Université de Montréal

Génération efficace de graphes d'appels dynamiques complets

par
Hajar Ikhlef

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

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Novembre, 2011

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

Génération efficace de graphes d'appels dynamiques complets

présenté par:

Hajar Ikhlef

a été évalué par un jury composé des personnes suivantes:

Houari Sahraoui,	président-rapporteur
Bruno Dufour,	directeur de recherche
Marc Feeley,	membre du jury

Mémoire accepté le: . . . . . .

RÉSUMÉ

Analyser le code permet de vérifier ses fonctionnalités, détecter des bogues ou améliorer sa performance. L'analyse du code peut être statique ou dynamique. Des approches combinants les deux analyses sont plus appropriées pour les applications de taille industrielle où l'utilisation individuelle de chaque approche ne peut fournir les résultats souhaités. Les approches combinées appliquent l'analyse dynamique pour déterminer les portions à problèmes dans le code et effectuent par la suite une analyse statique concentrée sur les parties identifiées. Toutefois les outils d'analyse dynamique existants génèrent des données imprécises ou incomplètes, ou aboutissent en un ralentissement inacceptable du temps d'exécution.

Lors de ce travail, nous nous intéressons à la génération de graphes d'appels dynamiques complets ainsi que d'autres informations nécessaires à la détection des portions à problèmes dans le code. Pour ceci, nous faisons usage de la technique d'instrumentation dynamique du bytecode Java pour extraire l'information sur les sites d'appels, les sites de création d'objets et construire le graphe d'appel dynamique du programme. Nous démontrons qu'il est possible de profiler dynamiquement une exécution complète d'une application à temps d'exécution non triviale, et d'extraire la totalité de l'information à un coup raisonnable. Des mesures de performance de notre profileur sur trois séries de benchmarks à charges de travail diverses nous ont permis de constater que la moyenne du coût de profilage se situe entre 2.01 et 6.42.

Notre outil de génération de graphes dynamiques complets, nommé *dyko*, constitue également une plateforme extensible pour l'ajout de nouvelles approches d'instrumentation. Nous avons testé une nouvelle technique d'instrumentation des sites de création d'objets qui consiste à adapter les modifications apportées par l'instrumentation au bytecode de chaque méthode. Nous avons aussi testé l'impact de la résolution des sites d'appels sur la performance générale du profileur.

Mots clés: profilage, analyse du code, analyse dynamique, instrumentation dynamique.

ABSTRACT

Code analysis is used to verify code functionality, detect bugs or improve its performance. Analyzing the code can be done either statically or dynamically. Approaches combining both analysis techniques are most appropriate for industrial-scale applications where each one individually cannot provide the desired results. Blended analysis, for example, first applies dynamic analysis to identify problematic code regions and then performs a focused static analysis on these regions. However, the existing dynamic analysis tools generate inaccurate or incomplete data, or result in an unacceptably slow execution times.

In this work, we focus on the generation of complete dynamic call graphs with additional information required for blended analysis. We make use of dynamic instrumentation techniques of Java bytecode to extract information about call sites and object creation sites, and to build the dynamic call graph of the program. We demonstrate that it is possible to profile real-world applications to efficiently extract complete and accurate information. Performance measurement of our profiler on three sets of benchmarks with various workloads places the overhead of our profiler between 2.01 and 6.42.

Our profiling tool generating complete dynamic graphs, named *dyko*, is also an extensible platform for evaluating new instrumentation approaches. We tested a new adaptive instrumentation technique for object creation sites which accommodates instrumentation to the bytecode of each method. We also tested the impact of call sites resolution on the overall performance of the profiler.

Keywords: Profiling, code analysis, dynamic analysis, dynamic instrumentation.

TABLE DES MATIÈRES

RÉSUMÉ . . . . . **iii**

ABSTRACT . . . . . **iv**

TABLE DES MATIÈRES . . . . . **v**

LISTE DES TABLEAUX . . . . . **viii**

LISTE DES FIGURES . . . . . **ix**

NOTATION . . . . . **x**

DÉDICACE . . . . . **xi**

REMERCIEMENTS . . . . . **xii**

CHAPITRE 1 : INTRODUCTION **1**

1.1 Motivation 1

1.2 Problématique 3

1.3 Contribution 5

1.4 Organisation du mémoire 5

CHAPITRE 2 : ÉTAT DE L'ART **7**

2.1 Introduction 7

2.2 Techniques de profilage 8

2.2.1 Profilage par échantillonnage 8

2.2.2 Profilage exact 10

2.3 Représentation des données 11

2.4 Instrumentation du code 14

2.4.1 Instrumentation du code source 15

2.4.2 Instrumentation du code binaire 17

2.5 Travaux connexes 20
2.5.1 Synthèse 25

CHAPITRE 3 : LE PROFILEUR DYKO 27
3.1 Présentation 27
3.1.1 Approche 27
3.1.2 Caractéristiques 29
3.1.3 Utilisation 30
3.2 Instrumentation du bytecode 30
3.2.1 Défis de l'instrumentation 31
3.2.2 Outils d'instrumentation 33
3.2.3 Exemple d'instrumentation avec ASM 33
3.3 Structures de données de profilage 37
3.3.1 Noeud 37
3.3.2 Graphe d'appels 39
3.4 Étendue du profilage 41
3.4.1 Couverture des classes et méthodes 41
3.4.2 Couverture des sites d'appel 42
3.4.3 Couverture des sites de création d'objets 43
3.5 Implémentation 43
3.5.1 Version initiale 43
3.5.2 Retrait de la pile 44
3.5.3 Réduction de l'usage de mémoire 45
3.5.4 Gestion des sites de création d'objets 45
3.5.5 Résolution des sites d'appels 46
3.6 Exemple de transformation 47

CHAPITRE 4 : ÉVALUATION 51
4.1 Environnement d'expérimentation 51
4.1.1 Matériel 51
4.1.2 Benchmarks 52

4.1.3 Démarche 53
4.2 Métriques 56
4.2.1 Temps d'instrumentation 57
4.2.2 Facteur de ralentissement 58
4.3 Résultats 58
4.3.1 Facteur de ralentissement 59
4.3.2 Temps d'instrumentation 63
4.4 Comparaison 68

CHAPITRE 5 : CONCLUSION 72
5.1 Profilage avec *dyko* 72
5.2 Performance 73
5.3 Travaux futurs 74
5.3.1 Profilage de mémoire 74
5.3.2 Profilage partiel 74
5.3.3 Profilage adaptatif 74
5.3.4 Profilage avec contexte 75

BIBLIOGRAPHIE 76

LISTE DES TABLEAUX

2.I Tableau comparatif de quelques travaux existants. 25

4.I Description des benchmarks de la série SPECjvm98 52
4.II Description des benchmarks de la série SPECjvm2008. 54
4.III Description des benchmarks de la série DaCapo. 55
4.IV Temps d'instrumentation des séries SPECjvm98, SPECjvm2008 et DaCapo en secondes 64
4.V Moyenne de ralentissement pour SPECjvm98, SPECjvm2008 et DaCapo 68

LISTE DES FIGURES

2.1 Exemple d'un graphe d'appels (*Call Graph*), d'un arbre d'appel (*Call Tree*) et d'un arbre de contextes d'appels (*Calling Context Tree*) 12
2.2 Architecture du système de Filman et al. [39] 16

3.1 Organisation de *dyko* 28
3.2 Exemple avec Bytecode Outline 31

4.1 Facteur de ralentissement pour SPECjvm98 59
4.2 Facteur de ralentissement pour SPECjvm2008 60
4.3 Facteur de ralentissement pour DaCapo 60
4.4 Résultats de performance pour SPECjvm98 65
4.5 Résultats de performance pour SPECjvm2008 66
4.6 Résultats de performance pour DaCapo 67

NOTATION

JVM	Java Virtual Machine, machine virtuelle Java
VM	Virtual Machine, machine virtuelle
JVMTI	Java Virtual Machine Tool Interface
CT	Call Tree, arbre d'appels
DCT	Dynamic Call Tree, arbre d'appels dynamique
CG	Call Graph, graphe d'appels
DCG	Dynamic Call Graph, graphe d'appels dynamique
CCT	Calling Context Tree, arbre de contextes d'appels
PCCT	Partial Calling Context Tree, arbre de contextes d'appels partiel
ACCT	Approximate Calling Context Tree, arbre de contextes d'appels approximatif

À mes chers parents.

REMERCIEMENTS

Je tiens à remercier vivement mon directeur de recherche, Dr Bruno Dufour, de m'avoir procuré tout le support requis pour mener à bien mes activités de recherche. Je le remercie également pour ses conseils et ses orientations qui m'ont guidé tout au long de ma maitrise.

Je remercie respectueusement les membres du jury, Dr Houari Sahraoui et Dr Marc Feeley d'avoir accepté à émettre leurs conseils et jugements sur ce travail.

Mes remerciements également envers les différents organismes subventionnaires qui ont contribué au financement de cette étude, la faculté des études supérieures de l'Université de Montréal, le conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) et le département d'informatique et de recherche opérationnelle (DIRO) de l'Université de Montréal. Je les remercie pour l'appui financier qu'ils apportent aux étudiants et chercheurs, ce qui contribue considérablement à la dynamique de la recherche universitaire.

Je remercie tous les membres de l'équipe du laboratoire génie logiciel GÉODES pour leur sympathie, leur esprit de cohésion et l'agréable ambiance qu'ils y procurent.

Je remercie chaleureusement mes parents pour leur amour et support inconditionnels tout au long de mes études. Je remercie également mes amis de Montréal pour les moments agréables, ainsi que mes amis de France, du Maroc et d'ailleurs pour leurs continuels encouragements.

Merci à toutes et à tous.

CHAPITRE 1

INTRODUCTION

Produire du logiciel fiable, le maintenir et le faire évoluer devraient être la préoccupation principale d'un informaticien. On peut classifier l'exercice de ce dernier en deux catégories généralisées. D'une part, la tâche de conception des programmes consiste à appliquer les méthodes de conception et à choisir les langages et environnements appropriés. D'autre part, la tâche d'analyse des programmes concerne la détection des erreurs, l'affirmation d'absence d'erreurs, aussi bien que la compréhension des programmes.

Cependant, avec la révolution technologique qu'a connue le secteur, la performance des ordinateurs ainsi que la taille des données traitées sont passées à une échelle supérieure. De plus, les produits logiciels se sont vus également affectés par ce changement. Le nombre de lignes de code ainsi que le nombre d'erreurs et de bogues reportés sont généralement très élevés[1]. La réutilisation de bibliothèques, de cadres d'applications et d'autres composants, est fortement présente dans le développement des logiciels industriels modernes. Développer des applications de grande complexité est devenu ainsi d'une rapidité et d'une simplicité remarquables. Par contre, cette façon de développer entraîne une mauvaise compréhension du comportement des logiciels du côté des développeurs, et rend les tâches de maintenance et de débogage très ardues. Les méthodes classiques de vérification manuelle, comme par lecture de code, ne sont plus applicables. Il est donc nécessaire de fournir des outils aux développeurs pour accomplir ces tâches.

1.1 Motivation

L'analyse du code source est l'extraction automatisée d'informations concernant un programme à partir de son code source ou artefact généré depuis son code source [21]. En d'autres termes, l'analyse du code permet de déduire automatiquement une ou plu-

[1] Un projet logiciel comme Eclipse a reçu plus de 2.764 rapports de bogues sur une période de 3 mois (entre le 01/10/2009 et le 01/01/2010) ; Mozilla et GNOME ont reçu respectivement 6.976 et 3.263 rapports sur la même période.

sieurs propriétés du programme analysé par application mécanique d'outils et techniques diverses.

Il existe différentes techniques d'analyse du code, utilisées selon le besoin. Identifier les failles de sécurité, détecter les défauts nuisant à la qualité ou optimiser la performance du logiciel, sont tous de bonnes raisons pour mettre le code sous la loupe. Une analyse peut être statique, dynamique ou parfois une combinaison des deux.

L'analyse statique permet de statuer sur le comportement général du programme à partir de son code source ou compilé. Elle produit automatiquement un résultat unique par programme qui illustre toutes ses exécutions possibles. Le résultat de cette analyse est unique puisqu'elle ne prend pas en considération les entrées du programme. L'analyse statique permet de déduire des propriétés vraies pour toutes les exécutions d'un programme sans réellement l'exécuter. L'analyse statique est souvent utilisée pour détecter très tôt les situations à problèmes. Elle peut par exemple renseigner sur les structures des classes et méthodes pour identifier des irrégularités dans la programmation, des désaccords avec la conception ou détecter certains types de bogues. Cependant, représenter toute l'information nécessaire aux calculs d'une analyse statique complexe est souvent difficile ou même impossible à produire dans un temps et avec des ressources raisonnables. Tel est le cas des applications de grande taille.

Contrairement à l'analyse statique qui produit des résultats généralisés, l'analyse dynamique s'avère plus spécifique. Elle informe sur les propriétés d'une seule exécution du programme. Analyser une seule exécution peut être parfois d'une grande utilité. Par exemple, lorsqu'une exécution présente des résultats incorrects ou résulte en une performance médiocre, l'analyse de cette exécution en particulier permet d'identifier le problème. L'analyse dynamique collecte des informations pendant l'exécution du programme et produit des données précises comme le nombre d'invocations ou les types d'objets créés. De telles informations sont très utiles pour l'étude du comportement du logiciel et ne peuvent guère être fournies par l'analyse statique. Néanmoins, analyser dynamiquement une application augmente considérablement son temps d'exécution. Aussi, la taille des données collectées lors de l'exécution devient vite ingérable pour les applications de grande taille.

L'analyse combinée [37, 40] (*blended analysis*) est une combinaison des deux types d'analyses statique et dynamique. Le but de l'analyse combinée est d'aboutir à la précision de l'analyse dynamique tout en réduisant son impact sur le temps d'exécution. L'analyse combinée applique des techniques d'analyse statique sur un sous-ensemble de l'application déterminé par l'analyse dynamique. L'analyse combinée s'intéresse à étudier les propriétés d'un ensemble fini d'exécutions concrètes au lieu de toutes les exécutions possibles comme est le cas pour l'analyse statique. En limitant l'étendue de l'analyse statique à une portion exécutée de l'application, il devient possible d'analyser des programmes de taille industrielle qui ne peuvent être analysés par une analyse statique traditionnelle. De plus, en excluant de l'analyse le code non-exécuté, on augmente la précision des résultats de l'analyse qui représente par conséquent de plus près le comportement du logiciel contrairement à une analyse purement statique qui délivre une vue généralisée.

L'analyse combinée nécessite la génération d'un graphe d'appel dynamique qui représente l'information concernant des appels de méthodes effectués durant l'exécution du programme. Les outils existants de génération des graphes d'appels dynamiques souffrent de plusieurs incommodités. Soit l'outil augmente considérablement le temps d'exécution du programme, soit il fournit des informations imprécises ou incomplètes.

Ce travail a pour but de procurer un outil de génération de graphes d'appels dynamiques qui compense les inconvénients des outils existants.

1.2 Problématique

Dans ce mémoire, nous nous intéressons à l'analyse des programmes de grande taille tels que les applications industrielles en Java. La nature et la complexité de ce type de logiciels présentent un challenge intéressant aux chercheurs en matière de profilage et d'analyse de code en général. En effet, les analyses statiques et dynamiques utilisées séparément sont insatisfaisantes ou même parfois inadéquates.

D'un côté, l'analyse statique, lorsqu'elle est applicable, livre une représentation du code sans l'exécuter, ce qui manque de précision pour statuer sur le comportement du

programme. De l'autre, l'analyse dynamique ralentit l'exécution des programmes ou limite la quantité de l'information fournie. L'analyse dynamique accumule souvent une quantité de données énorme dans le cas des applications industrielles, proportionnelle habituellement au nombre d'événements observés à l'exécution. En limitant la collecte d'informations, le résultat de l'analyse n'est pas suffisant pour induire le comportement du logiciel. Ainsi l'analyse combinée vient compenser ces lacunes par union des deux techniques statique et dynamique.

L'application de l'analyse combinée nécessite un graphe d'appel dynamique représentant tous les appels de méthodes, leurs sites d'appels, ainsi que l'information sur les sites d'allocations d'objets et les types des objets alloués par méthode. Un graphe d'appel est un graphe orienté où les nœuds représentent les méthodes et les arcs orientés représentent les appels entre les méthodes. Un graphe d'appel dynamique décrit l'ensemble des appels de méthodes provoqués par une exécution du programme.

Toutefois, les outils de génération de graphes d'appels dynamiques existants ne fournissent pas la totalité de l'information nécessaire pour l'analyse combinée ou dégradent considérablement le temps d'exécution des programmes. Certains ne couvrent pas toutes les classes et méthodes appelées pour limiter l'impact sur l'exécution et s'avèrent ainsi incomplets. D'autres manquent de précision dans les informations fournies en cas d'exception lors de l'exécution. Ceux-ci ne produisent pas les sites d'appels ou d'allocations, ou ne représentent pas sur le graphe les appels aux méthodes *native*[2] ni les méthodes appelées par des méthodes *native*.

Ce projet se concentre sur la production d'un profileur qui génère des graphes d'appels dynamiques complets agrémentés par de l'information sur les sites d'appels et d'allocations, ainsi que le nombre et les types d'objets alloués. Notre outil est conçu de façon à combler les lacunes d'imprécision et d'incomplétude des outils existants tout en gardant un minimum d'impact sur le temps d'exécution.

[2] méthodes déclarées native en Java.

1.3 Contribution

Les contributions apportées par ce travail se résument en trois points.

Premièrement, la contribution principale est la conception et l'implémentation de notre nouveau profileur nommé *dyko*, capable de produire des graphes d'appels dynamiques et complets pour une exécution concrète. Autre que les appels de méthodes, l'outil rajoute de l'information aux graphes telle que les sites d'appels et sites de création d'objets. La méthodologie suivie pendant le développement de l'outil est guidée par les propriétés présentes dans la majorité des applications industrielles orientées objet pour réduire l'impact du profilage sur l'exécution.

Deuxièmement, l'outil a été conçu pour permettre de prototyper et d'évaluer rapidement de nouvelles stratégies de profilage. Il peut facilement être amélioré pour rajouter plus d'information aux graphes générés ou pour rehausser la performance. Nous avons fait preuve de la flexibilité de *dyko* avec l'ajout de deux nouvelles approches. La première est une approche non-uniforme qui adapte individuellement le traitement au code source de chaque méthode du programme profilé, et la deuxième est la résolution des sites d'appels.

Troisièmement, nous établissons des mesures de performance de l'outil sur les séries de benchmarks SPECjvm98[16], SPECjvm2008[49] et DaCapo[15]. Nous présentons également une comparaison avec quelques outils existants en terme de performance. Nous montrons aussi l'impact des optimisations appliquées lors du développement de l'outil, depuis la version initiale jusqu'à la dernière version en s'attardant pour justifier les choix adoptés et mettre l'accent sur la valeur ajoutée de chaque changement.

1.4 Organisation du mémoire

Le reste de ce document est organisé comme suit. Nous commençons par discuter des travaux existants au chapitre subséquent. Le chapitre 3 présente en détails notre approche et explique certaines directions adoptées lors du développement de notre outil. Il présente également les détails d'implémentation et caractéristiques de l'outil. Nous enchaînons ensuite avec le chapitre 4 qui traite l'évaluation par tests de performances de

quelques benchmarks. Enfin, nous achevons avec une récapitulation et des perspectives de recherche dans le chapitre 5.

CHAPITRE 2

ÉTAT DE L'ART

2.1 Introduction

Le profilage, ou *profiling* en anglais, s'intéresse à l'étude du comportement du logiciel en utilisant des informations sur l'exécution. Le profileur est l'outil qui collecte ou analyse les données de l'exécution. Un profil est l'ensemble des attributs et fréquences associés aux événements survenus lors de l'exécution. Un profileur classique surveille l'exécution, mesure le temps consacré à des portions du code ou rapporte la durée d'exécution des méthodes et leurs fréquences d'appels. Certains profileurs fournissent de l'information additionnelle comme des données sur la performance ou l'utilisation de mémoire.

Parmi les cas d'utilisation du profilage figurent l'optimisation par rétroaction (*feedback directed optimization*) [12, 26, 28], le débogage et isolement des bogues [56], les tests de couverture (*coverage testing*) [73] et la compréhension des interactions programme / architecture [43]. Idéalement, un profileur doit fournir de l'information précise, causer un minimum de ralentissement, converger rapidement pour décider des candidats à optimiser, être flexible et portable, travailler en toute transparence et occuper un minimum d'espace mémoire.

Dans le profilage, un compromis existe entre l'information collectée et le ralentissement engendré par rapport au temps initial d'exécution. Il s'agit souvent de sacrifier la précision au profit de la performance ou vice-versa, lors de la collecte d'information et la construction d'une structure de données contenant les résultats du profilage. On peut classer les techniques de profilage en deux catégories. La première est le profilage exact, souvent à base d'instrumentation, les profileurs utilisant cette technique peuvent atteindre une haute précision mais à un coût relativement élevé. La deuxième est le profilage par échantillonnage qui consiste à ne recueillir qu'un échantillon représentatif de l'information. Cette technique a tendance à avoir un petit ralentissement mais manque

de précision. Cependant cette subdivision n'est pas toujours claire comme il existe des profileurs combinant les deux [9, 13, 78]. Une trace d'exécution, un arbre d'appels, un graphe d'appels ou un arbre de contextes d'appels sont des représentations utilisées par les profileurs. Chaque profileur choisit judicieusement la technique de profilage à adopter ainsi que la structure de données à construire selon le niveau de précision demandé et la disponibilité des ressources.

