dup2_x1` : Duplique les 2 premières opérande sur le haut de la pile et les insère 2 éléments plus bas.
94. `dup2_x2` : Duplique les 2 premières opérande sur le haut de la pile et les insère 2, 3 ou 4 éléments plus bas.
95. `swap` : Échange les 2 opérandes sur le haut de la pile.
96. `iadd` : Additionne (et *pop*) les 2 *integer* sur le haut de la pile et y ajoute le résultat.
97. `ladd` : Additionne (et *pop*) les 2 *long* sur le haut de la pile et y ajoute le résultat.
98. `fadd` : Additionne (et *pop*) les 2 *float* sur le haut de la pile et y ajoute le résultat.
99. `dadd` : Additionne (et *pop*) les 2 *double* sur le haut de la pile et y ajoute le résultat.
100. `isub` : Soustrait (et *pop*) les 2 *integer* sur le haut de la pile et y ajoute le résultat.
101. `lsub` : Soustrait (et *pop*) les 2 *long* sur le haut de la pile et y ajoute le résultat.
102. `fsub` : Soustrait (et *pop*) les 2 *float* sur le haut de la pile et y ajoute le résultat.
103. `dsub` : Soustrait (et *pop*) les 2 *double* sur le haut de la pile et y ajoute le résultat.
104. `imul` : Multiplie (et *pop*) les 2 *integer* sur le haut de la pile et y ajoute le résultat.
105. `lmul` : Multiplie (et *pop*) les 2 *long* sur le haut de la pile et y ajoute le résultat.
106. `fmul` : Multiplie (et *pop*) les 2 *float* sur le haut de la pile et y ajoute le résultat.
107. `dmul` : Multiplie (et *pop*) les 2 *double* sur le haut de la pile et y ajoute le résultat.
108. `idiv` : Divise (et *pop*) les 2 *integer* sur le haut de la pile et y ajoute le résultat.
109. `ldiv` : Divise (et *pop*) les 2 *long* sur le haut de la pile et y ajoute le résultat.
110. `fdiv` : Divise (et *pop*) les 2 *float* sur le haut de la pile et y ajoute le résultat.
111. `ddiv` : Divise (et *pop*) les 2 *double* sur le haut de la pile et y ajoute le résultat.
112. `irem` : Divise (et *pop*) les 2 *integer* sur le haut de la pile et y ajoute le reste.
113. `lrem` : Divise (et *pop*) les 2 *long* sur le haut de la pile et y ajoute le reste.
114. `frem` : Divise (et *pop*) les 2 *float* sur le haut de la pile et y ajoute le reste.
115. `drem` : Divise (et *pop*) les 2 *double* sur le haut de la pile et y ajoute le reste.