Ce chapitre commence par des généralités sur les techniques et types de profileurs parmi ceux existants. Nous enchaînons ensuite avec les différentes structures de données qu'emploient les profileurs. Puis nous consacrons une section à l'instrumentation, ses types et ses utilisations. Après avoir fait le tour des éléments auxquels touchent notre travail, nous discutons dans la dernière section en détail de divers travaux connexes.

2.2 Techniques de profilage

C'est depuis une quarantaine d'années qu'il a été constaté que la qualité du code pourrait être améliorée en utilisant des profils de l'exécution pour guider les optimisations [48]. Le profilage est une forme d'analyse dynamique des programmes qui étudie le comportement du logiciel en collectant l'information pendant l'exécution. Les profileurs diffèrent selon la granularité des données utilisées pour produire le profil du logiciel et selon la méthode employée pour collecter l'information. On peut distinguer entre le profilage par échantillonnage et le profilage exact.

2.2.1 Profilage par échantillonnage

Les profileurs statistiques emploient l'échantillonnage statistique (*sampling*), par exemple pour identifier les hot-spots dans un programme. Le principe commun du profilage par échantillonnage est comme suit. Un compteur temporel interrompt l'exécution du programme à intervalles réguliers. À l'interruption, un échantillon des méthodes présentes sur la pile d'exécution est créé et enregistré. Lors de l'implémentation d'un profileur échantillonneur, deux facteurs sont à considérer ; (i) la méthode de déclenchement de l'échantillonnage et (ii) la technique d'échantillonnage. Les profileurs varient selon

les deux mécanismes adoptés.

Le déclenchement du processus d'échantillonnage peut être provoqué par des fonctions du système d'exploitation comme la fonction *setitimer*, par des événements fournis par les moniteurs de performance matérielle HPM (*Hardware Performance Monitors*) ou par des événements des machines virtuelles. Des explications plus détaillées sur chacune des techniques suivront.

L'échantillonnage est soit direct soit utilise l'attente active (*polling*). Il est direct lorsqu'il inspecte les threads en cours d'exécution, accède directement à la pile d'exécution et récupère la liste des méthodes courantes. L'approche de l'attente active manie un bit dans la machine virtuelle pour permettre l'échantillonnage. Les threads de l'application profilée vérifient ce bit à des points bien précis appelés *polling points*, avant de déclencher le processus d'échantillonnage. L'échantillonnage direct reste l'approche la plus utilisée vue la précision limitée de l'attente active. Bien que le compteur manipulant le bit opère sur des intervalles réguliers, l'échantillonnage ne commence qu'une fois le polling point suivant est atteint. Ceci engendre des délais variables entre la prise des échantillons. De plus, si le point se trouve après un appel à du code natif qui ne contient pas généralement des polling points, un long temps lui sera accrédité à tord.

2.2.1.1 Types des profileurs échantillonneurs

À base de temps :

L'échantillonnage à base de temps, *timer-based*, utilise généralement des fonctions du système dont la fonction *setitimer*. Cette technique cause un faible ralentissement à l'exécution et peut être utilisée pour toutes les formes du code. Cependant elle souffre de certaines limitations. (i) Ne procure pas assez de points de collecte de données ; La granularité du compteur temporel dépend des propriétés du système d'exploitation. Linux 2.6 par exemple fournit une granularité de 4ms permettant 250 échantillons par seconde. Une version plus ancienne a une granularité de 10ms seulement, ce qui ne laisse place que pour 100 échantillons par seconde. (ii) Manque de robustesse entre différentes implémentations matérielles ; Les mi-

cro architectures et les vitesses d'horloge diffèrent d'une implémentation à une autre, par conséquent une machine rapide peut exécuter plus d'instructions entre deux interruptions de compteur. Le profileur va rassembler de moins en moins d'information avec des microprocesseurs pouvant atteindre des performances supérieures.

À base d'événements HPM :

Les moniteurs de performance matérielle (HPM) sont des registres intégrés dans les microprocesseurs modernes spécialement pour mesurer les activités matérielles. Chaque compteur peut être programmé pour surveiller un type d'événement. Au débordement du compteur, le moniteur envoie un signal à la machine virtuelle qui capture immédiatement un échantillon. Les événements supportés par les HPMs et la façon de les programmer varient selon le constructeur. Le coût d'accès aux données des moniteurs diffère également entre les micro architectures, néanmoins il est généralement faible. Le profilage à base d'HPMs permet de rassembler des échantillons à une granularité plus fine menant ainsi à une meilleure précision.

À base d'événements VM :

Les profileurs à base d'événements, *event-based*, utilisent les événements des machines virtuelles ou des langages comme .Net [60], Python [41], etc pour déclencher l'échantillonnage ou exécuter du code d'instrumentation. À savoir, Java est muni de l'outil JVMTI [6] (Java Virtual Machine Tool Interface) qui interrompt la machine virtuelle sur capture des événements comme des appels de méthodes, le chargement de classe, l'entrée ou la sortie de threads. Sur attrape d'événement, le profileur peut procéder à l'échantillonnage. Cependant certains événements ne peuvent pas être observé du à la limite de l'étendue des événements fournis.

2.2.2 Profilage exact

Contrairement au profilage par échantillonnage où le profileur passe la plupart du temps au repos et s'active périodiquement (toutes les quelques millisecondes) pour en-

registrer la trace courante de la pile, le profilage exact observe tous les événements réalisés. Ce type de profilage, comme son nom l'indique, produit des profils exacts et non des approximations statistiques.

Bien que la plupart des profileurs utilisent la technique d'instrumentation, le profilage à base d'événement peut aussi livrer des profils exacts. À titre d'exemple, JVMTI [6] est capable d'envoyer des notifications aux entrées et sorties des méthodes. L'outil permet alors de faire du traitement à l'entrée de toutes ou de certaines méthodes choisies. On peut notamment enregistrer les signatures des méthodes et obtenir à la fin de l'exécution la liste de toutes les méthodes invoquées.

Les profileurs d'instrumentation instrumentent l'application à profiler pour collecter l'information nécessaire. Instrumenter signifie injecter directement des instructions dans le code. Instrumenter un programme cause toujours un ralentissement de son exécution. Néanmoins, le code injecté ainsi que son emplacement sont contrôlables de façon à résulter en un impact minimal sur l'exécution. Nous discutons plus en détails de l'instrumentation dans la section 2.4.

2.3 Représentation des données

Le style orienté objet favorise la décomposition du traitement en plusieurs petites méthodes reliées par des appels virtuels fréquents. La haute fréquence d'appels de méthodes dans les applications orientées objet rend difficile l'intervention sur chaque appel de méthode pour accumuler les données de profilage. Le défi est de construire efficacement une structure assez large permettant des optimisations précieuses sans trop perturber l'exécution.

Les graphes d'appels sont une représentation succincte couramment utilisée pour schématiser les invocations de méthodes. Un graphe d'appels est un graphe $G = (N, A)$ où N est l'ensemble des noeuds et A est l'ensemble des arcs. Chaque noeud de N correspond à une et une seule méthode. Chaque arc est un triplet (N_1, CS, N_2) représentant un appel du site d'appel CS de la méthode N_1 vers la méthode N_2. Dans le cas des appels virtuels, le même site d'appel peut invoquer plusieurs méthodes distinctes. Un graphe

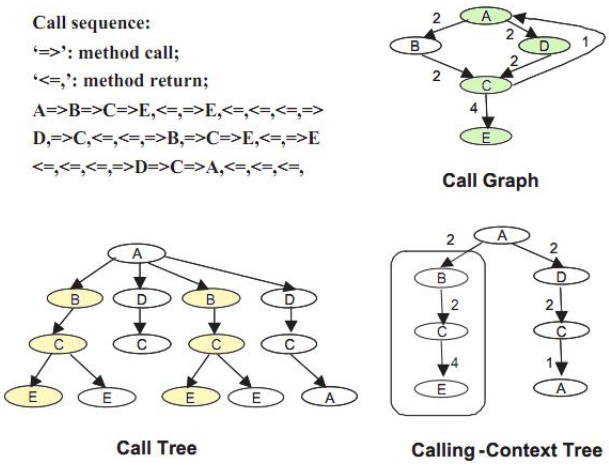

Figure 2.1 – Exemple d'un graphe d'appels (*Call Graph*), d'un arbre d'appel (*Call Tree*) et d'un arbre de contextes d'appels (*Calling Context Tree*)

d'appels dynamique (DCG) est un graphe d'appels contenant seulement les appels ayant concrètement eu lieu à l'exécution. Les arcs du graphe d'appels dynamique sont agrémentés par de l'information additionnelle telle que les fréquences d'appels.

Cependant le DCG manque de précision, il peut par exemple contenir des chemins irréalisables correspondant à des séquences d'appels qui n'ont pas eu lieu à l'exécution. La figure 2.1 représente le graphe d'appels à partir d'une séquence des entrées et sorties de méthodes. Le chemin A-D-C-E du graphe d'appels n'existe pas sur la séquence d'appels.

En revanche, pour représenter les appels de méthodes à l'exécution, un arbre d'appels dynamique (DCT) est la structure de données la plus précise mais la plus inefficace en espace mémoire. La taille d'un DCT est proportionnelle au nombre total des appels à l'exécution puisque chaque appel de méthode est représenté par un nouvel arc et doit créer un nouveau noeud.

L'arbre de contextes d'appels (*Calling Context Tree*) permet une représentation compacte de tous les contextes d'appels présents sur l'arbre d'appels. Un contexte d'appels est définit par la chaîne de méthodes présentes simultanément sur la pile d'exécution. Un CCT élimine les redondances de l'arbre d'appels mais préserve les contextes d'appels. Sur la figure 2.1, bien que le DCG a moins de noeuds que le CCT, ce dernier distingue

entre les chemins A-B-C et A-D-C offrant ainsi plus de précision.

L'importance de l'information sur les contextes d'appels à été démontré dans plusieurs travaux existants [3, 9, 11, 29, 45, 68, 71, 80]. Les contextes d'appels révèlent l'état complet de la pile pendant l'exécution. Ils permettent de retrouver en parfaite précision, les méthodes actives à tout moment de l'exécution, leurs méthodes appelantes et aussi leurs temps d'exécution. Toutefois, enregistrer et générer tant d'informations dégrade vivement le temps d'exécution du programme profilé.

Ammons et al. [3] introduisent l'idée de construire un arbre de contextes d'appels (CCT) pour enregistrer dynamiquement l'information des contextes d'appels. L'article aspire que la taille de telles structures de données peut être raisonnable même en profilant des programmes de taille large.

La façon exhaustive pour construire un arbre de contextes d'appels utilise une trace d'exécution contenant les entrées et sorties de toutes les méthodes. À l'aide d'un pointeur marquant initialement la méthode racine, un noeud de méthode est ajouté si nécessaire ou modifié pour rectifier les métriques enregistrées. Le pointeur est déplacé en avant en cas d'appel d'une nouvelle méthode et en arrière en cas d'un retour de méthode. En cas de threads multiples, un pointeur par thread est requis. Zhuang et al [82] ont testé cette façon naïve de construction d'arbres de contextes d'appels complets. Leur implémentation utilise une infrastructure à base de JVMPI qui instrumente les entrées et les sorties de toutes les méthodes. Leurs résultats d'expérimentation montrent que le coût du traçage peut causer jusqu'à 50 fois de ralentissement à l'exécution.

Différentes techniques ont été proposées pour la génération des CCT [3, 9, 23, 68, 78, 82]. La construction d'un CCT complet s'avère très coûteuse en temps d'exécution. Des techniques plus économiques ont été proposées pour construire des structures de données analogues au CCT sans conserver la complétude. Elles sont utilisées dans des cas où les contraintes de temps et d'espace ne permettent pas la construction d'un CCT complet. Tel est le cas pour la sélection des candidats d'optimisation par les compilateurs dynamiques. Whaley [78] présente une structure nommée PCCT (Partial Calling Context Tree), un arbre de contextes d'appels partiel vu qu'il ne représente pas la totalité des contextes. Le parcours de la pile d'exécution s'arrête à un chiffre k de méthodes exami-

nées lors de la collecte des contextes. La technique adoptée construit une structure d'une précision de 90% comparée à un CCT complet pour k=16. ACCT (Approximate Calling Context Tree) est une autre structure semblable au CCT utilisée par Arnold et Sweeney [9]. Leur technique parcours la pile d'exécution depuis la méthode courante jusqu'à la méthode racine pour déterminer les contextes d'appels. Ils utilisent l'échantillonnage et peuvent aboutir à un impact minimal sur l'exécution en réduisant la fréquence de la prise d'échantillons.

Le présent mémoire s'intéresse à la génération de graphes d'appels dynamiques complets pour les applications Java. Il ne fait pas de doute que les arbres de contextes d'appels sont plus intéressants et plus efficaces pour le profilage, ainsi ce travail consiste en la première phase d'un projet visant éventuellement à produire dynamiquement des CCTs complets. Nous nous sommes intéressés à bâtir une infrastructure pour profiler la totalité des méthodes exécutées en réduisant au minimum son impact sur l'exécution. Les graphes d'appels dynamiques générés diffèrent des graphes classiques par les renseignements supplémentaires qu'ils fournissent comme l'information des allocations. Les graphes générés représentent les relations entre les méthodes exécutées sous la forme appelante-appelée en distinguant entre les sites d'appels.

2.4 Instrumentation du code

L'instrumentation est l'usage ou l'application d'instruments pour fins d'observation, de mesure ou de contrôle. Instrumenter du code revient à modifier le code par ajout ou suppression de bouts de code pour des raisons d'observation de comportement du programme ou de mesure de propriétés pendant l'exécution, ou de prise de contrôle de l'exécution.

L'instrumentation peut se faire au niveau du code source ou au niveau du code binaire. Elle peut aussi avoir lieu au moment de la compilation ou de l'édition des liens. La suite de la section présente chacune des deux premières techniques et étale quelques travaux existants reliés.

2.4.1 Instrumentation du code source

Une transformation[1] *source-to-source* [65] ajoute du code d'instrumentation directement au code source du programme. De telles transformations sont souvent utilisées pour mesurer certaines caractéristiques du code [52, 53], profiler les programmes [61], analyser le temps d'exécution [58] ou l'usage de mémoire [74]. L'instrumentation du code source permet d'éviter d'enregistrer une trace détaillée des événements lors de l'exécution du programme.

Pereira et al. [65] proposent une méthode d'*address tracing* par instrumentation du code source des programmes en langage C. Ils citent que collecter les traces d'adresses se fait soit par injection de code d'instrumentation au niveau de l'exécution, soit par application de transformations directement sur le code source en C. La première selon les auteurs, bien qu'efficace, dépend cependant de l'architecture matérielle. Par souci de portabilité, ils adoptent la deuxième méthode lors du développement de leur outil qui a pour but de faciliter l'optimisation de mémoire du système auquel ils s'intéressent. Pour instrumenter du code source, ils utilisent une bibliothèque orientée objet, Sage++ [22] adaptée aux transformations en C, pour construire leur système de transformations. En une première phase, l'arbre de syntaxe abstraite(AST) est transformé en un AST modifié équivalent qui permet l'ajout de l'instrumentation. Une passe du nouveau AST est effectuée à la suite pour insérer du code permettant la collecte des données. En dernier, vient l'étape de reconstitution, appelé aussi *unparsing*, qui traduit l'arbre modifié en du code en C. Les fichiers résultants contiennent alors du code source instrumenté.

Filman et al. [39] font usage de l'instrumentation de code source Java en programmation orientée aspects. Selon les auteurs, le code source est plus lisible que le code binaire et permet des transformations compréhensibles pour les novices sans avoir à comprendre la JVM et les actions du compilateur. Même s'ils affirment la précision des données révélées au niveau du langage machine, ils trouvent la complexité de transformer du code binaire trop élevée et optent donc pour des transformations au niveau du code source. L'objectif des auteurs est de développer un système où le comportement peut être dû

[1] On entend par transformation toute modification du code.

à n'importe quel événement au cours de l'exécution du programme. Ils voudraient pouvoir exécuter un comportement spécifique lorsqu'un certain événement se produit, et ceci sans avoir à marquer au préalable les endroits où le comportement désiré doit avoir lieu. L'architecture envisagée pour leur nouveau système est illustrée dans la figure 2.2. Ils prévoient un mécanisme où une description d'un ensemble de paires événement-action sera présenté au compilateur. Chaque paire comprendra une phrase décrivant l'événement intéressant dans le langage des événements et une action qui sera un programme à exécuter lorsque l'événement se produit. L'article décrit l'implémentation planifiée et cite quelques applications possibles de leur système.

Même si l'instrumentation du code source demande moins d'effort et de connaissances techniques que l'instrumentation du code binaire, elle n'est cependant pas toujours applicable. L'instrumentation du code source ne peut être employée en absence des fichiers sources. Or les programmes protègent souvent les fichiers sources ou ne livrent que des fichiers compilés. De plus, si le code source est disponible, il ne représente que les classes de l'application elle-même. Les classes de librairies partagées ou de packages provenant d'autres projets qu'utilisent l'application sont absentes. L'instrumentation du code source ne permet qu'une couverture partielle des classes.

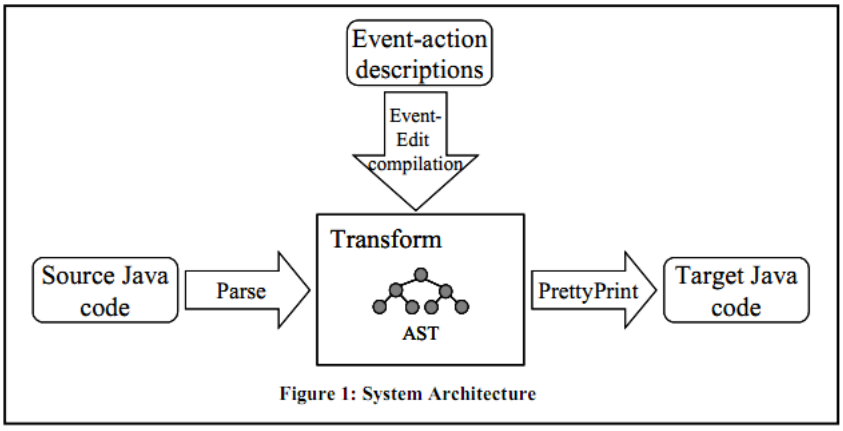

Figure 2.2 – Architecture du système de Filman et al. [39]

2.4.2 Instrumentation du code binaire

Les classes Java compilées sont stockées dans des fichiers de classes dont le format est disponible dans la spécification de la machine virtuelle Java [57]. Un fichier de classe contient du bytecode, à savoir, des instructions qui sont interprétées par la machine virtuelle Java(JVM). Chaque instruction se compose d'un opcode d'un octet suivi de zéro ou plusieurs opérandes. Ces instructions manipulent le contenu de la pile d'exécution. Dans tout ce qui suit, il s'agit d'instrumentation du code binaire (bytecode) Java sauf si indiqué autrement.

2.4.2.1 Présentation

L'instrumentation du bytecode [1] est le processus d'ajout de nouvelles propriétés à un programme en modifiant le bytecode d'un ensemble de classes avant leur chargement par la machine virtuelle. Le profilage par instrumentation[2] pour fins de monitoring ne s'intéresse pas à l'ajout de nouvelles fonctionnalités en soi, mais à l'amélioration d'un programme par usage de la trace de son exécution afin de recueillir des données de profilage, de surveiller l'utilisation de mémoire, etc. Par exemple, un outil de profilage voudrait probablement instrumenter les constructeurs des différentes classes pour garder trace de la création des objets dans une application. De même, la fonctionnalité de point d'arrêt d'un débogueur peut être mise en œuvre par insertion de bytecode personnalisé dans les classes à inspecter.

2.4.2.2 Types d'instrumentation

L'instrumentation peut être statique ou dynamique. Elle est statique lorsque les fichiers de classes sont modifiés d'avance pour produire une nouvelle version instrumentée. Cette technique sépare l'étape d'instrumentation de l'exécution, elle effectue l'instrumentation avant l'exécution et réduit ainsi le ralentissement à l'exécution par soustraction du coût de l'instrumentation. Cependant l'instrumentation statique doit enregistrer les fichiers instrumentés sur le disque et nécessite parfois un nettoyage manuel et le

[2]Pour tout le reste du document, le terme instrumentation désigne l'instrumentation du bytecode.

lancement d'une nouvelle instrumentation en cas de toute modification des classes.

En contrepartie, l'instrumentation dynamique modifie les fichiers de classes au fur et à mesure de leur utilisation. Avec ce procédé, les fichiers sont modifiés en mémoire et utilisés par la suite. Aucun usage de mémoire sur disque n'est requis ainsi il n'y a aucun nettoyage à faire. Le problème de ré-instrumentation ne se pose pas non plus, puisque même en cas de modification, les classes sont toujours instrumentées juste avant leur utilisation.

Malgré le ralentissement qu'elle cause lors du processus de chargement, l'instrumentation dynamique procure une information complète, chose que l'instrumentation statique ne peut atteindre. Ce n'est qu'au moment de l'exécution que le comportement complet des programmes est observé. Les applications modernes font usage de bibliothèques partagées, de fonctions virtuelles, de code généré dynamiquement et bien d'autres mécanismes dynamiques. Elles ne sont assemblées et définies qu'à l'exécution. L'information disponible statiquement est souvent insuffisante. L'inconvénient majeur de l'instrumentation statique est qu'elle n'instrumente pas les classes générées dynamiquement, alors que l'instrumentation dynamique ne laisse échapper aucune classe chargée par l'application incluant le code généré dynamiquement. De plus, l'instrumentation statique est appliquée systématiquement sur toutes les classes dont des classes de librairies qui ne seront pas utilisées tandis que seule les classes qui ont été véritablement chargées sont altérées par l'instrumentation dynamique. Cette propriété de l'instrumentation dynamique est exploitable pour améliorer la performance des systèmes de profilage.