116. `ineg` : Remplace l'opérande de type *integer* sur le haut de la pile par sa négation.
117. `lneg` : Remplace l'opérande de type *long* sur le haut de la pile par sa négation.
118. `fneg` : Remplace l'opérande de type *float* sur le haut de la pile par sa négation.
119. `dneg` : Remplace l'opérande de type *double* sur le haut de la pile par sa négation.
120. `ishl` : Remplace l'opérande de type *integer* sur le haut de la pile par sa valeur décalée (décalage arithmétique) par la gauche.
121. `lshl` : Remplace l'opérande de type *long* sur le haut de la pile par sa valeur décalée (décalage arithmétique) par la gauche.
122. `ishr` : Remplace l'opérande de type *integer* sur le haut de la pile par sa valeur décalée (décalage arithmétique) par la droite.
123. `lshr` : Remplace l'opérande de type *long* sur le haut de la pile par sa valeur décalée (décalage arithmétique) par la droite.
124. `iushr` : Remplace l'opérande de type *integer* sur le haut de la pile par sa valeur décalée (décalage logique) par la droite.
125. `lushr` : Remplace l'opérande de type *long* sur le haut de la pile par sa valeur décalée (décalage logique) par la droite.
126. `iand` : Effectue un `AND` avec les 2 *integer* sur le haut de la pile et y ajoute le résultat.
127. `land` : Effectue un `AND` avec les 2 *long* sur le haut de la pile et y ajoute le résultat.
128. `ior` : Effectue un `OR` avec les 2 *integer* sur le haut de la pile et y ajoute le résultat.
129. `lor` : Effectue un `OR` avec les 2 *long* sur le haut de la pile et y ajoute le résultat.
130. `ixor` : Effectue un `XOR` avec les 2 *integer* sur le haut de la pile et y ajoute le résultat.
131. `lxor` : Effectue un `XOR` avec les 2 *long* sur le haut de la pile et y ajoute le résultat.
132. `iinc` : Incrémente une variable locale de type *integer* par une constante.
133. `i2l` : Remplace l'*integer* sur le haut de la pile par sa valeur convertie en *long*.
134. `i2f` : Remplace l'*integer* sur le haut de la pile par sa valeur convertie en *float*.
135. `i2d` : Remplace l'*integer* sur le haut de la pile par sa valeur convertie en *double*.
136. `l2i` : Remplace le *long* sur le haut de la pile par sa valeur convertie en *integer*.
137. `l2f` : Remplace le *long* sur le haut de la pile par sa valeur convertie en *float*.
138. `l2d` : Remplace le *long* sur le haut de la pile par sa valeur convertie en *double*.
139. `f2i` : Remplace le *float* sur le haut de la pile par sa valeur convertie en *integer*.
140. `f2l` : Remplace le *float* sur le haut de la pile par sa valeur convertie en *long*.
141. `f2d` : Remplace le *float* sur le haut de la pile par sa valeur convertie en *double*.
142. `d2i` : Remplace le *double* sur le haut de la pile par sa valeur convertie en *integer*.
143. `d2l` : Remplace le *double* sur le haut de la pile par sa valeur convertie en *long*.
144. `d2f` : Remplace le *double* sur le haut de la pile par sa valeur convertie en *float*.
145. `i2b` : Remplace l'*integer* sur le haut de la pile par sa valeur convertie en *byte*.
146. `i2c` : Remplace l'*integer* sur le haut de la pile par sa valeur convertie en *char*.

147. `i2s` : Remplace l'*integer* sur le haut de la pile par sa valeur convertie en *short*.
148. `lcmp` : Les deux *long* sur le haut de la pile sont comparés (et enlevés de la pile). Le résultat de la comparaison (`val1 > val2`) est mis sur le haut de la pile. Le résultat peut être soit 1, 0 où -1.
149. `fcmpl` : Les deux *float* sur le haut de la pile sont comparés (et enlevés de la pile). Le résultat de la comparaison (`val1 < val2`) est mis sur le haut de la pile. Le résultat peut être soit 1, 0 où -1.
150. `fcmpg` : Les deux *float* sur le haut de la pile sont comparés (et enlevés de la pile). Le résultat de la comparaison (`val1 > val2`) est mis sur le haut de la pile. Le résultat peut être soit 1, 0 où -1.
151. `dcmpl` : Les deux *double* sur le haut de la pile sont comparés (et enlevés de la pile). Le résultat de la comparaison (`val1 < val2`) est mis sur le haut de la pile. Le résultat peut être soit 1, 0 où -1.
152. `dcmpg` : Les deux *float* sur le haut de la pile sont comparés (et enlevés de la pile). Le résultat de la comparaison (`val1 > val2`) est mis sur le haut de la pile. Le résultat peut être soit 1, 0 où -1.
153. `ifeq` : Branche si la comparaison (`val = 0`) de l'*integer* sur le haut de la pile avec 0 est vraie.
154. `ifne` : Branche si la comparaison (`val != 0`) de l'*integer* sur le haut de la pile avec 0 est vraie. La valeur `val` est retirée du haut de la pile.
155. `iflt` : Branche si la comparaison (`val < 0`) de l'*integer* sur le haut de la pile avec 0 est vraie. La valeur `val` est retirée du haut de la pile.
156. `ifge` : Branche si la comparaison (`val >= 0`) de l'*integer* sur le haut de la pile avec 0 est vraie. La valeur `val` est retirée du haut de la pile.
157. `ifgt` : Branche si la comparaison (`val > 0`) de l'*integer* sur le haut de la pile avec 0 est vraie. La valeur `val` est retirée du haut de la pile.
158. `ifle` : Branche si la comparaison (`val <= 0`) de l'*integer* sur le haut de la pile avec 0 est vraie. La valeur `val` est retirée du haut de la pile.
159. `if_icmpeq` : Branche si la comparaison (`val1 = val2`) des deux *integer* sur le haut de la pile est vraie. Les valeurs `val1` et `val2` sont retirées du haut de la pile.
160. `if_icmpne` : Branche si la comparaison (`val1 != val2`) des deux *integer* sur le haut de la pile est vraie. Les valeurs `val1` et `val2` sont retirées du haut de la pile.
161. `if_icmplt` : Branche si la comparaison (`val1 < val2`) des deux *integer* sur le haut de la pile est vraie. Les valeurs `val1` et `val2` sont retirées du haut de la pile.
162. `if_icmpge` : Branche si la comparaison (`val1 >= val2`) des deux *integer* sur le haut de la pile est vraie. Les valeurs `val1` et `val2` sont retirées du haut de la pile.
163. `if_icmpgt` : Branche si la comparaison (`val1 > val2`) des deux *integer* sur le haut de la pile est vraie. Les valeurs `val1` et `val2` sont retirées du haut de la pile.
164. `if_icmple` : Branche si la comparaison (`val1 <= val2`) des deux *integer* sur le haut de la pile est vraie. Les valeurs `val1` et `val2` sont retirées du haut de la pile.