2.4.2.3 Domaines d'application

L'instrumentation du bytecode est une technique bien connue applicable dans plusieurs domaines. Elle peut être utilisée pour modifier une ou plusieurs fonctionnalités d'un programme aussi bien qu'elle peut surveiller l'exécution du programme sans modifier son comportement. Nous discutons dans cette section de quelques outils d'instrumentation utilisés dans divers domaines.

Le monitoring est le domaine d'application de l'instrumentation du bytecode par excellence. JProfiler [51] est un exemple de profileur commercial pour Java qui utilise

l'instrumentation du code. En plus du profilage par échantillonnage, JProfiler offre l'option de profiler par instrumentation soit en mode instrumentation complète qui traque tous les appels de méthodes, soit en mode instrumentation dynamique qui signifie pour lui un mode plus léger qui cache le fonctionnement interne de certaines API externes utile pour des applications en J2EE par exemple. JFluid [7] est un profileur intégré à l'éditeur NetBeans [8] qui utilise également l'instrumentation dynamique du bytecode. Il peut profiler l'application entière ou être appliqué à une portion de code en le mettant en marche une fois la partie souhaitée est atteinte et en le désactivant par la suite pour laisser le programme rouler à sa vitesse normale. Cependant JFluid requiert une version modifiée de la machine virtuelle Java [36].

Plusieurs outils font usage de l'instrumentation pour collecter de l'information sur l'exécution et construire des graphes d'appels ou des arbres de contextes d'appels. JFluid [36] cité ci dessus permet de générer un sous-graphe du graphe d'appels dynamique pour une section du programme pour laquelle le profilage a été activé. FERRARI [18, 19] est aussi un des frameworks d'instrumentation de bytecode pour générer des CCT. L'outil joint l'instrumentation statique à l'instrumentation dynamique, il instrumente statiquement les *core classes* et applique l'instrumentation dynamique pour le reste des classes. JP [17] est un autre outil qui emploie également l'instrumentation statique et dynamique. Il a été amélioré dans un travail plus récent pour assurer sa portabilité à travers les plateformes [20].

Par ailleurs, des outils de *code coverage* utilisent l'instrumentation du code. Lors de réalisation de tests unitaires par exemple, de tels outils sont appliqués pour juger de la qualité des tests. Plus les instructions exécutées par le test sont similaires à celles déroulées lors de l'exécution de l'application de base, mieux est la qualité du test. Cobertura [31] est un outil gratuitement disponible qui calcule le pourcentage du code accessible par un test et identifie les parties du programme non couvertes par le test. Il utilise l'instrumentation du bytecode avec ASM pour connaître les lignes de code visitées lors de l'exécution du test. CodeCover [32], Emma [38] et Clover [30] sont aussi des outils permettant d'effectuer des tests de couverture.

L'instrumentation du code est également utilisée pour illustrer le flux de contrôle de

n'importe quelle application Java. L'usage de la technique d'instrumentation du code assure une flexibilité remarquable pour de tels outils. Certains outils l'emploient pour générer dynamiquement des diagrammes de séquences illustrant le flux de contrôle réel pendant l'exécution [2].

L'instrumentation est aussi bien présente dans la programmation orientée aspects (POA). Un grand souci de la POA est de séparer les préoccupations transverses (*cross-cutting concerns*) [1]. Les préoccupations transverses sont difficiles à encapsuler séparément dans les applications modernes orientées objet. Un exemple fréquent est l'accès à une base de données. Le code de connexion à la base de données est éparpillé et répété dans chaque méthode interagissant avec la base. En utilisant un outil à base d'instrumentation pour la POA comme AspectWerkz [14] , les méthodes qui manient la base de données peuvent être interceptées à l'exécution et modifiées de sorte à ajouter le traitement de connexion à la base d'une façon transparente. Par conséquent, le code source devient moins répétitif, plus élégant et plus simple à maintenir.

2.5 Travaux connexes

Il existe deux façons pour collecter l'information du profilage, l'échantillonnage et l'instrumentation. Le profilage par échantillonnage est plus léger, il influence moins la performance du programme profilé et ne modifie pas son code. Cependant, ce profilage manque de précision puisqu'il produit des estimations statistiques. Il n'y a aucune garantie que le profilage du même programme puisse toujours produire le même profile. En contrepartie, l'instrumentation aboutit généralement au même profil et se manifeste par la haute précision de ses profiles. Cependant, profiler par instrumentation induit un ralentissement indéniable au programme qui croît avec le nombre de threads et des méthodes en cours d'exécution. L'instrumentation modifie le code de l'application, elle change les chemins d'exécution et empêche par conséquent l'application d'optimisations susceptibles d'améliorer la performance. En général, l'échantillonnage est utilisé là où la performance passe en premier et l'instrumentation lorsque le souci de précision domine.

Cependant, des approches d'instrumentation à base d'échantillonnage temporel per-

mettent de réduire considérablement le temps additionnel de l'instrumentation. Par exemple, Arnold et Sweeney [9] échantillonnent périodiquement la pile d'exécution pour construire un arbre de contextes d'appels approximatif (ACCT). Whaley utilise un échantillonnage similaire pour bâtir un arbre de contextes d'appels partiel (PCCT) [78]. Construire un ACCT ou un PCCT altère moins la performance puisque le travail ne se fait qu'à la prise d'un échantillon plutôt qu'à chaque entrée et sortie de méthode lors de la construction d'un CCT. Arnold et Ryder [13] présentent un système pour la machine virtuelle Jalapeno qui crée deux versions de chaque méthode. L'exécution échange entre une version contenant le code original de la méthode et une deuxième version instrumentée sur déclenchement d'un compteur temporel. Leur implémentation cause une dégradation minimale en performance de 3% en moyenne.

Construire des profils de haute précision à un coût minimal est particulièrement important pour la compilation à la volée (*just-in-time compilation*) puisque tout ralentissement dû au profilage doit être compensé en performance. Un compilateur dynamique doit être très spécifique en choisissant les portions à optimiser pour ne sélectionner que celles qui ont de grandes chances à améliorer la performance. La majorité des machines virtuelles utilisent le profilage par échantillonnage pour trouver les candidats à optimiser [4, 10, 27, 44, 46, 70, 72, 78, 81]. Whaley [78] présente un système de profilage capable de fournir de l'information précise à petit coût pour guider le compilateur du JIT en temps réel. La technique proposée utilise l'échantillonnage pour construire la structure de données PCCT à 2-4% de temps additionnel à l'exécution. Arnold et Grove [11] proposent une technique à base d'échantillonnage timer-based et counter-based pour construire un graphe d'appels dynamique pour les optimisations par rétroaction. La particularité est qu'au déclenchement de l'échantillonnage, au lieu de prendre un seul échantillon, N échantillons sont enregistrés. L'évaluation de leur implémentation sur les machines virtuelles Jikes RVM et J9 par comparaison au résultat d'un profil parfait, montrent une hausse de précision pour les benchmarks à exécution longue. Ils rapportent une moyenne de précision de 69% pour Jikes RVM [67] et 74% pour J9 [42], contre 50% et 46% respectivement pour le système de base. Buytaert et al. [27] montrent que l'échantillonnage à base des HPM pour guider les compilateurs dynamiques est plus

bénéfique que l'échantillonnage temporel. Les auteurs affirment que l'échantillonnage à base des HPM est plus précis, moins coûteux et améliore la performance par 5.7% en moyenne. Leur évaluation avec Jikes RVM rapporte une évolution en performance de 18.3% par rapport au système de base, sans modification du compilateur.

Il existe plusieurs techniques de génération d'arbres de contextes d'appels ou de graphes d'appels ayant pour but d'améliorer la précision des profils par échantillonnage. Arnold et Sweeney [9] utilisent l'échantillonnage et parcourent la pile d'exécution pour enregistrer les contextes d'appels. Les méthodes vues pour la première fois sont ajoutées à l'arbre et le poids des méthodes existantes est incrémenté. Cependant en plus des inconvénients de l'échantillonnage, cette technique peut produire des résultats incorrects dans le cas où le programme passe la majorité de son temps d'exécution dans une seule méthode, l'approche supposera l'existence d'appels fréquents à cette méthode. Pour améliorer la précision des CCTs, Zhuang et al. [82] proposent de collecter au moment de l'échantillonnage une rafale (*burst*), une séquence d'entrées et sorties de méthodes, avec un parcours de la pile juste avant pour avoir l'information du contexte actuel. Cependant les auteurs trouvent cette technique inefficace en temps d'exécution et collecte beaucoup d'informations redondantes. Ils améliorent donc la technique en désactivant la collecte de rafales pour les contextes déjà vus et la réactive selon l'historique d'exécution et un indice de réactivation. Avec cette nouvelle approche, ils peuvent atteindre près de 90% de précision avec 20% de temps additionnel à l'exécution.

Néanmoins, les auteurs démontrent dans [63] que les résultats du profilage par échantillonnage sont biaisés pour les applications Java. Ils étudient le désaccord de quatre profileurs célèbres à identifier les portions à problèmes dans le code. Les profileurs pris en compte sont hprof, xprof, jprofiler et yourkit. Les auteurs expliquent la divergence par l'utilisation des *yield points* pour déclencher l'échantillonnage, méthode adoptée dans la plupart des profileurs pour Java. Or ces points ne sont pas indépendants puisque les compilateurs optimisent souvent leurs emplacements. La notion de l'aléatoire lors de la collecte des échantillons n'est pas totalement respectée en Java. Tandis que pour les profileurs des programmes en C comme gprof, le problème ne se pose pas puisque le déclenchement de l'échantillonnage est contrôlé par un compteur du système d'exploi-

tation. L'article implémente une stratégie pour effectuer aléatoirement l'échantillonnage pour les applications en Java.

Malgré que l'échantillonnage peut être utile pour identifier les contextes les plus chauds, il n'est pas assez pertinent pour servir en débogage ou en évaluation par exemple où la couverture de la totalité des contextes est cruciale. Par définition, l'échantillonnage ne peut pas générer la totalité de l'information sur l'exécution tandis que l'instrumentation peut couvrir toutes les instructions exécutées. L'instrumentation est capable de fournir précisément le nombre des événements comme les invocations de méthodes et le temps d'exécution exactes. Elle peut être appliquée pour la totalité du programme pour une complétude parfaite ou sélectivement pour des portions du code, comme pour certaines méthodes lorsque appelées avec des valeurs spécifiques en paramètres.

Plusieurs outils font usage de l'instrumentation pour collecter de l'information sur l'exécution et construire des graphes d'appels ou des arbres de contextes d'appels. Dmitriev [36] présente une technique d'instrumentation dynamique du bytecode pour la machine virtuelle Java HotSpot. L'approche s'intéresse à instrumenter pendant l'exécution une portion du code pour réduire la dégradation du profilage. L'article présente l'outil JFluid qui génère un sous-graphe d'appels dynamique. Cependant l'outil ne procure qu'une vue partielle du comportement lorsqu'une instrumentation partielle est effectuée, or lors de l'instrumentation totale du programme en entier un ralentissement de jusqu'à 50 fois est enregistré. FERRARI [18, 19] et JP [17, 20] emploient l'instrumentation du bytecode pour générer des arbres de contextes d'appels. Les deux outils nécessitent une étape préexécution pour instrumenter statiquement les classes de librairies Java et instrumentent dynamiquement le reste des classes lors de l'exécution. Les auteurs rapportent une moyenne du facteur de ralentissement de 3.93 et 4.23 pour FERRARI et JP respectivement. JP souffre cependant de certaines limites en précision. Il ne fournit pas la totalité des contextes puisque l'outil impose une profondeur limitée en cas de récursion. Les méthodes appelées par du code natif sont représentées comme des fils du noeud racine. Aussi, en cas d'exceptions, des imprécisions existent dans les informations rapportées. Dans [62], les auteurs ont modifié FERRARI pour aboutir à un CCT plus complet. Ils définissent la complétude par toutes les invocations (i) de méthode Java par une mé-

thode Java, (ii) de méthode Java par du code natif et (iii) du code natif par méthode Java. Reste que la collecte de l'information des contextes n'est pas supportée pendant la phase du bootstrap. L'outil continue d'instrumenter statiquement les classes des librairies Java qu'il utilise par la suite pendant l'instrumentation dynamique des classes.

Rothlisberger et al. [66] présentent *Senseo*, un plugin pour l'éditeur Eclipse qui agrémente le code source et les vues d'Eclipse par des métriques dynamiques. Pour aider les developpeurs à comprendre les systèmes à hiérarchie complexe employant des classes abstraites et interfaces, et à identifier les problèmes de performance, les auteurs trouvent plus convenable de fournir sur l'éditeur des informations telles que les méthodes qui sont souvent exécutées concrètement, leurs fréquences d'appel, les méthodes invocatrices. le nombre d'objets et l'espace mémoire alloués pour certaines méthodes. Les données affichées sont l'agrégation des métriques dynamiques collectionnées lors de plusieurs exécutions. *Senseo* utilise *MAJOR* [76, 77], un outil permettant de faire le tissage d'aspects dans toutes les classes chargées par la machine virtuelle Java et qui utilise la technique d'instrumentation du bytecode de FERRARI. Pour représenter les différentes métriques, les auteurs utilisent un CCT étendu capable d'enregistrer toute l'information. La construction du CCT et la collecte des métriques dynamiques sont implémentées comme aspect, chose que les auteurs jugent en mesure de rendre l'implémentation compacte et la maintenance et extensibilité plus faciles.

Loin des applications Java, plusieurs outils ont été conçus à base d'instrumentation statique, dynamique ou les deux. ATOM [69] et EEL [54] sont deux des premières implémentations des outils pour analyse des programmes par instrumentation statique du code binaire. ATOM est disponible pour la plateforme Alpha seulement cependant ce processeur n'est plus en production. EEL fournit une interface portable pour édition des exécutables en cachant l'architecture sous-jacente. PEBIL [55] est un outil plus récent pour instrumentation statique des programmes sur Linux pour la famille x86/x86_64. Contrairement à EEL, les auteurs de PEBIL exposent des détails d'implémentation pour les développeurs de tirer avantage de leur connaissance de la plateforme pour améliorer la performance de l'application instrumentée. Dyninst [25] est un outil qui peut instrumenter statiquement ou dynamiquement. Il peut soit modifier les exécutables sur le

disque ou instrumenter à l'exécution. Pin [59], DynamoRIO [24] et Valgrind [64] sont des outils d'instrumentation dynamique du code binaire qui opèrent à la façon du JIT. L'application à instrumenter s'exécute par dessus le profileur qui intervient pour instrumenter.

Pour conclure, échantillonnage ou instrumentation, chacune des techniques a ses avantages et ses inconvénients. Pour remédier au dilemme, certains profileurs, souvent commerciaux [51], implémentent les deux et laissent ainsi la liberté à l'utilisateur de choisir selon son besoin laquelle appliquer.

2.5.1 Synthèse

L'échantillonnage et l'instrumentation sont deux techniques différentes permettant la collecte d'informations pour le profilage. Le tableau 2.I rappelle certains des travaux existants que nous avons abordés dans ce chapitre. Idéalement, un profileur doit être en mesure de fournir l'information complète sans ralentir considérablement l'exécution du programme profilé. Or d'une part, pour assurer la complétude de l'information collectée, il faut opter pour l'instrumentation dynamique. Toutefois celle-ci engendre le plus de ralentissement, comme le montre le tableau 2.I. D'autre part, pour un profilage plus rapide, l'échantillonnage est la solution à moindre coût. Cependant, l'échantillonnage ne peut fournir que des approximations statistiques n'atteignant pas la précision de l'instrumentation dynamique.

Travail	Échantillonnage	Instrumentation	Précision	Ralentissement
Whaley [78]	X	-	90%	2-4%
Arnold et Ryder [13]	X	dynamique	93-98%	3-6%
Arnold et Grove [11]	X	-	Jikes RVM 69% J9 74%	0% 0.4%
Zhuang et al. [82]	X	-	90%	20%
Buytaert et al. [27]	X	-	78%	1%
Dimitriev [36]	-	dynamique	CCT complet CCT partiel	50 fois 22 fois
Binder et Hulas [17]	-	statique & dynamique	CCT complet	4.23 fois
Binder et al. [18]	-	statique & dynamique	CCT complet	3.93 fois

Tableau 2.I – Tableau comparatif de quelques travaux existants.

JP [17] de Binder et Hulas et FERRARI [18] de Binder et al. sont deux outils pour générer dynamiquement des arbres de contextes d'appels au coût d'un ralentissement raisonnable. Ces travaux génèrent des arbres de contextes d'appels complets mais souffrent de certaines imprécisions dans l'information rapportée et imposent une étape préparatoire d'instrumentation statique de certaines classes.

Au meilleur de nos connaissances, il n'existe pas d'outil capable d'instrumenter dynamiquement toutes les classes pendant l'exécution d'un programme et de fournir une couverture complète et une précision exacte en gardant un ralentissement acceptable lors du profilage. Ce projet vient donc remplir ce vide et répondre aux exigences de précision et de performance des applications de taille industrielle.

CHAPITRE 3

LE PROFILEUR DYKO

Les applications modernes font usage de librairies partagées, d'appels virtuels, de plugins, de code généré dynamiquement et de plusieurs autres mécanismes dynamiques. Elles ne sont assemblées qu'au moment de l'exécution. Peu d'information concernant le comportement est disponible statiquement. Les outils statiques utilisent parfois les profils enregistrés préalablement mais ne peuvent fournir ainsi qu'une estimation du vrai comportement. Par contre, au-delà de procurer l'image complète du comportement disponible uniquement lors de l'exécution, les outils dynamiques se concentrent sur le code concrètement exécuté au lieu de gaspiller des ressources sur le code non exécuté. Les outils dynamiques traitent toutes les méthodes des différents modules uniformément, ils ne nécessitent pas le code source de l'application profilée et n'exigent pas non plus sa recompilation.

Disposer des outils puissants est essentiel pour l'application de tâches d'analyses comme le profilage, l'évaluation de performance et la détection des bogues. ***Dyko*** est notre outil de génération de graphes d'appels **dy**namiques **co**mplets. Ce chapitre présente l'outil, son fonctionnement, ses caractéristiques et son implémentation.

3.1 Présentation

3.1.1 Approche

Dyko est notre profileur capable de générer des graphes d'appels dynamiques et complets. L'outil utilise l'instrumentation dynamique pour profiler toutes les classes de l'application. Le code d'instrumentation permet de collecter et d'enregistrer les informations rapportées sur les graphes. La figure 3.1 illustre l'organisation de dyko, ses différents composants et le cheminement des classes suite à son interaction avec la machine virtuelle de l'application.

L'application à profiler s'exécute normalement sur la machine virtuelle, avec son

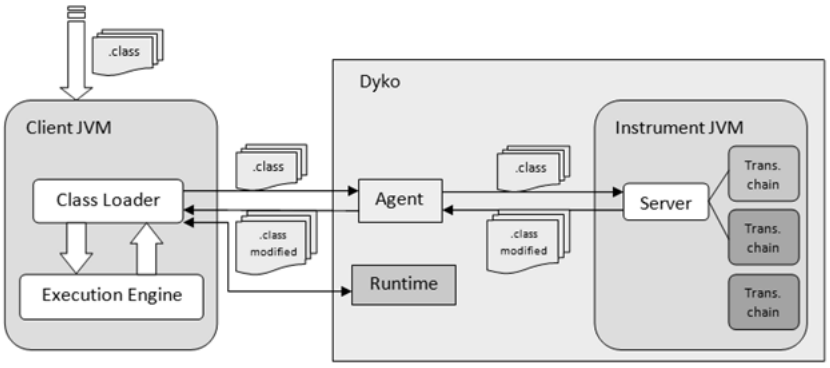

Figure 3.1 – Organisation de *dyko*

moteur d'exécution qui demande le chargement des classes au *chargeur de classes*. Sur la figure 3.1, les fichiers de classes compilées sont représentés par les *.class*.

Dyko intervient au niveau du chargement des classes. Pour ne pas compromettre l'exécution, *dyko* utilise une deuxième machine virtuelle indépendante pour instrumenter le code. *Dyko* peut utiliser toute machine virtuelle standard compatible avec JVMTI[6]. L'agent de dyko prend possession des classes avant leur chargement et les achemine vers la machine virtuelle de dyko. Les classes sont transmises par une connexion réseau établie entre l'agent de dyko et le serveur d'instrumentation. Le serveur traite les classes parvenues une à une. Chaque classe subit alors une ou plusieurs chaînes de transformation sélectionnées. Le code d'instrumentation correspondant est injecté dans le bytecode de la classe. La classe modifiée est par la suite rendue à l'agent qui la transfère au chargeur de classes de la machine virtuelle de l'application.

Vu du côté de l'application, le moteur d'exécution a demandé le chargement d'une classe qui lui a été fournie sauf que son bytecode a été modifié. Quoique les données de sortie et le résultat final de l'application restent tout de même inchangés.

Le moteur d'exécution de la machine virtuelle cliente procède par la suite à l'exécution du code de la classe instrumentée. Parmi le code inséré, figurent des appels de méthodes spécialement implémentées dans des classes du *runtime* de dyko pour la collecte d'information et la construction des graphes. Le chargeur de classes s'occupe de charger les classes du runtime de dyko nécessaires à l'exécution du code d'instrumentation de la même façon qu'il procède pour les classes de l'application.

3.1.2 Caractéristiques

Dyko se distingue par les caractéristiques principales suivantes :

Complétude : Dyko assure une couverture complète du code exécuté. Il traite toutes les classes chargées uniformément. Bien que les modifications apportées au code puissent changer d'une classe à une autre, elles sont toutes acheminées de la même façon et subissent la même chaîne de transformations. L'outil instrumente dynamiquement les classes du bootstrap, les classes des librairies standards Java et les classes de l'application. Aucune classe ne lui échappe quel que soit son origine.

Précision : Les données collectées rapportent précisément les appels de toutes les méthodes lors de l'exécution. Le profilage suit les méthodes *native* et inclus sur les graphes les méthodes Java invoquées par le code *native*. La technique utilisée veille de plus à garder la cohérence en cas d'exceptions ou de fin anormale de l'exécution.

Fléxibilité : L'instrumentation du bytecode permet un contrôle total du code. Changer les signatures des méthodes, ajouter de nouvelles variables ou injecter du code sont toutes des opérations possibles à ce niveau. Manipuler le code à un niveau plus haut comme au niveau du code source, ne permettrait pas de manipulations telles que l'ajout des paramètres aux méthodes par exemple. De plus, *dyko* utilise le framework ASM pour générer le bytecode. Il bénéficie par conséquent de la simplicité de composer ou décomposer les transformations à appliquer, nous allons expliquer ceci plus en détail dans ce qui suit.

Extensibilité : Grâce à la modalité qu'offre ASM, l'ajout d'une nouvelle caractéristique peut se faire indépendamment sans compromettre celles existantes. Un utilisateur pourra tester sa nouvelle implémentation et par la suite décider de l'intégrer ou non au reste.

Simplicité : L'outil est simple à utiliser comme nous l'avons expliqué auparavant. Il ne requiert aucune étape préparatoire d'instrumentation statique comme utilisé dans

FERRARI[19]. Ainsi, modifier des classes de l'application ne nécessite pas la recompilation ni la réinstrumentation des classes.

Performance : Le profilage dynamique cause toujours un ralentissement à l'exécution des programmes par rapport à leurs temps initiaux. *dyko* a été optimisé sur plusieurs étapes pour réduire le plus que possible le temps additionnel nécessaire à la collecte de l'information requise.

Transparence : L'application ne se rend pas compte qu'elle se fait profiler. Il n'y a pas de thread spécialement pour le profilage qui roule en parallèle avec le programme. Une fois l'exécution terminée, l'application est dans le même état que le programme initial. Son code source sur le disque n'est pas modifié en aucune façon et elle est prête pour une nouvelle exécution ou une nouvelle session de profilage. Dyko peut cependant enregistrer sur demande les classes instrumentées sur le système de fichiers, pour des inspections postérieures par exemple.

3.1.3 Utilisation

Profiler avec *dyko* est simple. Aucune étape préparatoire n'est requise.

L'exécution normale d'une application Java est lancée comme suit.

```
java -jar Application.jar
```

Pour instrumenter cette exécution avec *dyko*, il suffit de rajouter certains paramètres. L'option -agentpath permet de charger l'agent de dyko. L'option -Xbootclasspath permet de rendre les classes de dyko *runtime* disponibles au chargement.

```
java -agentpath:Path/dyko.profile.agent/Debug/libdyko.so -Xbootclasspath/a:Path/dyko/bin/
 -jar Application.jar
```

3.2 Instrumentation du bytecode

Cette section discute de la technique d'instrumentation du bytecode, ses défis et les outils qu'elle utilise. La section présente ensuite un exemple d'instrumentation du bytecode en faisant usage du framework de manipulation de bytecode ASM, comme est le cas pour l'instrumentation des classes dans *dyko*.

Pour des fins de débogage, les classes Java peuvent être inspectées par des outils capables de décompiler du code Java. *Javap*[5] est un outil capable d'afficher les détails de la classe décompilée comme ses méthodes, package etc. Il suffit de compiler la classe avec javac et puis la décompiler par la commande javap suivie du nom de la classe. Un autre outil est le *Bytecode Outline*, un plugin pour l'éditeur Eclipse qui permet de désassembler les fichiers de classe Java. Il peut aussi traduire le code source d'un élément de la classe ou de toute la classe en bytecode Java ou en format ASM[35] dont nous traitons plus tard. La figure 3.2 présente un exemple d'utilisation du plugin Bytecode Outline pour traduction du code source au bytecode Java.

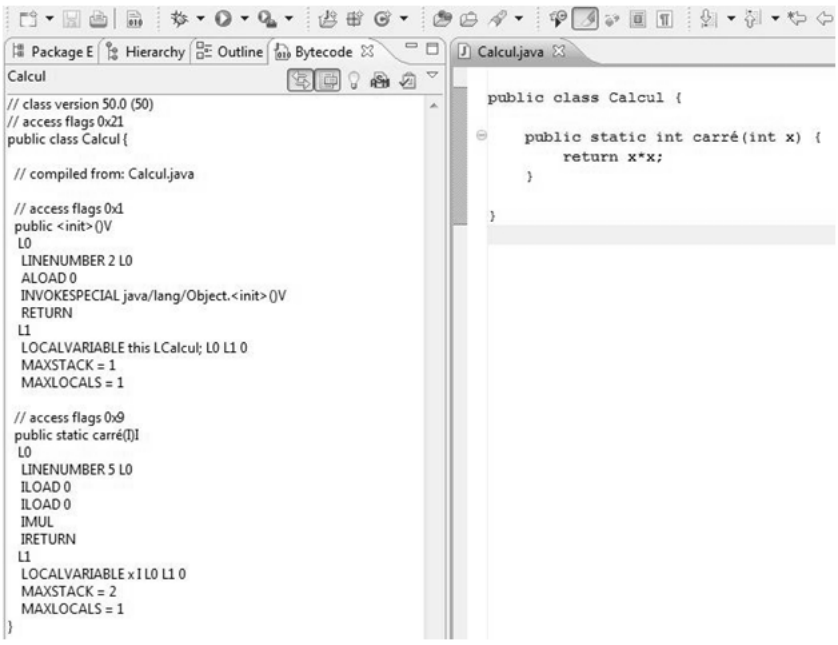

Figure 3.2 – Exemple avec Bytecode Outline

3.2.1 Défis de l'instrumentation

Modifier directement des instructions en bytecode n'est pas une tâche facile. Les spécifications de la machine virtuelle Java imposent des limites au nombre d'instructions dans une méthode par exemple ou le nombre d'attributs dans une classe, ce qui restreint le nombre d'instructions pouvant être injectées. Modifier le *constant pool* en

ajoutant de nouvelles constantes par exemple exige de mettre à jour le compteur d'entrées. L'ajout à la réserve de constantes est moins compliqué s'il est effectué à la fin sans modification de l'ordre original, sinon des changements d'instructions sont inévitables. En cas de branchement, il se peut que des instructions précédentes nécessitent des changements, une instruction 'ifeq' pourrait devenir un 'ifneq' ou un 'goto'. Au cas où l'instrumentation avant la sortie d'une méthode est souhaitée, une attention spéciale doit être accordée en cas de branchement précédant le retour ou en cas d'instructions de retour multiples de façon à insérer le code d'instrumentation avant tous les points de sortie. Aussi, si le code d'instrumentation ajoute une nouvelle valeur sur la pile ou une nouvelle variable à la méthode, les attributs `max_stack` et `max_locals` doivent être remplacés par les valeurs convenables. Il existe encore bien d'autres changements qui donnent du fil à retordre comme les offsets dans la table des exceptions, la table des variables locales etc.

Le risque d'erreur est omniprésent lors de l'instrumentation du bytecode. Bien que des outils tels que ASM[35] simplifient le processus de génération du bytecode, il est assez facile de créer accidentellement des séquences d'instructions invalides qui peuvent conduire à des exceptions de format de classes à l'exécution. De telles erreurs sont très difficiles à surveiller.

Toutefois travailler directement au niveau du bytecode offre plusieurs avantages. Cette technique permet de contourner le besoin de compiler le code source de l'application cible, une tâche qui s'avère particulièrement encombrante pour des programmes de grande taille à dépendances multiples. De plus, cette approche donne la possibilité de modifier des classes sans disposer de leur code source. D'ailleurs, manipuler du bytecode permet un haut degré de contrôle du code. Les valeurs précises des variables, les noms et signatures de méthodes qui seront véritablement appelées et bien d'autres informations sont disponibles à ce niveau. De plus, il existe plusieurs outils de génération de bytecode et de traduction de code en bytecode qui facilitent la tâche de l'instrumentation et réduisent la complexité de l'écriture manuelle des instructions en bytecode.

3.2.2 Outils d'instrumentation

Il existe plusieurs librairies pour fins d'instrumentation de bytecode comme BCEL[34], Javassist[50] et Serp[79]. Soot[75] est un framework d'analyse et de transformation de bytecode qui offre quatre représentations intermédiaires du code. Il est souvent utilisé pour l'optimisation du bytecode. JOIE[33] et JikesJT[47] sont d'autres librairies de manipulation du bytecode implémentées en Java.

ASM [35] est un framework d'analyse et de manipulation de bytecode Java. Il permet de modifier des classes existantes aussi bien que de générer dynamiquement des classes en format binaire. ASM fournit des transformations et des algorithmes d'analyses communs faciles à assembler en des transformations complexes. Bien que ASM offre certaines fonctionnalités présentes également dans d'autres librairies, sa particularité est sa petite taille et sa rapidité, ce qui le rend très sollicité par les systèmes dynamiques où la performance est critique. ASM est basé sur une approche originale qui consiste à utiliser le patron de conception 'visiteur' qui peut parcourir l'arborescence visitée sans la représenter explicitement sous forme d'objets. Une documentation complète et un plugin Eclipse pour transformer le code en format ASM sont disponibles pour simplifier l'usage de l'outil et aider à surmonter la difficulté de créer des transformations sophistiquées en cachant les complexités du bytecode.

Notre outil *dyko* utilise ASM pour transformer les classes vu les avantages qu'offre celui-ci. La rapidité de ASM permet de réduire le temps d'instrumentation des classes. Son architecture à base de patron 'visiteur', permet l'instrumentation des systèmes de grande taille sans se soucier de contraintes de limite de représentation des données. Sa modularité permet d'assembler ou de désassembler facilement plusieurs transformations. Ainsi que d'autres facilités de calcul et de mise à jour automatique d'offsets lors d'ajouts de nouvelles constantes ou de variables locales par exemple.

3.2.3 Exemple d'instrumentation avec ASM

Pour transformer les classes en ASM, un lecteur de classes `ClassReader` et un générateur de classes `ClassWriter` sont nécessaires. Le `ClassReader` décompose la

classe et le `ClassWriter` réécrit la nouvelle classe dans son format de classe standard, entre les deux, une ou plusieurs transformations peuvent être appliquées. `InvokeStats` est le nom de la transformation considérée ici comme exemple.

```
ClassReader cr = new ClassReader(bytecode);
ClassWriter cw = new ClassWriter(cr, ClassWriter.COMPUTE_MAXS);
ClassVisitor cv = new CheckClassAdapter(cw);
cv = new InvokeStats(cv);
cr.accept(cv, 0);
```

La transformation `InvokeStats` permet de calculer le nombre des instructions d'appels dans une méthode réparti selon le type de l'instruction ; `INVOKESTATIC`, `INVOKESPECIAL` ou `INVOKEVIRTUAL` etc. La méthode `visit` de `InvokeStats` permet de visiter la classe courante. Si aucune condition sur les classes à modifier n'est spécifiée, toutes les classes vont subir la transformation.

Il est possible de préciser des conditions pour limiter la portée de l'instrumentation à certaines classes ou méthodes. Un exemple de condition simple serait de restreindre les méthodes considérées aux méthodes des classes de l'application par exemple. Le code suivant illustre un exemple d'un tel filtre.

```
@Override
    public MethodVisitor visitMethod(int access, String name,
            String desc, String signature, String[] exceptions) {
        if(isStandardLibrary(className)) {
            return libVisitor.visitMethod(access, name, desc,
                signature, exceptions);
        } else {
            return appliVisitor.visitMethod(access, name, desc,
                signature, exceptions);
        }
    }
```

À l'entrée d'une méthode, le nom de sa classe qui n'est autre que la classe en cours de visite est consulté. S'il s'agit d'une classe des librairies standards, la méthode est visitée selon le visiteur de méthodes `libVisitor`, sinon le visiteur `appliVisitor` est appliqué. `AppliVisitor` peut ainsi contenir une ou plusieurs transformations désirées à être appliquées sur les classes de l'application uniquement.

Revenons à présent à l'exemple du compteur des instructions d'appels. `InvokeStats` est un `ClassAdapter` qui est une implémentation de l'interface `ClassVisitor` de ASM.

```
public class InvokeStats extends ClassAdapter {
    private String className;
    public InvokeStats(ClassVisitor cv) {
        super(cv);
    }

    @Override
    public void visit(int version, int access, String name,
            String signature, String superName,
            String[] interfaces) {
        super.visit(version, access, name, signature,
            superName, interfaces);
        className = name;
    }

    @Override
    public MethodVisitor visitMethod(int access, String name,
            String desc, String signature, String[] exceptions) {
        MethodVisitor mv = cv.visitMethod(access, name, desc,
                signature, exceptions);
        return new InvokesCounter(mv, className + "." + name
                + desc);
    }
}
```

Après une variable pour enregistrer le nom de la classe visitée et un constructeur, figure la méthode `visit`. Cette méthode permet de visiter la classe dont les propriétés sont précisées en paramètres. Ici, nous ne faisons que sauvegarder le nom de la classe qui est disponible à ce niveau pour une utilisation postérieure.

L'interface `ClassVisitor` possède d'autres méthodes pour visiter tous les éléments de classe, comme `visitField` pour visiter les attributs de classe ou `visitInnerClass` pour visiter une classe interne etc. Les méthodes de l'interface `ClassVisitor` doivent être visitées selon l'ordre suivant tel que précisé dans la documentation de ASM :

```
visit visitSource? visitOuterClass?
```

```
( visitAnnotation | visitAttribute )*
( visitInnerClass | visitField | visitMethod )*
visitEnd
```

La méthode `visitMethod` permet de son côté de transférer le contrôle au niveau de la méthode. Elle retourne un `MethodVisitor` qui est une autre interface de ASM pour visiter les éléments de la méthode. Dans la classe `InvokeStats`, toutes les méthodes de la classe seront transformées par le visiteur `InvokesCounter`.

```
public class InvokesCounter extends MethodAdapter {
    /* Déclarations */
    public InvokesCounter(MethodVisitor mv, String fqn) {
            super(mv);
            currentInvokes = new int[5];
            method = fqn;
        }

    @Override
    public void visitMethodInsn(int opcode, String owner,
            String name, String desc) {
        switch (opcode) {
        case Opcodes.INVOKESTATIC:
            currentInvokes[STATIC_INDEX]++; break;
        case Opcodes.INVOKESPECIAL:
            currentInvokes[SPECIAL_INDEX]++; break;
        case Opcodes.INVOKEVIRTUAL:
            currentInvokes[VIRTUAL_INDEX]++; break;
        case Opcodes.INVOKEINTERFACE:
            currentInvokes[INTERFACE_INDEX]++; break;
        case Opcodes.INVOKEDYNAMIC:
            currentInvokes[DYNAMIC_INDEX]++; break;
        default:
            break;
        }
        super.visitMethodInsn(opcode, owner, name, desc);
    }

    @Override
    public void visitEnd() {
        super.visitEnd();
        invokes.put(method, currentInvokes);
    }
}
```

Un adaptateur de classe permet des transformations au niveau des éléments de la classe, alors qu'un adaptateur de méthode peut accéder et modifier les instructions de la méthode. Pour chaque type ou groupe d'instructions, correspond une méthode de l'interface `MethodVisitor`, dont la méthode `visitJumpInsn` utilisée pour les instructions de branchement, ou la méthode `visitVarInsn` qui sert à manipuler les variables locales.

Dans notre exemple, la méthode `visitMethodInsn` qui correspond à la visite des instructions d'appels de méthodes, est réimplémentée. Pour calculer le nombre des instructions d'appels, des variables correspondantes aux différents types d'instructions d'appels sont sauvegardées par méthode. A chaque visite d'une des instructions d'appels, la variable correspondante est incrémentée.

3.3 Structures de données de profilage

Notre approche consiste à modifier le bytecode pour créer le graphe d'appels dynamique complet du programme profilé. Le code d'instrumentation inséré dans les méthodes, sert à construire la structure du graphe d'appels et à collecter l'information rapportée sur les appels et les allocations d'objets. Cette section présente les structures de données utilisées par notre outil.

3.3.1 Noeud

Un noeud du graphe d'appel dynamique est associé à une méthode unique invoquée lors de l'exécution. Il rassemble toute l'information relative à cette méthode.

- `int methodId` : Identifiant de la méthode.
- `int[][] callsites` : Méthodes appelées par la méthode courante organisées par site d'appel. `callsites[i]` sont tous les identifiants des méthodes appelées du site d'appel i.
- `int[] allocTotal` : Pour chaque numéro de site d'allocation d'objet i, `allocTotal[i]` représente le nombre total d'allocations au site i.

- `int[] allocTypes` : Types des objets alloués indexés par numéro de site d'allocation.

Le noeud correspondant à la méthode M est distingué par un identifiant unique. Il contient tous les appelés par M à tous les sites d'appels lors de l'exécution courante. Le nombre, les sites et les types des allocations d'objets créés par M sont aussi reportés par le noeud de cette méthode.

3.3.1.1 Identifiant de méthode

A chaque méthode est attribué un identifiant de type `int` selon sa signature unique de la forme `CLASSE.NOM_DE_METHODE(PARAMETRES)`. Manipuler des entiers comme identifiants de méthode au lieu de noms en chaînes de caractères permet un gain important en performance.

La gestion des identifiants de méthodes est assurée du côté du serveur d'instrumentation. Une table de type `Map<String,Integer>` sauvegarde les identifiants attribués aux méthodes. Elle enregistre les signatures de méthodes instrumentées ainsi que leurs identifiants.

- `MethodIdAssigner.reset() : void`
  Réinitialise la table des identifiants de méthodes. Cette méthode est invoquée au début de toute nouvelle session de profilage d'un programme.

- `MethodIdAssigner.assignId(String method) : int`
  Crée un nouvel identifiant pour la signature de méthode reçue en paramètre, ou retourne son attribut si la méthode existe déjà dans la table.

- `MethodIdAssigner.generateFile(File f) : void`
  Génère un fichier contenant tous les noms de méthodes avec leurs identifiants respectifs.

3.3.1.2 Nombre de sites d'appels

Un compteur `calleesPC` par méthode visitée est incrémenté avant tous les appels de méthodes. À savoir, avant toutes les instructions `INVOKEVIRTUAL`, `INVOKESPECIAL`, `INVOKESTATIC`, `INVOKEINTERFACE` ou `INVOKEDYNAMIC` apparaissant dans le corps de la méthode instrumentée.

Calculer le nombre maximal exact des sites d'appels pour chaque méthode est une exigence d'implémentation. Cette donnée nous est très utile pour réduire en mémoire la taille de notre structure de graphes.

3.3.1.3 Nombre d'allocations d'objets

Un compteur `allocPC` par méthode visitée est incrémenté avant toutes les allocations d'objets, ce qui signifie avant toutes les instructions `MULTIANEWARRAY`, `ANEWARRAY` et `NEW`. En se basant sur l'apparition d'une de ces instructions dans le code au lieu des invocations des constructeurs d'objets, nous sommes certains qu'une instruction correspond à l'allocation d'un seul objet. Sachant que tous les constructeurs d'objets autres que celui de `java.lang.Object` doivent faire appel à un autre constructeur, le nombre d'invocations de constructeurs ne correspond pas au nombre d'allocations d'objets.

3.3.2 Graphe d'appels

La construction des graphes d'appels commence dès l'atteinte d'une classe `main` de l'application ou d'un initialiseur de classe `<clinit>` de l'application. Une variable statique globale déclenche la collecte des informations pour les graphes. Elle est inactive jusqu'à la rencontre du premier point d'entrée du programme. Une autre variable par thread assure l'activation et la désactivation de l'instrumentation des méthodes. Elle est désactivée par exemple pour le thread `DestroyJavaVM` et les threads du groupe `system`, qui sont exclus du profilage.

3.3.2.1 Création du graphe d'appels

Un graphe par thread est maintenu en permanence. Le graphe contient la liste des noeuds de toutes les méthodes appelées par ce thread, ainsi que le nom et le groupe du thread. Sa racine est un noeud fictif nommé `START`, ayant comme fils un ou plusieurs points d'entrée de l'application.

- `initialize(): void`
  Initialise la construction du graphe pour le thread courant. Cette méthode est appelée une seule fois à un des points d'entrée du programme.

- `createCGM(String threadName, ThreadGroup threadGroup): CallGraphManager`
  Crée un nouveau graphe d'appel pour le thread `threadName` appartenant au groupe `threadGroup`.

3.3.2.2 Gestion du graphe d'appels

Le bytecode des méthodes est modifié de façon à insérer des appels aux méthodes responsables de la création et la gestion des noeuds des graphes d'appels. Les opérations sur les graphes d'appels sont effectuées par les méthodes suivantes :

- `enter(int caller,int site,int method,int totalAlloc, int totalCallees)`
  Ajoute la méthode[1] `method` au graphe d'appel à son entrée comme appelant de la méthode courante `caller`. Elle crée le noeud pour la méthode au besoin en utilisant `totalAlloc` et `totalCallees` pour allouer les tableaux de ses appelants avec la taille exacte.