165. `if_acmpeq` : Branche si la comparaison (`val1 = val2`) des deux *référence* sur le haut de la pile est vraie. Les valeurs `val1` et `val2` sont retirées du haut de la pile.
166. `if_acmpne` : Branche si la comparaison (`val1 != val2`) des deux *référence* sur le haut de la pile est vraie. Les valeurs `val1` et `val2` sont retirées du haut de la pile.
167. `goto` : Saute vers à une adresse (index simple).
168. `jsr` : Saute à une sous-routine (index simple).
169. `ret` : Retour d'une sous-routine.
170. `tableswitch` : Accède à une table d'adresse par un index et détermine le branchment à effectuer.
171. `lookupswitch` : Recherche dans une table d'adresse par clé et détermine le branchment à effectuer.
172. `ireturn` : Retourne une valeur de type *integer* d'une méthode.
173. `lreturn` : Retourne une valeur de type *long* d'une méthode.
174. `freturn` : Retourne une valeur de type *float* d'une méthode.
175. `dreturn` : Retourne une valeur de type *double* d'une méthode.
176. `areturn` : Retourne une valeur de type *reference* d'une méthode.
177. `return` : Retourne une valeur de type *void* d'une méthode.
178. `getstatic` : Obtient la valeur du champ statique d'une classe.
179. `putstatic` : Modifie la valeur du champ statique d'une classe.
180. `getfield` : Obtient la valeur du champ d'une instance.
181. `putfield` : Modifie la valeur du champ d'une instance.
182. `invokevirtual` : Invocation d'une méthode d'instance, méthode déterminée selon la classe.
183. `invokespecial` : Invocation d'une méthode d'instance pour les cas spéciaux, les super classes, les méthodes *private* et les constructeurs.
184. `invokestatic` : Invocation d'une méthode de classe (*static*).
185. `invokeinterface` : Invocation d'une méthode d'interface.
186. code inutilisé
187. `new` : Crée une nouvelle instance.
188. `newarray` : Crée une nouvelle instance de tableau.
189. `anewarray` : Crée une nouvelle instance d'un tableau de *référence*.
190. `arraylength` : Obtiens la longueur d'un tableau.
191. `athrow` : Lance une exception ou une erreur.
192. `checkcast` : Vérifie si un objet peut être convertit en un type donné.
193. `instanceof` : Vérifie si un objet est d'un type donné.
194. `monitorenter` : Acquiert le verrou d'exclusion mutuelle pour un objet.
195. `monitorexit` : Relâche le verrou d'exclusion mutuelle pour un objet.

196. `wide` : Élargit la taille d'un index pour une variable locale.
197. `multianewarray` : Crée une nouvelle instance de tableau multidimensionnel.
198. `ifnull` : Branche si la *référence* sur la pile est `null`.
199. `ifnonnull` : Branche si la *référence* sur la pile n'est pas `null`.
200. `goto_w` : Saute vers une adresse (index double).
201. `jsr_w` : Saute à une sous-routine (index double).

Les codes d'opération suivants sont réservés

202. `breakpoint`
254. `impdep1`
255. `impdep2`

Bibliographie

[AAB+05] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project : Building an open-source research community. *IBM Syst. J.*, 44(2) :399–417, 2005.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bel73] James R. Bell. Threaded code. *Commun. ACM*, 16(6) :370–372, 1973.