- `enterClinit(int clinit,int totalAlloc,int totalCallees)`
  Est utilisée pour les initialiseurs de classes `<clinit>`. Ce traitement spécial crée un noeud pour l'initialiseur de classe comme fils du noeud fictif de la racine.

[1] Une méthode désigne un constructeur ou une méthode

- `recordAllocation(int method,String type,int pc, int totalAlloc,int totalInvokes)`
  Enregistre une allocation d'objet de type `type` au site `pc` par la méthode `method`.

- `recordAllAllocs(int method,int[] allocs,int totalInvokes)`
  Enregistre toutes les allocations d'objets `allocs` effectuées par la méthode `method`.

- `recordCall(int methodId,int callPC)`
  Sauvegarde sur une pile l'identifiant de la méthode `methodId` appelée depuis la méthode courante et le site d'appel `callPC`. Cette technique est utilisée spécialement pour certaines méthodes qui présentent une exception au processus d'ajout de paramètres aux méthodes invoquées. Les méthodes natifs ainsi qu'une liste de méthodes citée plus loin dans la section 3.5.2 ne permettent pas la modification de leurs signatures dû à certaines limites de la JVM. Ainsi le gestionnaire de graphes d'appels assure la gestion d'une pile par thread dédiée à ces méthodes aux signatures immuables. À l'entrée d'une méthode, son identifiant et le site d'appel sont empilés. Cette information reste présente sur la pile aussi longtemps que la méthode est active. À la sortie de la méthode, la pile est dépilée avec `removeCall`.

- `removeCall(int methodId, int site)`
  Dépile la méthode présente sur le sommet de la pile à la fin de son exécution. Cette méthode, ainsi que `recordCall` gèrent l'entrée et sortie des méthodes *native* et de quelques méthodes spéciales qui ne permettent pas l'ajout d'un nouveau paramètre.

3.4 Étendue du profilage

3.4.1 Couverture des classes et méthodes

Dyko instrumente toutes les classes chargées, qu'elles soient des classes de l'application, des librairies standards ou des librairies de parties tierces. L'outil est implémenté de façon à traiter uniformément toutes les classes. Bien que les modifications apportées au code peuvent changer d'une classe à une autre selon les transformations applicables

pour chacune d'elles, elles sont toutes acheminées de la même façon. Précisons que les classes de *dyko* lui-même ne sont pas et ne doivent pas faire objet de l'instrumentation.

Quant aux méthodes, toutes celles exécutées sont instrumentées sauf les cas d'exceptions suivants :

- `native registerNatives` de `java.lang.Object`
- `finalize()V` de `java.lang.Object`
- L'initialiseur de classe de `java.lang.Object`
- Toute méthode créée par l'instrumentation

Les trois premières méthodes, si instrumentées, font échouer le processus de démarrage de la machine virtuelle. Ainsi leur exclusion était nécessaire. De plus, les nouvelles méthodes ajoutées aux classes par *dyko* ne doivent pas figurer sur les graphes et par suite ne nécessitent pas d'être instrumentées.

3.4.2 Couverture des sites d'appel

À l'exception des méthodes non instrumentées citées plus haut, tous les sites d'appels aux méthodes sont traités. L'instrumentation ne diffère pas, dans la majorité des transformations appliquées, entre un constructeur, un initialiseur de classe ou une méthode ordinaire. Seules les méthodes *native* possèdent une transformation propre.

Pour chaque méthode qualifiée *native*, une nouvelle méthode java non native est créée dont la signature est similaire à la première avec un préfixe ajouté au nom de la méthode. La nouvelle méthode préfixée invoque l'originale et toutes les méthodes faisant appel à la méthode native sont transformées pour appeler à la place la nouvelle méthode Java. Sachant que le code des méthodes *native* n'est pas modifiable ou parfois inaccessible, créer des méthodes intermédiaires en code Java donne la possibilité d'instrumenter les sites d'appels aux méthodes *native*.

3.4.3 Couverture des sites de création d'objets

Dyko assure une couverture complète des sites de création d'objets. Chaque noeud de méthode enregistre les sites de création d'objets dans cette méthode avec leurs sites et le nombre d'objets créés. Tous les sites de création d'objets sont rapportés sur le noeud au fur et à mesure de l'exécution.

Une des améliorations apportées était de permettre l'enregistrement des objets alloués une seule fois à la sortie de la méthode si le nombre d'objets créés dépasse 10 objets. Cependant pour garantir l'enregistrement de tous les objets créés en cas de fin anormale, le code original de la méthode est entouré par un try-finally qui force le traitement avant la sortie de la méthode.

3.5 Implémentation

Cette section présente les étapes principales de l'implémentation de notre outil.

3.5.1 Version initiale

Initialement, dyko utilisait une pile pour garder trace de toutes les méthodes invoquées. L'information concernant une méthode est empilée à son entrée et dépilée à sa sortie. Au sommet de la pile figure l'information de la méthode en cours d'exécution. Son noeud, le site d'appel auquel elle a été appelée ainsi que la liste des gestionnaires des exceptions (*exception handlers*) actifs sont enregistrés.

À l'appel d'une méthode ou à l'allocation d'un objet, il suffit de lire l'information au dessus de la pile pour connaître la méthode responsable. En cas de lancement d'une exception, la liste des gestionnaires actifs permet de retrouver la méthode responsable en dépilant les méthodes présentes sur la pile jusqu'à avoir cette méthode au sommet de la pile. Ce traitement permet de garder la consistance de la liste des méthodes en cours d'exécution même lors d'un comportement inhabituel.

Bien que simple, cette approche souffre de certaines limites. Elle n'assure pas la couverture de la totalité des méthodes appelées. Elle ne permet pas d'enregistrer les appels

de méthodes Java invoquées par le code natif. De plus, elle résulte en une performance médiocre.

3.5.2 Retrait de la pile

Dû à l'interaction massive avec la pile et le temps d'exécution important investit dans sa mise à jour, enlever la dépendance à cette structure fut la première amélioration apportée. La pile nécessitait une actualisation au moins à chaque entrée et sortie de méthode ainsi qu'à chaque entrée et sortie de bloc try-catch pendant toute l'exécution. La haute fréquence des appels de méthodes dans la plupart des applications en Java, explique la dégradation en performance causée par cette version.

Pour rendre disponible l'information que fournissait la pile, la solution est de faire passer en paramètre à toute méthode appelée le graphe d'appel du thread courant, l'identifiant de son appelant et son numéro de site d'appel. Il suffit de localiser dans le graphe le noeud de l'appelant par le biais de son identifiant disponible et de le modifier pour insérer ou mettre à jour le noeud de la méthode appelée. Avec cette technique, nous nous sommes détachés de l'obligation de garder trace des gestionnaires d'exceptions par nous même.

Néanmoins, cette approche n'est pas applicable pour toutes les méthodes appelées. Certaines méthodes n'autorisent pas la modification de leurs signatures par définition ou font échouer le processus de démarrage si leurs signatures sont altérées. Voici la liste des méthodes pour lesquelles nous ne pouvons pas insérer des paramètres :

- les initialiseurs de classes `<clinit>`
- `hashCode()I`
- `toString()Ljava/lang/String;`
- `equals(Ljava/lang/Object;)Z`
- `clone()Ljava/lang/Object;`
- `find(Ljava/lang/String;)J`

Ainsi, un traitement spécial est effectué pour ces méthodes. Une pile enregistre la liste des méthodes spéciales en cours d'exécution. Avec le même principe que sa précédente, une méthode est empilée à l'entrée et dépilée à la sortie. De cette manière, si une méthode est appelée par une des méthodes spéciales, elle extrait l'information de son appelant depuis la pile.

3.5.3 Réduction de l'usage de mémoire

Par la suite, nous nous sommes intéressés à rendre notre structure de graphes la plus compacte possible. Pour se faire, il fallait réduire au minimum l'espace en mémoire occupé par les noeuds des méthodes.

À l'aide de la structure `MethodNode` de ASM, nous avons pu calculer d'avance le nombre exact maximal de sites d'appels et de sites de création d'objets pour chaque méthode. Cette information disponible au moment de l'entrée de la méthode, permet de créer des noeuds de méthodes concis. Seule la taille nécessaire pour les tableaux enregistrant les méthodes appelées et les objets alloués est utilisée lors de la création du noeud.

3.5.4 Gestion des sites de création d'objets

D'autre part, nous avons rendu le traitement pour les allocations d'objets dans certaines méthodes plus paresseux. Au lieu d'enregistrer une allocation à la fois, l'information est emmagasinée et est enregistrée une seule fois à la sortie de la méthode. Nous avons senti le besoin de faire de la sorte pour les méthodes qui créent un grand nombre d'objets. Une expérimentation du seuil pour lequel nous allons fixer le nombre maximal d'objets alloués par méthode pour adopter cette technique, nous a amené à trouver que la valeur de 10 objets est avantageuse pour la majorité des résultats observés. Cependant plusieurs précautions ont été prises pour garantir l'enregistrement de l'information même lors de fin anormale de l'exécution.

3.5.5 Résolution des sites d'appels

Rendue à ce point, la méthode `enter`, responsable de la création du noeud de la méthode et de l'enregistrement du nouvel appel, est automatiquement appelée à l'entrée de chaque méthode. Cependant, dans le cas de certains appels, l'appelant et le site d'appel sont toujours les mêmes. Le traitement de la méthode `enter` devient ainsi redondant et son invocation n'est plus nécessaire dès la deuxième visite du même site d'appel. Il s'agit des sites d'appels résolus invoqués par les instructions `INVOKESTATIC` et `INVOKESPECIAL`.

Pour les sites résolus, il suffit d'enregistrer l'information à la première visite du site d'appel et d'éviter la répétition du même traitement lors des visites suivantes. Un drapeau *(flag)* par site est donc nécessaire pour connaître si le site a déjà été visité ou non. Ainsi, un compteur du nombre total de sites résolus par classe nous permet de choisir la structure appropriée pour chaque classe. La transformation insère un entier pour les classes dont le nombre total des sites résolus est inférieur à 32 sites, ou un tableau d'entier avec la taille exacte nécessaire pour la représentation des sites si le nombre dépasse 32.

Chaque bit de l'entier ou d'un élément du tableau est un booléen qui informe sur l'état de visite d'un site d'appel. Ainsi, lors de l'exécution du code d'instrumentation inséré avant le site d'appel, l'appel de la méthode `enter` est conditionné par le booléen correspondant.

Remarque

Tout au long de l'implémentation de notre outil, nous avions à faire face à plusieurs difficultés dues à la nature intrusive de l'injection du code d'instrumentation. D'une part, la dépendance entre les classes des librairies et leur chargement non séquentiel a rendu difficile l'instrumentation des méthodes de librairies. Certaines méthodes de librairies standards présentent des cas d'exceptions et nécessitent des traitements adaptés. D'autre part, le code d'instrumentation modifie la signature des méthodes existantes et rajoute de nouvelles méthodes à certaines classes, chose qui ébranle le processus de réflexion.

De plus, le calcul des blocs d'activation[2] (*stack frames* ou *activation records*) est complètement perturbé par les changements apportés au bytecode. Puisque l'instrumentation rajoute des paramètres aux méthodes, injecte de nouvelles variables locales et même modifie la structure de contrôle de flux de la méthode, il est impératif de recalculer ou voir ajouter de nouveaux blocs d'activation. Cependant le calcul manuel des blocs d'activation s'avère très compliqué. Par conséquent, nous transformons les classes à la version 5 qui ne fait pas usage des blocs d'activation.

3.6 Exemple de transformation

Pour illustrer certaines des transformations qu'effectue *dyko*, considérons comme exemple la méthode suivante. La méthode comprend une création d'objet, un appel virtuel et un appel à une méthode statique.

```
static void method() {
    Object var = new Object();
    var.toString();
    delete(var);
}
```

En réalité, les modifications se font au niveau du bytecode en rajoutant de nouvelles instructions au bytecode ou en modifiant les existantes. Pour améliorer la lisibilité, nous expliquons ici les transformations avec du code haut niveau en cachant les détails du langage de la machine virtuelle.

L'instrumentation de la méthode `method`, crée deux versions pour celle-ci. La première version est la suivante. Elle a la même signature de la méthode originale mais pas le même corps de méthode.

```
static void method() {
    CallGraphManager graph;
    if (instrument)
        graph = Thread.currentThread().getCGM();
    else
```

[2] À l'appel d'une méthode, un bloc d'activation est créé pour enregistrer l'information nécessaire pour l'exécution, telles que les valeurs des paramètres et des variables locales de la méthode. Ce bloc est stocké sur une pile de blocs d'activation.

```
            graph = DummyCGM.instance();

        // Appel de la nouvelle méthode avec les paramètres
        // method(graphe d'appels, ID appelant, site d'appel, boolean)
        method(graph, -1, -1, false);
    }
```

La première version de la méthode récupère le graphe d'appel courant. Pour ceci, elle teste si l'instrumentation est activée et retourne le graphe d'appel courant en faisant appel à une méthode d'instrumentation `getCGM` qui a été injectée dans la classe Thread. Si l'instrumentation est désactivée, le graphe est un DummyCGM qui est un type null à méthodes vides. La méthode ensuite fait appel à la deuxième version de la méthode avec les paramètres injectés. Les valeurs passées en paramètre ici ne sont que symboliques, les vraies valeurs sont récupérables du côté de la méthode appelée si elle a été invoquée par cette version. En général, cette version de la méthode ne sera pas appelée car la version avec les paramètres sera invoquée avec les bonnes valeurs pour les paramètres empilées avant l'instruction de l'appel.

La méthode suivante est la deuxième version de la méthode instrumentée. Celle-ci rajoute des paramètres à la signature de la méthode mais conserve le traitement initial de la méthode. Du code d'instrumentation est injecté entre les lignes originales de la méthode pour collecter les données pour la construction du graphe d'appel ou pour préparer du traitement à cet effet.

```
static void method(Graph graph, int caller, int callsite, boolean visited) {
    // Données précalculées:
    // Identifiant de la méthode courante 2
    // Nombre total des sites d'allocations 1
    // Nombre total des sites d'appels 3

    if (!visited)
        // Crée le noeud de la méthode courante à son entrée
        // enter(graphe d'appel, appelant, site d'appel, ID méthode, total
        // des sites d'allocations, total des sites d'appels);
        enter(graph, caller, callsite, 2, 1, 3);

    // Enregistre une nouvelle allocation
    // recordAllocation(ID méthode, type, site d'allocation, total des sites
    // allocations, total des sites d'appels);
```

```
    recordAllocation(2, "java.lang.Object", 0, 1, 3);

    // visited2 = boolean correspondant à l'état du site d'appel 0
    // Ajout de paramètres à la méthode appelée: graph d'appel, ID de
    // l'appelant, site d'appel et boolean
    Object var = new Object(graph, 2, 0, visited2);

    try {
        // La signature de la méthode toString ne peut être modifiée
        // Enregistre l'appelant et le site d'appel sur la pile
        recordCall(2, 1);
        var.toString();
    } finally {
        // Dépile l'appelant et le site d'appel
        removeCall(2, 1);
    }

    // visited3 = boolean correspondant à l'état du site d'appel 2
    // Ajout de paramètres à la méthode appelée: graph d'appel, ID de
    // l'appelant, site d'appel et boolean
    delete(var, graph, 2, 2, visited3);
}
```

Dès les premiers stades de l'instrumentation, des données sont calculées et rendues disponibles avant l'exécution de la méthode. Un identifiant unique relatif à sa signature lui est affecté et les nombres exacts de sites d'allocations d'objets et de sites d'appels sont calculés.

L'entrée de la méthode est le moment d'ajouter un noeud pour la méthode au graphe d'appel. Puisque notre méthode d'exemple est statique, l'appel à la méthode `enter` responsable de la création du noeud n'est pas automatique, il est conditionné par un booléen (ici `visited`) qui informe si la méthode a été visité précédemment ou non. Si la méthode est visité pour la première fois, l'exécution procède à l'appel de `enter` sinon l'appel est passé.

La première ligne de code de la version originale de la méthode est une création d'objet. Cet événement correspond à deux sous-événements ; l'enregistrement de l'allocation par l'instruction `new` et l'appel du constructeur, nécessitant deux traitements indépendants au niveau de l'instrumentation. D'un côté, la méthode `recordAllocation` en-

registre sur le noeud de la méthode courante du graphe l'information sur l'objet alloué, à savoir son type et le numéro de site d'allocation. De l'autre, l'appel du constructeur est instrumenté de façon similaire à tout appel de méthode Java. Des paramètres sont rajoutés à la méthode constructeur de la classe `Object` invoquée.

La ligne apparaissant à la suite dans le code original de la méthode est un appel de la méthode `toString` sur l'objet créé. `toString` est une méthode Java qui présente une exception à l'approche d'injection de paramètres dans les méthodes. Puisque sa signature est immuable, les valeurs qui sont passées en paramètres dans le cas général sont stockées sur une pile par l'appel de la méthode `recordCall` qui précède l'invocation de `toString`. Les deux invocations sont encapsulées dans un try-finally pour assurer l'exécution de la méthode `removeCall` qui dépile l'information précédemment empilée. L'usage du try-finally s'avère indispensable en cas de lancement d'une exception qui peut être déclenchée par quasiment n'importe quelle instruction bytecode. Exécuter `removeCall` lors de lancement d'une exception permet de garder la consistance de la pile maintenue même en cas de fin anormale.

Ensuite, la dernière ligne du code de la méthode est un appel virtuel. La méthode appelée est une méthode Java qui ne présente aucun cas d'exception, l'instrumentation lui rajoute alors en paramètres le graphe d'appel, l'identifiant de la méthode courante, le site d'appel ainsi qu'un booléen confirmant si elle a déjà été invoquée ou non.

Le calcul du booléen se fait à ce niveau, c'est à dire avant de procéder à l'appel lui-même. Dans chaque classe contenant des sites résolus, l'instrumentation injecte un nouvel attribut de type `int` ou `int[]` selon le nombre de sites dans chaque classe. Cet attribut sert à sauver l'état de visite de tous les sites de la classe à raison d'un bit par site. Avant l'appel d'un site résolu, des opérations logiques permettent de lire le bit correspondant au site et de faire passer l'information à la méthode appelée (le paramètre `visited` dans l'exemple). Cependant si la méthode n'a pas été visitée auparavant, une autre série d'opérations logiques est exécutée pour mettre le bit à 1 pour le marquer comme visité aux prochains appels.

CHAPITRE 4

ÉVALUATION

Le présent chapitre expose l'évaluation de notre outil *dyko* de génération de graphes d'appels dynamiques complets pour les programmes Java. L'évaluation a pour but de mesurer et d'étudier l'effet de ralentissement de l'exécution causé par le profilage, ainsi que la portion de temps occupée par l'instrumentation lors de l'application des différentes transformations du code. En validant l'outil par trois séries de benchmarks, nous cherchons à prouver la possibilité de profiler pour générer dynamiquement les graphes d'appels complets pour une diversité de programmes en un temps acceptable. Les versions considérées ont été présentées en détail dans le chapitre précédent. Il convient de noter que seules ces cinq versions à modifications majeures ont été choisies pour faire objet de l'étude d'évaluation, d'autres étapes intermédiaires ont été omises.

Le chapitre est organisé comme suit. Nous commençons par définir l'environnement de l'exécution, les séries de benchmarks utilisées et la démarche suivie pour la collecte de nos mesures de performances. Puis nous présentons des métriques utilisées pour l'évaluation. Le chapitre ensuite détaille les résultats observés pour les versions considérées parmi la progression du processus de développement du profileur. Nous terminons par une comparaison de nos résultats avec quelques travaux existants.

4.1 Environnement d'expérimentation

4.1.1 Matériel

Nous avons utilisé un serveur de calcul à 24 processeurs double-coeur *Intel Xeon X5650 2.67GHz*. Chaque processeur possède 12Mb de mémoire cache. La station dispose de 96Gb de mémoire vive et exécute le système d'exploitation Fedora Core 14.

4.1.2 Benchmarks

Pour évaluer notre outil, nous effectuons des tests sur les séries de benchmarks SPECjvm98, SPECjvm2008 et DaCapo.

4.1.2.1 SPECjvm98

SPECjvm98[16] est une suite de tests qui mesure le rendement en performance pour des plateformes clientes de la machine virtuelle Java (JVM). Elle contient huit tests différents, dont cinq sont des applications réelles ou sont dérivées des applications réelles. Sept tests sont utilisés pour le calcul des métriques de performance. Le tableau 4.I présente une brève description des benchmarks de la série SPECjvm98.

Lors de nos tests, nous utilisons la taille 100 pour les benchmarks de cette série.

Benchmark	Description
_201_compress	Un programme de compression à base de l'algorithme Lempel-Ziv-Welch (LZW).
_202_jess	Une version Java du système expert à base de règles CLIPS de NASA, agréé par Sandia Laboratories.
_205_raytrace	Un traceur de rayons qui fonctionne sur une scène représentant un dinosaure de Sun Microsystems.
_209_db	Logiciel de gestion de données pour tests de performance, propriété de IBM.
_213_javac	Le compilateur Java de JDK 1.0.2 par Sun Microsystems.
_202_mpegaudio	Une application qui décompresse les fichiers audio conformes à la spécification ISO MPEG-3, Fraunhofer Institut fuer Integrierte Schaltungen.
_227_mtrt	Une variante de _205_raytrace utilisant plusieurs fils d'exécution (threads).
_228_jack	Un générateur d'analyseur syntaxique pour Java, Sun Microsystems.

Tableau 4.I – Description des benchmarks de la série SPECjvm98

4.1.2.2 SPECjvm2008

SPECjvm2008 (Benchmark Java Virtual Machine)[49] est une série de tests pour mesurer la performance d'un environnement d'exécution Java (JRE). La série contient plusieurs applications simulant la charge de travail dans le milieu industriel ainsi que des tests qui se concentrent sur les fonctionnalités de base de Java. La série se focalise sur la performance de la JRE exécutant une application unique, elle reflète la performance du matériel et de la mémoire, mais a une faible dépendance des entrées/sorties de fichiers et ne comprend aucune interaction avec le réseau des machines. La charge de travail pour SPECjvm2008 est similaire au calcul d'une variété d'applications communes à usage général. Ces caractéristiques reflètent l'intention de ce benchmark d'être applicable pour les mesures de performance de base sur une grande variété de systèmes client et serveur. Le tableau 4.II présente une brève description des benchmarks de cette série.

Pour nos besoins, nous exécutons la version `lagom` de la série SPECjvm2008 qui utilise une charge de travail fixe.

4.1.2.3 DaCapo

DaCapo-9.12-bach[15] est la dernière version des séries de benchmarks DaCapo. Cette suite de tests est conçue comme un outil de benchmarking pour Java par les communautés du langage de programmation, de la gestion de la mémoire et des communautés d'architecture informatique. Elle se compose d'un ensemble de programmes open source similaires aux applications du monde réel avec des charges non-triviales de mémoire. Le tableau 4.III présente une brève description des benchmarks de la série DaCapo.

4.1.3 Démarche

Chaque programme des trois séries de benchmarks est exécuté selon deux modes de profilage. Le premier mode que nous appelons dans ce qui suit ***exécution en ignorant les transformations***, instrumente les classes mais procède à l'exécution de l'application sans prendre en considération les modifications apportées au code. Ce qui signifie

Benchmark	Description
compiler	Compilateur Java qui utilise le compilateur frontal OpenJDK (JDK 7 alpha)
compress	Un programme de compression à base de l'algorithme Lempel–Ziv–Welch (LZW).
crypto	Un programme de cryptographie utilisant aes, rsa et signverify.
derby	Successeur de _209_db de SPECjvm98, utilise une base de données open source écrite en Java pur.
mpegaudio	Similaire à _222_mpegaudio de SPECjvm98. La bibliothèque mp3 a été remplacé par JLayer, une bibliothèque LGPL mp3.
scimark	Programme largement utilisé par l'industrie comme une référence en virgule flottante, de NIST.
serial	Programme qui sérialise et désérialise les primitives et les objets en utilisant les données du benchmark JBoss.
startup	Programme pour mesurer le temps de démarrage de la JVM pour différentes charges de travail.
sunflow	Programme de tests de perfomance pour visualisation des graphiques.
xml	Transformations XSLT et validation de documents XML en utilisant l'implémentation JAXP

Tableau 4.II – Description des benchmarks de la série SPECjvm2008.

qu'à l'exécution, les classes originales sont utilisées et non les classes contenant le code d'instrumentation injecté. Le deuxième mode nommé ***exécution profilée*** instrumente de la même façon les classes de l'application mais contrairement au premier mode, celui-ci utilise les nouvelles classes lors de l'exécution. Ainsi l'instrumentation prend effet à l'exécution.

L'agent de dyko possède un argument qui permet de choisir ou non d'appliquer les modifications de l'instrumentation à l'exécution.

- En mode ***exécution en ignorant les transformations*** :

```
java -agentpath:Path/dyko.profile.agent/Debug/libdyko.so=ignoreModifs
-Xbootclasspath/a:Path/dyko/bin/ -jar Application.jar
```

- En mode ***exécution profilée*** :

Benchmark	Description
avrora	simule un certain nombre de programmes exécutés sur une grille de microcontrôleurs AVR.
batik	produit un certain nombre d'images Scalable Vector Graphics (SVG) basées sur les tests unitaires de Apache Batik.
eclipse	exécute certains des tests de performances (non-gui) jdt pour l'IDE Eclipse.
fop	analyse un fichier XSL-FO, l'analyse et le formate pour générer un fichier PDF.
h2	exécute un benchmark similaire à JDBCbench, exécutant un certain nombre d'opérations contre un modèle d'une application bancaire.
jython	interprète le benchmark de Python pybench
luindex	utilise lucene pour indexer un ensemble de documents ; travaux de Shakespeare et la Bible de King James.
lusearch	utilise lucene pour faire une recherche textuelle de mots clés sur un corpus de données comprenant les oeuvres de Shakespeare et la Bible King James.
pmd	analyse un ensemble de classes Java pour une gamme de problèmes de code source.
sunflow	traite un ensemble d'images en utilisant le ray tracing.
tomcat	exécute un ensemble de requêtes sur un serveur Tomcat en récupérant et en vérifiant les pages résultantes.
tradebeans	lance le benchmark daytrader via Java Beans vers un backend GERONIMO avec en mémoire h2 comme base de données sous-jacente.
tradesoap	lance le benchmark daytrader via SOAP vers un backend GERONIMO avec en mémoire h2 comme base de données sous-jacente.
xalan	transforme les documents XML en HTML

Tableau 4.III – Description des benchmarks de la série DaCapo.

```
java -agentpath:Path/dyko.profile.agent/Debug/libdyko.so
-Xbootclasspath/a:Path/dyko/bin/ -jar Application.jar
```

Nous mesurons les temps d'exécution de tous les programmes des trois séries de benchmarks pour les deux scénarios, d'abord en incluant les modifications apportées aux classes puis en les ignorant. Pour chaque cas, nous exécutons les programmes suc-

cessivement deux fois. La meilleure valeur observée parmi les deux est enregistrée pour chaque programme.

Pour mesurer le temps d'instrumentation et le ralentissement causé par la génération du profil du programme, il est nécessaire de bien définir les métriques à mesurer. Les transformations appliquées par *dyko* sont la composition de plusieurs adaptateurs de classes dont certains sont fournis par ASM, d'autres sont propres à dyko servent à la construction du graphe d'appel dynamique. Dans tous les cas, toutes les classes passent par le serveur d'instrumentation qui achemine les classes chargées vers la machine virtuelle de *dyko* responsable de l'instrumentation. Ainsi, il faut bien distinguer entre le temps attribué à l'instrumentation des classes seulement, entre l'exécution du code modifié et entre le temps rajouté par l'utilisation du serveur et les transformations ajoutées pour fins de vérification du bytecode.

Il est alors primordial de bien définir le ***temps de base*** d'exécution d'un programme. Notre cas de base ne peut pas être l'exécution normale non profilée des programmes, car le temps nécessaire pour faire passer les classes à travers le serveur d'instrumentation et le vérificateur de bytecode `CheckClassAdapter` de ASM sera attribué à tord au temps de profilage. Le temps de base doit représenter impartialement tout temps qui n'est pas dû à l'instrumentation ou à l'exécution de la version instrumentée des classes de l'application.

Pour se faire, nous avons exécuté les benchmarks à travers le serveur d'instrumentation mais sans modifier les classes. Les classes sont lues par le `ClassReader`, vérifiées par le `CheckClassAdapter` de ASM et réécrites par le `ClassWriter`. Le temps que prend l'opération est notre temps de base pour l'exécution d'un programme. De cette façon, la différence entre le temps du cas de base et le temps de la version profilée renseigne sur le temps additionnel causé par le profilage.

4.2 Métriques

Pour évaluer la performance de notre outil, il est nécessaire de mesurer le temps écoulé lors de l'instrumentation des classes ainsi que le ralentissement de l'exécution

causé par le profileur.

4.2.1 Temps d'instrumentation

Le temps d'instrumentation est le temps nécessaire pour modifier le bytecode des classes. C'est le temps qu'occupe la machine virtuelle de dyko à transformer toutes les classes qu'elle reçoit. Il est cependant difficile de calculer exactement le temps nécessaire à l'instrumentation des classes. Plusieurs facteurs différencient une exécution d'une autre. Le processus de chargement en soi n'est pas uniforme. De plus, d'autres opérations peuvent se produire pendant le chargement.

Nous calculons le temps d'instrumentation selon la formule suivante :

$$\textit{Temps d'instrumentation} = \textit{Temps d'exécution en ignorant les transformations} - \textit{Temps de base}$$

où,

- *Le temps d'exécution en ignorant les transformations* inclut le temps de l'instrumentation des classes et le temps d'exécution de l'application en utilisant la version originale des classes.
- *Le temps de base* est le temps d'exécution du programme sans instrumentation mais en faisant passer les classes par le serveur d'instrumentation pour appliquer le vérificateur de bytecode `CheckClassAdapter`.

En calculant ainsi le temps d'instrumentation, nous fournissons la meilleure estimation possible du temps que consomme l'instrumentation seulement. Bien que lors d'une exécution profilée du même programme, le chargement et l'instrumentation de nouvelles classes qui n'ont pas été observées à l'exécution non instrumentée peuvent survenir. Le temps d'instrumentation de ces classes n'est pas alors pris en compte dans le temps que la formule ci-dessus mesure. Vu que nous ne pouvons séparer le temps d'instrumentation du temps consacré pour le chargement ou le profilage, nous avons choisi d'estimer le

temps d'instrumentation de la sorte et d'attribuer tout autre temps additionnel au temps de profilage. Ce dernier signifie le temps d'exécution du code inséré. Nous n'avons pas d'autre choix que de calculer le temps d'instrumentation de cette façon puisque l'instrumentation des classes se déroule *online*, pendant l'exécution. De plus, en séparant le temps d'instrumentation calculé du temps total de l'exécution profilée, nous aboutissons à un temps de ralentissement dû à l'exécution du code seulement, ce qui nous permet de comparer nos résultats de performance à des travaux existants qui instrumentent au préalable statiquement.

4.2.2 Facteur de ralentissement

Instrumenter dynamiquement les applications dégrade leur performance. Il s'agit ici de mesurer l'affaiblissement en performance de l'application. Le facteur de ralentissement est le facteur avec lequel sera multiplié le temps d'exécution à cause du profilage. Nous calculons le facteur de ralentissement par la formule suivante :

$$\textit{Facteur de ralentissement} = \frac{\textit{Temps d'exécution profilée}}{\textit{Temps d'exécution en ignorant les transformations}}$$

Calculer de la sorte permet de déduire le ralentissement dû à l'exécution du code instrumenté.

4.3 Résultats

Cette section présente les résultats de performance des trois séries de benchmarks SPECjvm98, SPECjvm2008 et DaCapo. Pour chaque benchmark, nous énonçons le facteur de ralentissement des programmes à l'application du profilage. Nous présentons également les pourcentages qu'occupe le temps d'instrumentation des classes lors du profilage des programmes. Les résultats rapportés décrivent le comportement des programmes pour les cinq étapes d'implémentation présentées dans le chapitre précédent.

4.3.1 Facteur de ralentissement

Les figures 4.1, 4.2 et 4.3 montrent respectivement les facteurs de ralentissement observés pour les séries de benchmarks SPECjvm98, SPECjvm2008 et DaCapo à travers les cinq versions.

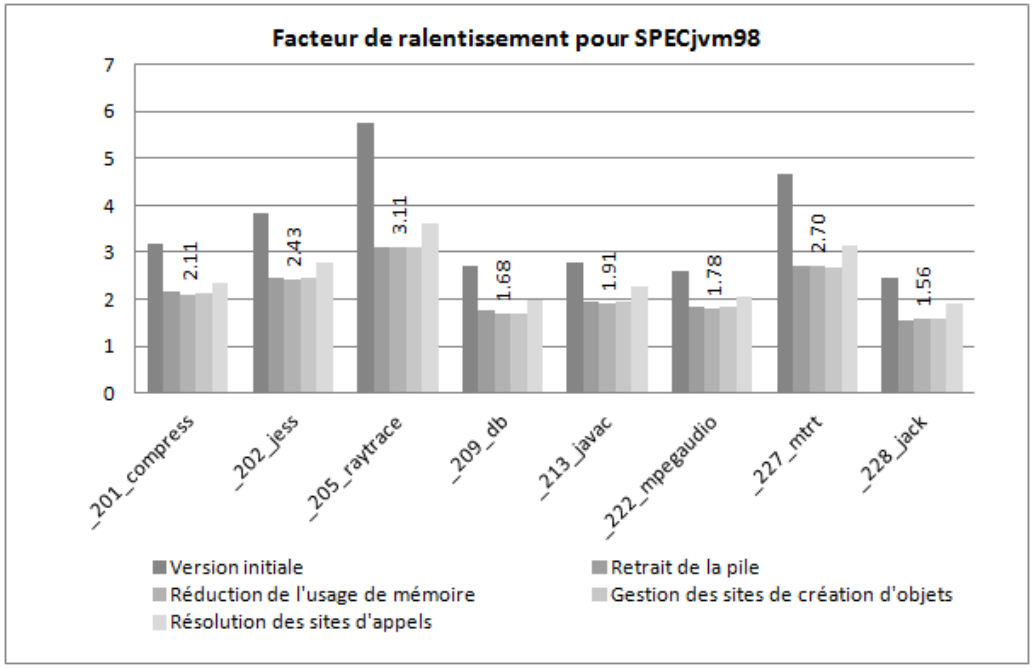

Figure 4.1 – Facteur de ralentissement pour SPECjvm98

La version initiale utilise une pile pour gérer les entrées et sorties des méthodes ainsi que pour enregistrer les exceptions actives. Cette version naïve enregistre le plus grand facteur de ralentissement parmi les versions considérées.

Notons que le profilage par *dyko* de tous les programmes de la série SPECjvm98 aboutit à une exécution correcte avec cette version. Elle multiplie leurs temps d'exécution par un facteur allant de 2.5 pour _228_jack jusqu'à 5.8 pour _205_raytrace. Elle aboutit également à une exécution correcte pour les programmes compress, scimark et startup de la série SPECjvm2008. Les benchmarks compress et scimark enregistrent un ralentissement de 11.7 et 7.7 respectivement. La version initiale profile la plupart des programmes de la série DaCapo. Les facteurs de ralentissement les plus élevés enregistrés sont 11.4 pour h2 et 7.5 pour sunflow. Le reste des programmes causent un

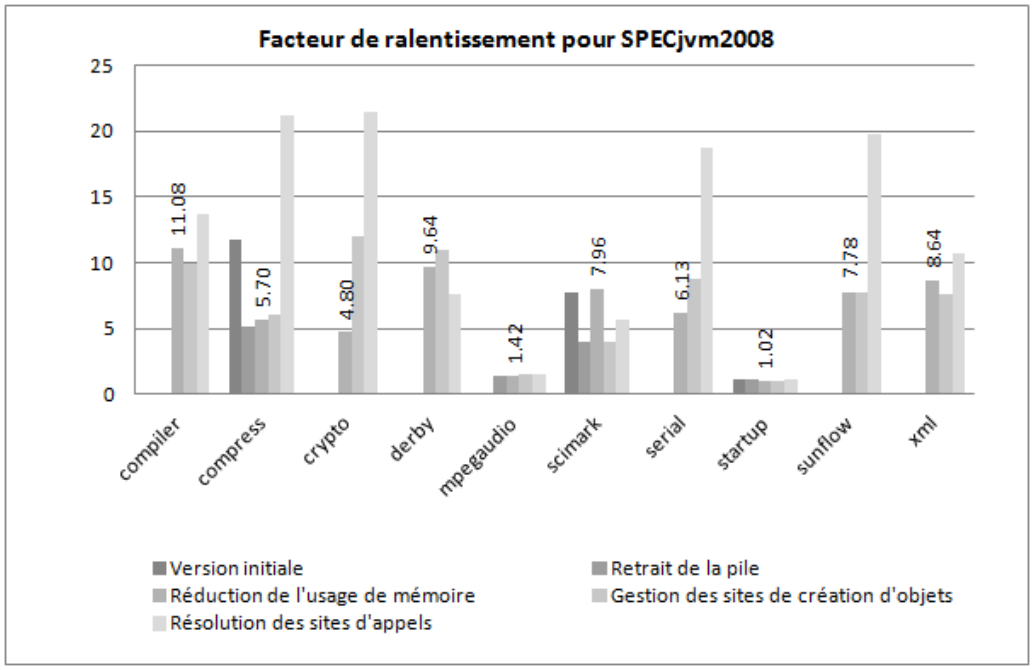

Figure 4.2 – Facteur de ralentissement pour SPECjvm2008

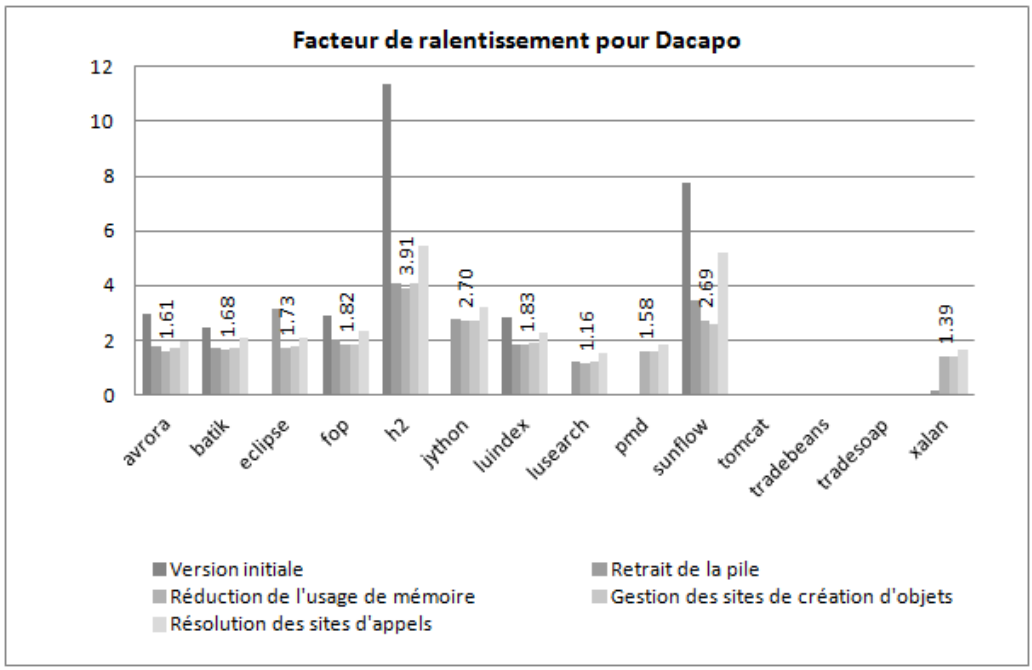

Figure 4.3 – Facteur de ralentissement pour DaCapo

ralentissement inférieur à 2.9.

Cependant des problèmes ont été constatés pour certains programmes. Vu que la version enregistre les exceptions actives sur la pile, l'implémentation utilise un type `long` qui peut garder simultanément l'état de jusqu'à 63 blocs try-catch. Or cette limite s'est avérée insuffisante pour le benchmark compiler de SPECjvm2008. Pour mpegaudio de la même série, un autre problème a été observé. L'instrumentation d'une méthode résultait en 74218 instructions, ce nombre dépasse la limite d'instructions permise par méthode selon la spécification de la machine virtuelle Java. D'ailleurs, des exceptions ont été levée par manque de mémoire nécessaire pour poursuivre l'exécution du programme. Tel est le cas pour crypto, derby, serial et sunflow de SPECjvm2008, et eclipse et pmd de DaCapo.

En supprimant la pile utilisée dans la version initiale, la charge de travail requise à l'entrée de toute méthode a augmenté. Il faut donc insérer du code pour récupérer l'information sur la méthode appelante disponible jadis au sommet de la pile. Cette version réduit d'une part le temps requis pour le traitement des exceptions, et d'autre part elle injecte plus de code dans les méthodes.

Le gain en performance avec la version «Retrait de la pile» est indéniable. Le facteur de ralentissement de tous les programmes de SPECjvm98 se situe en dessous de 3.1, valeur enregistrée par _205_raytrace. Les benchmarks de SPECjvm2008 ont aussi marqué une réduction du facteur de ralentissement. Les plus importantes baisses sont de 11.7 à 5.1 pour compress et de 7.7 à 3.9 pour scimark. Pour les programmes de DaCapo, le facteur de ralentissement se situe entre 4.1 de h2 et entre 1.2 de lusearch. Cette version rajoute à la liste des programmes profilables, mpegaudio de SPECjvm2008 ainsi que jython et lusearch de DaCapo. Toutefois, l'insuffisance de mémoire persiste avec compiler, crypto, derby et sunflow de SPECjvm2008 et avec eclipse, pmd et xalan de DaCapo.

La version nommée «Réduction de l'usage de mémoire» crée les noeuds de méthodes en adoptant la taille des tableaux utilisés dans la structure au nombre exact de sites d'appels selon chaque méthode. La figure 4.1 montre de très faibles variations du facteur de

ralentissement pour les benchmarks de SPECjvm98 comparé à la version précédente. De plus, cette version permet de profiler les benchmarks de SPECjvm2008 et DaCapo qui ne pouvait être profilé auparavant à cause de l'insuffisance de mémoire. Le ralentissement induit par cette version se situe entre 1.0 pour startup et 11.1 pour compiler de SPECjvm2008, et entre 1.2 pour lusearch et 3.9 pour h2 de DaCapo.

La gestion des sites de création d'objets consiste à implémenter une nouvelle solution pour l'enregistrement des données sur les objets créés. Au lieu de le faire à chaque site de création d'objet, l'enregistrement de tous les sites se fait une fois à la sortie de la méthode si le nombre d'objets créé dépasse un certain seuil. Dans la version exécutée, le seuil est fixé à 10 objets. La majorité des benchmarks des trois séries montrent une très faible à aucune variation dans le facteur de ralentissement. Cependant certaines augmentations importantes ont été aperçues pour crypto et serial de SPECjvm2008. Ceci peut s'expliquer par la nécessité d'implémentation à entourer le code de la méthode avec un try finally. Ce choix inévitable peut nuire à l'application des optimisations du JIT et peut par conséquent dégrader la performance.