[BVZB05] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading : A flexible and efficient dispatch technique for virtual machine interpreters. In *CGO '05 : Proceedings of the international symposium on Code generation and optimization*, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society.

[CEG05] Kevin Casey, M. Anton Ertl, and David Gregg. Optimizations for a Java interpreter using instruction set enhancement. Technical Report TCD-CS-2005-61, Department of Computer Science, University of Dublin, Trinity College, August 2005.

[EG01] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par '01 : Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 403–412, London, UK, 2001. Springer-Verlag.

[EG03] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *PLDI '03 : Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 278–288, New York, NY, USA, 2003. ACM Press.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[ETKW06] M. Anton Ertl, Christian Thalinger, Andreas Krall, and TU Wien. Superinstructions and replication in the cacao jvm interpreter. *Journal of .NET Technologies*, 4(1) :25–38, 2006.

[Gag02] Etienne M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, Montreal, Canada, 2002.

[GEK01] David Gregg, M. Anton Ertl, and Andreas Krall. Implementing an efficient Java interpreter. In *HPCN Europe 2001 : Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 613–620, London, UK, 2001. Springer-Verlag.

[GH01] Etienne M. Gagnon and Laurie J. Hendren. SableVM : A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 27–40. USENIX Association, April 2001.

[GH03] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of Java bytecode using preparation sequences. In *Compiler Construction : 12th International Conference, CC 2003*, volume 2622 of *LNCS*, pages 153–159, April 2003.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, 3rd Edition*. The Java Series. Prentice Hall., 2005.

[GVG04] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in JVM performance. In *Component and Middleware Performance Workshop, OOPSLA 2004*, 2004.

[GVG05] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Assessing the impact of optimization in Java virtual machines. Technical Report SABLE-TR-2005-4, Sable Research Group, McGill University, October 2005.

[GVG06] Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06 : Proceedings of the 2nd international conference on Virtual execution environments*, pages 111–121, New York, NY, USA, 2006. ACM Press.

[HmWHR97] Richard E. Hank, Wen mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation : Introduction, motivation, and initial experience. *Int. J. Parallel Program.*, 25(2) :113–146, 1997.

[Hot] HotSpot.
URL : `http://java.sun.com/products/hotspot/whitepaper.html`.

[Kaf] Kaffe.
URL : `http://www.kaffe.org`.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification, 2nd Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Ope] OpenJIT.
URL : `http://www.openjit.org`.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing direct-threaded code by selective inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[Pro95] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *POPL '95 : Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–332, New York, NY, USA, 1995. ACM Press.

[Saba] SableCC.
URL : `http://www.sablecc.org`.

[Sabb] SableVM.
URL : `http://www.sablevm.org`.

[Soo] Soot : A Java Optimization Framework.
URL : `http://www.sable.mcgill.ca/soot`.

[SPE] SPECjvm98 Benchmarks.
URL : `http://www.spec.org/jvm98`.

[SYN03] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *PLDI '03 : Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 312–323, New York, NY, USA, 2003. ACM Press.

[VA04] Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 : Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, pages 42–50, New York, NY, USA, 2004. ACM Press.

[VMK02] K. S. Venugopal, Geetha Manjunath, and Venkatesh Krishnan. sEc : A portable interpreter optimizing technique for embedded Java virtual machine. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 127–138, Berkeley, CA, USA, 2002. USENIX Association.

[Wha01] John Whaley. Partial method compilation using dynamic profile information. In *OOPSLA '01 : Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 166–179, New York, NY, USA, 2001. ACM Press.

[WP02] Tom Way and Lori L. Pollock. A region-based partial inlining algorithm for an ILP optimizing compiler. In *PDPTA '02 : Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 552–556. CSREA Press, 2002.

[ZA03] Peng Zhao and José Nelson Amaral. To inline or not to inline ? Enhanced inlining decisions. In *16th Workshop on Languages and Compilers for*

Parallel Computing, College Station, Texas, October 2003.

[ZA04] Peng Zhao and José Nelson Amaral. Splitting functions. Technical report, Department of Computer Sciences, University of Alberta, Edmonton, Canada, 2004.

[ZA05] Peng Zhao and José Nelson Amaral. Function outlining and partial inlining. In *17th International Symposium on Computer Architecture and High Performance Computing*, pages 101–108, Rio de Janeiro, Brazil, October 2005.