La résolution des sites d'appels INVOKESTATIC et INVOKESPECIAL a pour but d'éliminer la redondance du traitement requis à l'entrée des méthodes invoquées par ces instructions. Puisque l'information à enregistrer est immuable pour le cas de ces sites, cette version vise à gagner en temps d'exécution. Cependant les résultats ont révélé autrement, la résolution des sites d'appels marque une augmentation du facteur de ralentissement pour la majorité des programmes. Les valeurs les plus distinguées sont 21.4 pour crypto et 21.2 pour compress de la série SPECjvm2008, ainsi que 5.5 pour h2 et 5.2 pour sunflow de DaCapo. Il s'avère par conséquent que l'effort nécessaire pour implanter la solution rajoute au ralentissement causé par le profilage plus qu'il permet d'en gagner.

Remarque

Les benchmarks tomcat, tradebeans et tradesoap de DaCapo ne peuvent être profilées avec la version actuelle de *dyko*. Tradebeans et tradesoap résultent en une erreur d'initialisation interne affichant le message `GBean is now in the FAILED state`. Jusqu'à présent, une inspection des différentes transformations appliquées nous a permis de lier le problème à la modification des signatures de méthodes par ajout de nouveaux paramètres. Cependant, nous n'avons pas encore réussi à identifier la ou les méthodes qui causent ce comportement. Nous croyons que les Beans Java utilisent des mécanismes de réflexion différents des mécanismes réguliers gérés par *dyko*. Le programme tomcat montre également un autre comportement singulier où l'exécution se fige sans afficher aucune exception ou message d'erreur.

4.3.2 Temps d'instrumentation

Le temps occupé par l'instrumentation des classes varie d'un programme à un autre et d'une exécution à une autre. Il est difficile de comparer les temps d'instrumentation puisque plusieurs facteurs interagissent avec ces mesures. Le chargement des classes, le nombre de classes chargées, la taille des classes, les modifications apportées à chaque classe sont certains des paramètres qui entrent en jeu. Pour donner une idée des temps d'instrumentation mesurés, le tableau 4.IV fournit les mesures en secondes pour les programmes des trois séries de benchmarks exécutés.

Dans la section précédente, nous avons présenté le ralentissement induit par le profilage. Il serait intéressant d'étudier la proportion du temps qu'occupe l'instrumentation des classes parmi le temps additionnel rajouté par le profilage. Les figures 4.4, 4.5 et 4.6 présentent respectivement cette proportion pour les benchmarks des séries SPECjvm98, SPECjvm2008 et DaCapo.

Les programmes de la série SPECjvm98 sont des programmes aux temps d'exécution courts qui ne dépassent pas 5 secondes sans profilage. L'instrumentation des classes

Temps d'instrumentation (en secondes)	Version initiale	Retrait de la pile	Réduction de l'usage de mémoire	Gestion des sites de création d'objets	Résolution des sites d'appels
_201_compress	0.827	0.957	0.959	0.978	1.594
_202_jess	0.892	0.959	0.942	1.005	1.492
_205_raytrace	0.819	0.885	0.942	0.969	1.368
_209_db	0.887	0.951	1.012	1.018	1.488
_213_javac	1.035	1.059	1.026	1.089	1.588
_222_mpegaudio	0.893	0.941	0.947	0.979	1.458
_227_mtrt	0.921	0.827	0.93	0.937	1.405
_228_jack	0.841	0.823	0.993	1.003	1.405
compiler					3.491
compress			0.012	0.9	
crypto			2.595	5.752	7.554
derby			0.685	2.46	4.728
mpegaudio		2.978	2.362	4.57	3.335
scimark	28.166	19.098	8.626	1.545	20.944
serial			2.568	4.326	4.89
startup	1.058	0.248	0.473	0.363	2.238
sunflow			2.836	2.199	4.615
xml			1.182	0.623	3.141
avrora	0.972	0.421	1.119	0.253	1.447
batik	1.953	1.557	1.844	1.954	3.065
eclipse			1.411	2.111	4.327
fop	1.463	1.518	1.457	1.677	2.712
h2	0.932	1.337	0.642	1.081	2.091
jython		2.098	2.636	2.377	4.626
luindex	0.99	1.067	1.199	1.202	1.713
lusearch		1.306	1.467	1.444	1.626
pmd			1.281	1.1	2.122
sunflow	1.191	1.208	1.299	1.292	2.054
tomcat					
tradebeans					
tradesoap					
xalan			1.09	1.247	1.952

Tableau 4.IV – Temps d'instrumentation des séries SPECjvm98, SPECjvm2008 et DaCapo en secondes

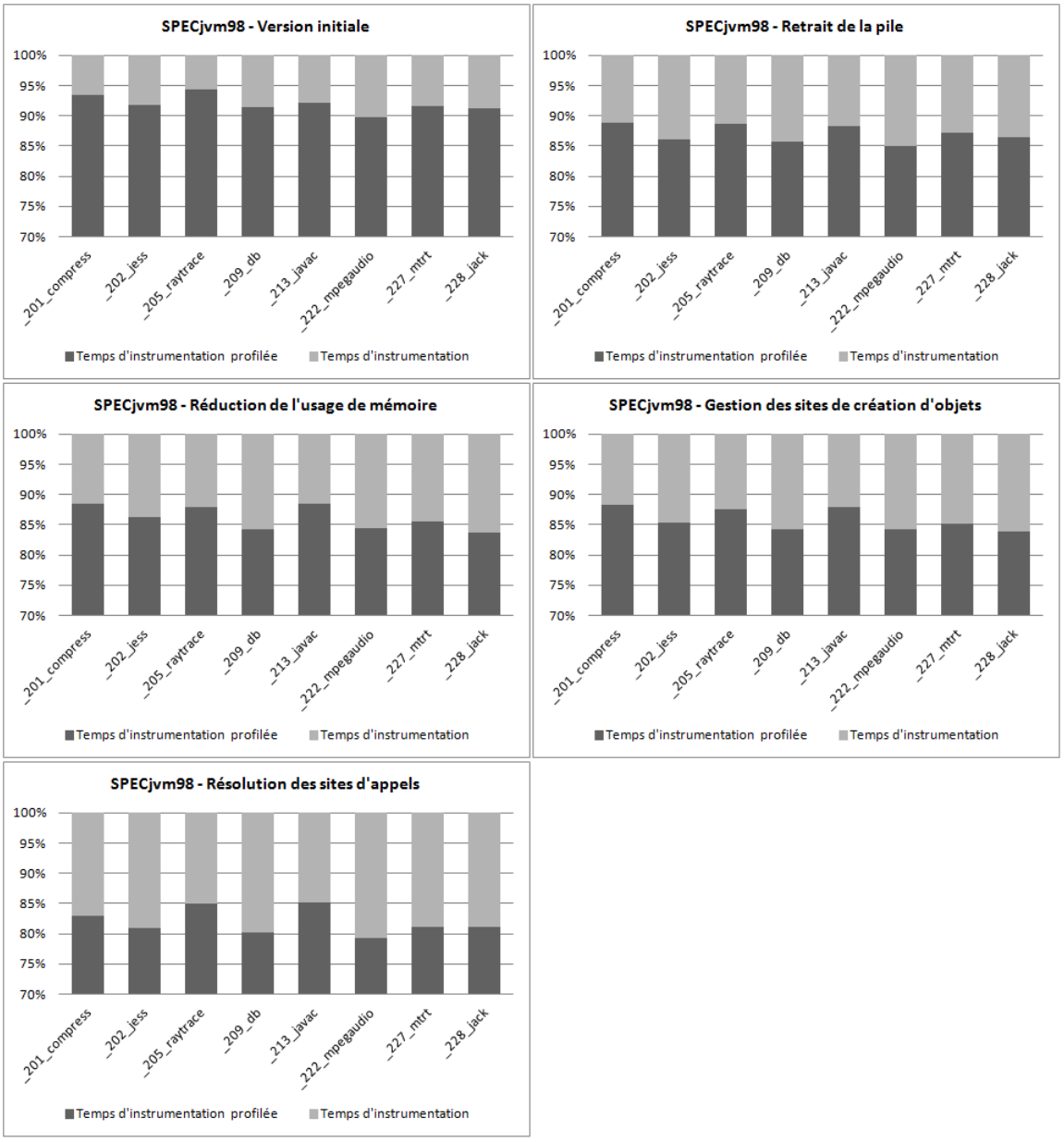

Figure 4.4 – Résultats de performance pour SPECjvm98

de la majorité de ces programmes occupe moins de 10% du temps de profilage à la version initiale. Cette proportion augmente avec chaque nouvelle version du profileur. Vu que la complexité de l'instrumentation croît avec les versions, il est naturel que l'instrumentation occupe une plus grande partie. Pour la version finale des résolutions des sites d'appels, l'instrumentation occupe une proportion de 15 à 20 % du temps de profilage.

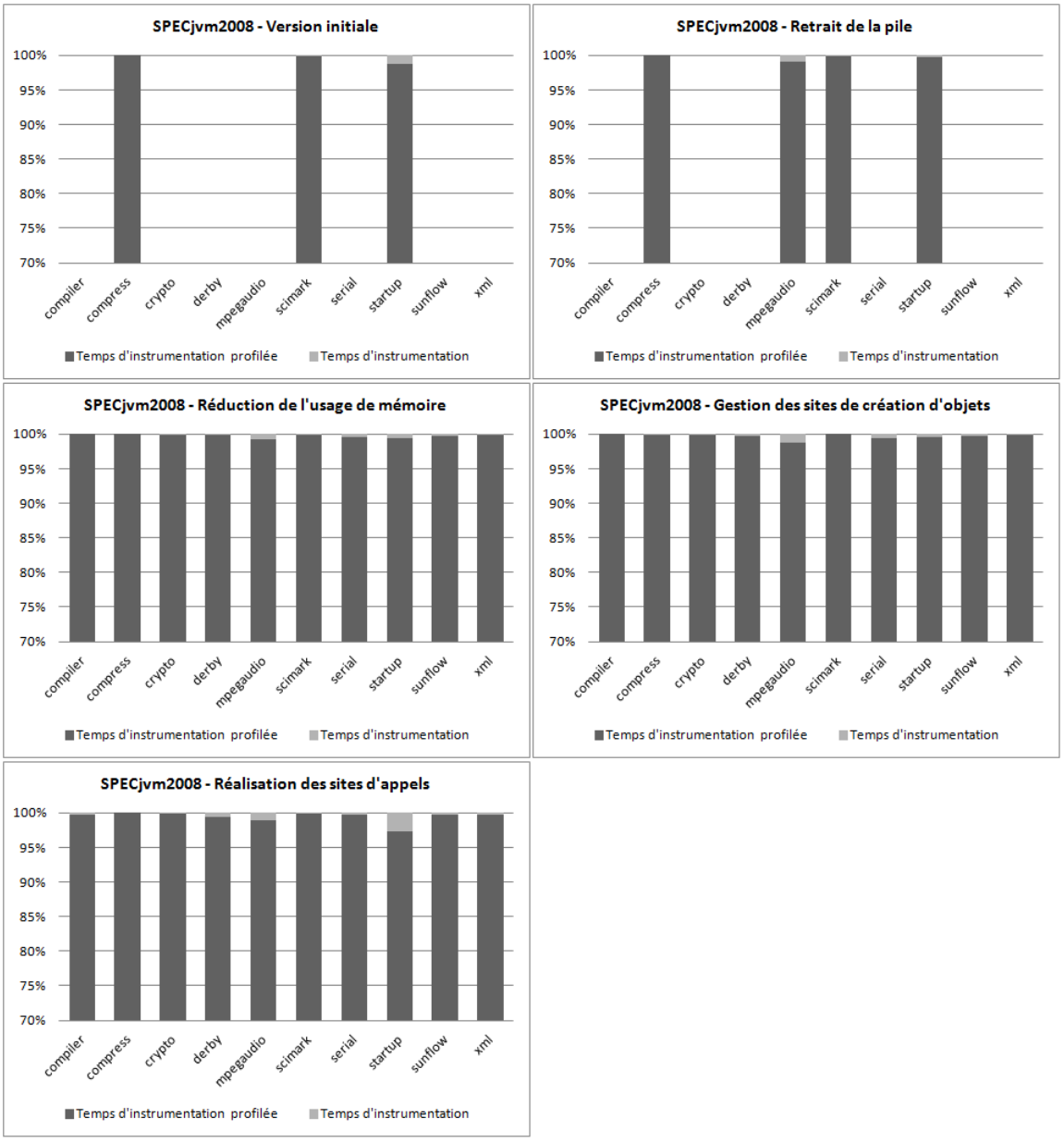

Figure 4.5 – Résultats de performance pour SPECjvm2008

Pour les benchmarks de la série SPECjvm2008, le temps d'instrumentation est presque négligeable pour la majorité des programmes. Sur la figure 4.5, le temps d'instrumentation est insignifiant. Ces programmes s'exécutent sur des durées plus importantes comparés aux programmes de la série précédente. Leurs temps d'exécutions sans profilage

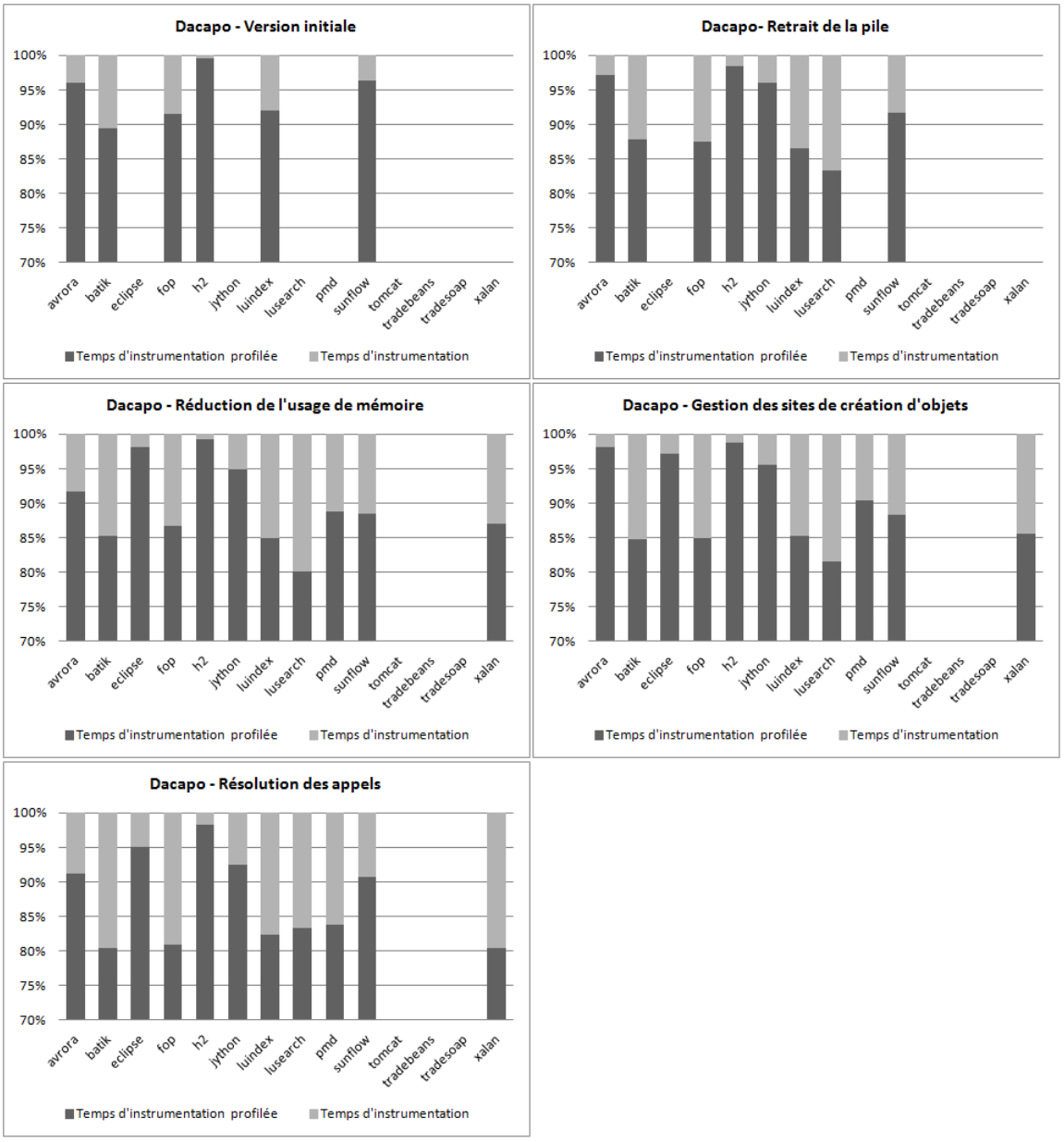

Figure 4.6 – Résultats de performance pour DaCapo

varient entre 80 secondes pour startup jusqu'à près de 65 minutes pour scimark.

Pour la série DaCapo, les programmes suivent la même allure. Sur les benchmarks à temps d'exécution court, la proportion du temps d'instrumentation devient plus perceptible aux dernières versions comme pour batik et fop. Pour h2 et eclipse dont le temps d'exécution est plus important, l'effet de l'instrumentation reste tout de même minime.

4.4 Comparaison

Dyko profile les benchmarks des séries SPECjvm98, SPECjvm2008 et DaCapo avec une moyenne de ralentissement illustrée par le tableau 4.V. La version nommée «Réduction de l'usage de mémoire» présente les meilleurs résultats. Elle fournit le même degré de précision que les deux versions qui suivent. Ces dernières font un traitement additionnel dans l'espoir de gagner en performance.

Moyenne du facteur de ralentissement	Version initiale	Retrait de la pile	Réduction de l'usage de mémoire	Gestion des sites de création d'objets	Résolution des sites d'appels
SPECjvm98	3.50	2.19	2.16	2.17	2.52
SPECjvm2008	6.84	2.88	6.42	6.96	12.13
DaCapo	5.04	2.37	2.01	2.05	2.70

Tableau 4.V – Moyenne de ralentissement pour SPECjvm98, SPECjvm2008 et DaCapo

Considérons l'outil JP de Binder et al[20]. Cet outil fait usage de l'instrumentation du bytecode Java pour construire des arbres de contextes d'appels. Bien que *dyko* ne peut produire à présent que des graphes d'appels dynamiques, nous pouvons comparer *dyko* à JP en matière de techniques d'instrumentation utilisées et de facteurs de ralentissement aperçus.

JP crée pour chaque thread un noeud pour regrouper toutes les invocations de méthodes partageant le même contexte d'appels. Analogiquement, pour éviter le coût élevé de la synchronisation, *dyko* crée un graphe d'appels par thread, contenant les noeuds de méthodes invoquées à raison d'un noeud pour chaque méthode.

Pour la génération d'identifiants de méthodes, JP ajoute pour chaque méthode un champ statique contenant le numéro qui lui est attribué. Dans l'initialiseur de la classe, une méthode responsable de l'assignation d'identifiants de méthode est appelée pour toutes les méthodes de cette classe. Contrairement, `dyko` gère la numérotation des méthodes invoquées dynamiquement à l'instrumentation pour ne pas alourdir le code par

cette étape, n'étant qu'une technique d'implémentation. Le serveur sauvegarde une liste de paires (identifiant, méthode), ainsi le code injecté utilise directement les numéros pour référer aux méthodes.

JP permet le profilage de mémoire par génération du nombre d'objets créés par type pour chaque contexte d'appel. *Dyko* calcule similairement le nombre d'objets alloués par type pour chaque méthode invoquée. D'autres travaux utilisent le nombre d'octets alloués comme métrique pour l'usage de mémoire. Dans notre cas ainsi que celui de JP, une estimation d'octets alloués par le programme peut être calculée à partir des données assemblées sans instrumentation additionnelle.

Cependant l'emplacement de l'instrumentation pour la collecte des données diffère. JP instrumente sur appel de constructeur et non sur l'instruction de l'allocation `new`. En bytecode Java, la création des objets consiste en deux étapes ; l'allocation avec l'instruction `new` et l'initialisation avec l'appel de constructeur.

JP instrumente sur appel de constructeur. Cependant l'allocation d'un objet en Java correspond à au moins deux appels de constructeurs dans la plupart des cas. JP utilise alors une analyse pour déterminer l'emplacement où insérer l'instrumentation pour éviter de refaire le traitement plusieurs fois pour le même objet. Toutefois cette technique est sujette à l'erreur dans certaines situations pour lesquelles elle renvoie un nombre incorrect d'objets créés. Si une exception est lancée après l'instruction `new` mais avant l'appel au constructeur, l'allocation de l'objet n'est pas visible puisque le code d'instrumentation n'est pas encore atteint. Aussi, des situations rares où des objets non initialisés peuvent être utilisés présentent une limite à cette technique vu que l'appel au constructeur n'a pas encore été observé.

Or *dyko* instrumente les allocations d'objets avant l'instruction `new` évitant ainsi des analyses supplémentaires puisque chaque allocation correspond à une seule instruction `new`. De plus, *dyko* rapporte exactement le nombre d'allocations survenues jusqu'au point où s'est rendue l'exécution même en cas d'exception ou fin anormale du programme.

En ce qui concerne sa technique de profilage, JP fournit deux niveaux de granularité pour collecter les données de contexte qui diffère selon le scénario de décomposition en blocs de base considéré. L'outil déclenche la collecte de données à base du nombre de blocs de base exécutés. Lors de l'exécution, une analyse permet de calculer le graphe de contrôle de flux et un compteur est continuellement incrémenté à l'entrée des blocs de base.

Le premier scénario, *default basic block analysis*, permet de subdiviser en de large blocs de base. Cette technique considère comme marque de fin d'un bloc de base toute instruction pouvant changer le contrôle de flux comme les sauts, les branchements, les retours de méthodes et les exceptions. Les appels de méthodes ne sont pas considérés par cette analyse. Bien qu'élargir les blocs de base permet de réduire les emplacements d'actualisation du compteur des instructions bytecode exécutées et par la suite réduire le ralentissement, cette technique souffre d'imprécisions en cas de fin anormale du programme. Au lancement d'une exception, le compteur de bytecodes dépasse le nombre réel d'instructions exécutées et ne peut pas par la suite localiser où s'est rendu l'exécution avant que l'exception survienne.

Le deuxième scénario d'analyse rajoute aux événements marquant la fin d'un bloc de base, les bytecodes susceptibles de lancer une exception comme une référence à un objet qui peut aboutir à `NullPointerException`. Par conséquent, un ralentissement plus important est observé à l'exécution pour une meilleure précision. Cependant en Java, toute instruction bytecode peut lancer une exception. Et vu que cette technique ne semble pas considérer une instruction par bloc de base, il nous est difficile de croire en l'exactitude du calcul des bytecodes exécutés en cas d'exception.

Par contre, même si *dyko* ne fait pas usage d'une analyse sophistiquée de flux de contrôle, il procure une précision parfaite en cas d'exception. Notre outil procède à l'analyse du code bytecode par bytecode. Certes l'instrumentation en début de blocs de base réduit le traitement supplémentaire du profilage à l'exécution mais *dyko* opte pour la précision et la complétude de l'information.

De plus, JP instrumente statiquement les classes des librairies standards et instrumente dynamiquement le reste des classes. JP soustrait ainsi le temps d'instrumenta-

tion des classes des librairies de son temps de profilage et le réduit davantage. L'outil requiert ainsi cette étape préparatoire d'instrumentation statique avant de lancer le profilage. Alors que *dyko* instrumente dynamiquement toutes les classes. Il est ainsi plus facile à utiliser malgré qu'il intègre au ralentissement l'instrumentation des classes de librairies.

Aussi, les auteurs de JP signalent une moyenne du facteur de ralentissement de 4.23 pour la série SPECjvm98, alors qu'elle est 2.16 avec *dyko*.

CHAPITRE 5

CONCLUSION

L'analyse du code est d'une importance cruciale pour détecter les vulnérabilités, vérifier les fonctionnalités et identifier des bogues et failles de sécurité susceptibles dans un programme. L'analyse du code peut être statique ou dynamique. Appliquer l'analyse statique pour des applications de taille industrielle s'avère cependant complexe et parfois impossible avec des ressources limitées. L'analyse dynamique par contre est capable de fournir des données précises concernant l'exécution en cours, très utiles pour l'étude du comportement. Néanmoins, l'analyse dynamique ralentit l'exécution et sature la mémoire à cause de la quantité importante de données collectées.

Pour les applications de taille industrielle auxquelles nous nous intéressons, il est plus adéquat de combiner les deux analyses statique et dynamique afin d'aboutir à la précision de l'analyse dynamique à un coup modéré. Ces approches se basent sur les données fournies par l'analyse dynamique pour déterminer les portions à problèmes dans le code, ces dernières feront l'objet de l'analyse statique par la suite. Cependant les outils d'analyse dynamique existants génèrent des données imprécises ou incomplètes, ou résultent en un ralentissement inacceptable de l'exécution.

Notre profileur *dyko* est capable de générer la totalité de l'information dynamique nécessaire à l'analyse comportementale des applications. L'outil produit les graphes d'appels dynamiques complets en instrumentant dynamiquement le bytecode à un coût raisonnable.

5.1 Profilage avec *dyko*

Notre outil *dyko* permet de fournir des données précises et complètes utiles pour l'étude de comportement des logiciels en Java. Il génère les graphes d'appels dynamiques complets incluant les appels aux méthodes des librairies et les appels aux méthodes non Java. L'outil fournit également de l'information additionnelle sur les sites

d'appels et les sites de création d'objets.

Dyko utilise la technique d'instrumentation dynamique du bytecode pour rassembler l'information et produire les graphes d'appels dynamiques. Modifier le bytecode lors de l'exécution offre un contrôle important sur l'étendue du profilage et permet d'accéder à toutes les classes chargées. Qu'elles proviennent de l'application, des librairies ou autres, toutes les classes chargées subissent uniformément l'instrumentation. De plus, travailler au niveau du bytecode offre une grande marge de manipulations qui ne pourraient être possibles au niveau du code source.

Profiler avec *dyko* est simple et ne nécessite aucune configuration spéciale ou préparation à l'exécution. *Dyko* peut être utilisé pour profiler toute application Java avec toute machine virtuelle compatible avec JVMTI.

5.2 Performance

Notre objectif est d'assurer la complétude de l'information rapportée et l'efficacité de l'instrumentation appliquée. Notre outil fait usage du framework ASM de manipulation de bytecode à base de patron visiteur. ASM se distingue par sa rapidité, sa petite taille et sa simplicité à créer des transformations complexes du bytecode. Ces caractéristiques le rendent très intéressant à exploiter dans les systèmes dynamiques comme le nôtre, où la performance est critique.

Nous avons testé *dyko* sur trois séries de benchmarks à charges de travail diverses et non triviales. Nous avons montré qu'avec *dyko*, il est possible de profiler des programmes du monde réel, obtenir une haute précision de l'information rapportée et ce à un coût raisonnable. La moyenne de ralentissement de l'exécution observée lors de notre évaluation varie entre 2.01 et 6.42.

5.3 Travaux futurs

5.3.1 Profilage de mémoire

Le profilage de mémoire rapporte le nombre d'octets utilisés par le programme. Même si nous nous sommes pas particulièrement intéressés à calculer cette métrique, très peu de changements sont nécessaires pour la mettre en place. *Dyko* fournit déjà le nombre et le type de tous les objets créés. Un calcul additionnel fournira une valeur approximative de la mémoire utilisée sans modification de l'outil. Toutefois, pour plus de précision, il serait nécessaire d'ajouter un traitement spécial pour les tableaux afin de calculer le nombre de cellules utilisées et rapporter exactement la taille qu'occupe chaque tableau.

5.3.2 Profilage partiel

Bien que notre objectif était de couvrir tous les appels exécutés, *dyko* peut être étendu pour permettre de choisir à profiler une section de l'exécution. Le profilage peut être déclenché à la première ou à la nième invocation d'une certaine méthode par exemple. Cette fonctionnalité est présente dans certains profileurs commerciaux pour permettre d'inspecter de près une partie du code et réduire le ralentissement induit par le profilage. Cependant elle fournit une vue partielle qui échoue à représenter fidèlement le comportement général du programme.

5.3.3 Profilage adaptatif

Les méthodes d'un programme n'étant pas toutes similaires, il serait plus judicieux de choisir le type d'instrumentation en fonction des méthodes à traiter. Il convient donc de penser qu'instrumenter automatiquement toutes les méthodes de manière similaire est une façon naïve de procéder. Choisir quelle instrumentation à appliquer pour chaque méthode selon son bytecode, est une idée qui nous semble très prometteuse. Lors de ce mémoire, nous avons appliqué ce principe pour sélectionner entre deux techniques d'instrumentation pour les sites d'allocation d'objets selon le nombre de sites présents

dans le code de chaque méthode. Une analyse sophistiquée du graphe de flux de contrôle peut être exploitée dans ce sens.

5.3.4 Profilage avec contexte

La génération des graphes d'appels dynamiques est la première étape vers la production des arbres de contextes d'appels des programmes. Lors de ce travail, nous avons exploré la possibilité de produire des graphes d'appels dynamiques complets au prix d'un ralentissement acceptable de l'exécution. Avec *dyko*, nous avons pu montrer qu'il est possible d'aller chercher la complétude avec des ressources admissibles.

L'auteur souhaite voir *dyko* évoluer vers un profileur d'arbres de contextes d'appels assurant la complétude des données fournies, avec un maximum de précision pour une analyse pointue du comportement.

BIBLIOGRAPHIE

[1] Jari Aarniala. Instrumenting Java bytecode. *Science*, 2005. URL http://www.cs.helsinki.fi/u/pohjalai/k05/okk/seminar/Aarniala-instrumenting.pdf.

[2] Manar H. Alalfi, James R. Cordy et Thomas R. Dean. Automated reverse engineering of UML sequence diagrams for dynamic web applications. Dans *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 287–294, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3671-2. URL http://portal.acm.org/citation.cfm?id=1547559.1548265.

[3] Glenn Ammons, Thomas Ball et James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. Dans *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. URL http://doi.acm.org/10.1145/258915.258924.

[4] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger et William E. Weihl. Continuous profiling : where have all the cycles gone ? *ACM Transactions on Computer Systems (TOCS)*, 15:357–390, November 1997. ISSN 0734-2071. URL http://doi.acm.org/10.1145/265924.265925.

[5] Oracle and/or its affiliates. Javap - The Java Class File Disassembler. URL http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javap.html. Copyright © 2004, 2010 Oracle and/or its affiliates.

[6] Oracle and/or its affiliates. JVM Tool Interface (JVMTI). URL http://download.oracle.com/javase/1.5.0/docs/guide/jvmti. Copyright © 2004 Oracle and/or its affiliates.

[7] Oracle Corporation and/or its affiliates. The Netbeans Profiler, JFluid Technology. URL http://profiler.netbeans.org/jfluid.html.

[8] Oracle Corporation and/or its affiliates. Netbeans IDE. URL http://netbeans.org/index.html. Copyright © 2012, Oracle Corporation and/or its affiliates.

[9] M. Arnold et P.F. Sweeney. *Approximating the calling context tree via sampling*. IBM T.J. Watson Research Center, New York, NY, USA, July 2000.

[10] Matthew Arnold, Stephen Fink, David Grove, Michael Hind et Peter F. Sweeney. Adaptive optimization in the Jalapeno JVM. Dans *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM. ISBN 1-58113-200-X. URL http://doi.acm.org/10.1145/353171.353175.

[11] Matthew Arnold et David Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. Dans *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. URL http://dx.doi.org/10.1109/CGO.2005.9.

[12] Matthew Arnold, Michael Hind et Barbara G. Ryder. Online feedback-directed optimization of Java. Dans *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 111–129, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1. URL http://doi.acm.org/10.1145/582419.582432.

[13] Matthew Arnold et Barbara G. Ryder. A framework for reducing the cost of instrumented code. Dans *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–

179, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. URL http://doi.acm.org/10.1145/378795.378832.

[14] AspectWerkz. AOP framework. URL http://aspectwerkz.codehaus.org/.

[15] DaCapo benchmark suite. Copyright 2001-2009 by the DaCapo Project. URL www.dacapobench.org.

[16] SPEC JVM98 Benchmarks. Last updated : Mon Sep 08 17 :25 :05 EDT 2008 Copyright 1995 - 2011 Standard Performance Evaluation Corporation. URL http://www.spec.org/jvm98/index.html.

[17] Walter Binder et Jarle Hulaas. Exact and Portable Profiling for the JVM Using Bytecode Instruction Counting. *Electronic Notes in Theoretical Computer Science*, 164:45–64, 2006.

[18] Walter Binder, Jarle Hulaas et Philippe Moret. Advanced Java bytecode instrumentation. Dans *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 135–144, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-672-1. URL http://doi.acm.org/10.1145/1294325.1294344.

[19] Walter Binder, Jarle Hulaas et Philippe Moret. Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation. Dans *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 91–100, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2880-5. URL http://portal.acm.org/citation.cfm?id=1306878.1307365.

[20] Walter Binder, Jarle Hulaas, Philippe Moret et Alex Villazon. Platform-independent profiling in a virtual execution environment. *Software Practice & Experience*, 39:47–79, January 2009. ISSN 0038-0644. URL http://portal.acm.org/citation.cfm?id=1464245.1464249.

[21] David Binkley. Source Code Analysis : A Road Map. Dans *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. URL http://dx.doi.org/10.1109/FOSE.2007.27.

[22] François Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas et Beata Winnicka. Sage++ : An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. Dans *The second annual object-oriented numerics conference (OON-SKI*, pages 122–136, 1994.

[23] Michael D. Bond et Kathryn S. McKinley. Probabilistic calling context. Dans *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 97–112, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. URL http://doi.acm.org/10.1145/1297027.1297035.

[24] Derek L. Bruening. Efficient, transparent and comprehensive runtime code manipulation. Rapport technique, Phd Thesis, M.I.T, 2004.

[25] Bryan Buck et Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14:317–329, November 2000. ISSN 1094-3420. URL http://portal.acm.org/citation.cfm?id=1080622.1080630.

[26] M. Burrows, U. Erlingsson, S-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker et W. E. Weihl. Efficient and flexible value sampling. Dans *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 160–167, New York, NY, USA, 2000. ACM. ISBN 1-58113-317-0. URL http://doi.acm.org/10.1145/378993.379236.

[27] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout

et Koen De Bosschere. Using HPM-sampling to drive dynamic compilation. Dans *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 553–568, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. URL http://doi.acm.org/10.1145/1297027.1297068.

[28] Brad Calder, Glenn Reinman et Dean M. Tullsen. Selective value prediction. Dans *Proceedings of the 26th annual international symposium on Computer architecture*, ISCA '99, pages 64–74, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0170-2. URL http://dx.doi.org/10.1145/300979.300985.

[29] Pohua P. Chang, Scott A. Mahlke, William Y. Chen et Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software Practice & Experience*, 22:349–369, May 1992. ISSN 0038-0644. URL http://portal.acm.org/citation.cfm?id=138720.138721.

[30] Clover. Java code coverage and test optimization . URL http://www.atlassian.com/software/clover/.

[31] Cobertura. Code Coverage Tool. URL http://cobertura.sourceforge.net/.

[32] CodeCover. A Free Glass-box Testing Tool. URL http://codecover.org/.

[33] Geoff A. Cohen, Jeffrey S. Chase et David L. Kaminsky. Automatic program transformation with JOIE. Dans *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, Berkeley, CA, USA, 1998. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1268256.1268270.

[34] Apache Commons. BCEL : The Byte Code Engineering Library. URL http://commons.apache.org/bcel/.

[35] OW2 Consortium. ASM Java bytecode manipulation framework. URL http://asm.objectweb.org/.

[36] M. Dmitriev. Selective profiling of Java applications using dynamic bytecode instrumentation. Dans *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 141–150, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8385-0. URL http://portal.acm.org/citation.cfm?id=1153925.1154598.

[37] Bruno Dufour, Barbara G. Ryder et Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. Dans *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 118–128, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-734-6. URL http://doi.acm.org/10.1145/1273463.1273480.

[38] EMMA. A free Java code coverage tool. URL http://emma.sourceforge.net/.

[39] Robert E. Filman et Klaus Havelund. Source-Code Instrumentation and Quantification of Events. Dans *Foundations of Aspect-Oriented Languages*, 2002.

[40] Marc Fisher, Bruno Dufour, Shrutarshi Basu et Barbara G. Ryder. Exploring the impact of context sensitivity on blended analysis. Dans *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-8630-4. URL http://dx.doi.org/10.1109/ICSM.2010.5609695.

[41] Python Software Foundation. Python Programming Language. URL http://www.python.org/.

[42] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley et Vijay Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. Dans *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, Berkeley, CA, USA, 2004.

USENIX Association. URL http://portal.acm.org/citation.cfm?id=1267242.1267254.

[43] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan et Michael Hind. Vertical profiling : understanding the behavior of object-oriented applications. Dans *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 251–269, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. URL http://doi.acm.org/10.1145/1028976.1028998.

[44] Kim Hazelwood et David Grove. Adaptive online context-sensitive inlining. Dans *Proceedings of the international symposium on Code generation and optimization : feedback-directed and runtime optimization*, CGO '03, pages 253–264, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL http://portal.acm.org/citation.cfm?id=776261.776289.

[45] Urs Holzle et David Ungar. Optimizing dynamically-dispatched calls with runtime type feedback. Dans *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. URL http://doi.acm.org/10.1145/178243.178478.

[46] Raymond J. Hookway et Mark A. Herdeg. DIGITAL FX !32 : combining emulation and binary translation. *Digital Technical Journal*, 9:3–12, January 1997. ISSN 0898-901X. URL http://portal.acm.org/citation.cfm?id=268940.268941.

[47] IBM. Jikes Bytecode Toolkit. URL http://alphaworks.ibm.com/tech/jikesbt.

[48] D. Ingalls. *The Execution Time Profile as a Programming Tool ; Design and Optimization of Compilers*. Prentice-Hall, Englewood Cliffs, 1971.

[49] SPECjvm2008 (Java Virtual Machine Benchmark). Last updated : Tue May 06 21 :31 :01 EDT 2008 Copyright 1995 - 2011 Standard Performance Evaluation Corporation. URL http://www.spec.org/jvm2008/index.html.

[50] JBoss. Javassist Java Programming Assistant. URL http://www.jboss.org/javassist/.

[51] JProfiler. URL http://www.ej-technologies.com/.

[52] A. Kishon, C. Consel et P. Hudak. Monitoring semantics : a formal framework for specifying, implementing and reasoning about execution monitors. Dans *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–352, 1991.

[53] M. Kumar. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *Computers, IEEE Transactions*, 37:1088–1098, September 1988. ISSN 0018-9340. URL http://dx.doi.org/10.1109/12.2259.

[54] James R. Larus et Eric Schnarr. EEL : machine-independent executable editing. Dans *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 291–300, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. URL http://doi.acm.org/10.1145/207110.207163.

[55] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington et Allan Snavely. PEBIL : Efficient static binary instrumentation for Linux. Dans *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 175–183.

[56] Ben Liblit, Alex Aiken, Alice X. Zheng et Michael I. Jordan. Bug isolation via remote program sampling. Dans *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 141–154, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. URL http://doi.acm.org/10.1145/781131.781148.

[57] Tim Lindholm et Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 1999. ISBN 0201432943.

[58] Y. A. Liu et G. Gomez. Automatic accurate time-band analysis for high-level languages. volume 1474 de *the ACM SIGPLAN'98 Workshop on Languages, Compilers, and Tools for Embedded Systems, Lectures Notes in Computer Science*, page 3144, New York, 1998. Springer-Verlag.

[59] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi et Kim Hazelwood. Pin : building customized program analysis tools with dynamic instrumentation. Dans *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. URL `http://doi.acm.org/10.1145/1065010.1065034`.

[60] Microsoft. The .NET Framework. URL `http://www.microsoft.com/net/`.

[61] Bernd Mohr, Darryl Brown et Allen D. Malony. TAU : A Portable Parallel Program Analysis Environment for pC++. Dans *Proceedings of the Third Joint International Conference on Vector and Parallel Processing : Parallel Processing*, CONPAR 94 - VAPP VI, pages 29–40, London, UK, 1994. Springer-Verlag. ISBN 3-540-58430-7. URL `http://portal.acm.org/citation.cfm?id=646743.703230`.

[62] Philippe Moret, Walter Binder et Alex Villazon. CCCP : complete calling context profiling in virtual execution environments. Dans *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 151–160, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-327-3. URL `http://doi.acm.org/10.1145/1480945.1480967`.

[63] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth et Peter F. Sweeney. Evaluating the accuracy of Java profilers. Dans *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. URL http://doi.acm.org/10.1145/1806596.1806618.

[64] Nicholas Nethercote et Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. Dans *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. URL http://doi.acm.org/10.1145/1250734.1250746.

[65] P. Pereira, L. Heutte et Y. Lecourtier. Source-to-Source Instrumentation for the Optimization of an Automatic Reading System. *The Journal of Supercomputing*, 18: 89–104, January 2001. ISSN 0920-8542. URL http://portal.acm.org/citation.cfm?id=370994.371027.

[66] David Röthlisberger, Marcel Harry, Alex Villazon, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz et Philippe Moret. Augmenting static source views in IDEs with dynamic metrics. Dans *International Conference on Software Maintenance*, pages 253–262, 2009.

[67] Jikes Research Virtual Machine (RVM). The Jikes RVM Project. URL www.jikesrvm.org.

[68] J. M. Spivey. Fast, accurate call graph profiling. *Software Practice & Experience*, 34:249–264, March 2004. ISSN 0038-0644. URL http://portal.acm.org/citation.cfm?id=991085.991087.

[69] Amitabh Srivastava et Alan Eustace. ATOM : a system for building customized program analysis tools. Dans *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 196–205,

New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. URL http://doi.acm.org/10.1145/178243.178260.

[70] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu et Toshio Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27:732–785, July 2005. ISSN 0164-0925. URL http://doi.acm.org/10.1145/1075382.1075386.

[71] Toshio Suganuma, Toshiaki Yasue et Toshio Nakatani. An Empirical Study of Method In-lining for a Java Just-in-Time Compiler. Dans *Proceedings of the 2nd JavaTM Virtual Machine Research and Technology Symposium*, pages 91–104, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-931971-01-3. URL http://portal.acm.org/citation.cfm?id=648042.744889.

[72] Vijay Sundaresan, Daryl Maier, Pramod Ramarao et Mark Stoodley. Experiences with Multi-threading and Dynamic Class Loading in a Java Just-In-Time Compiler. Dans *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 87–97, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. URL http://dx.doi.org/10.1109/CGO.2006.16.

[73] Mustafa M. Tikir et Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. Dans *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 86–96, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9. URL http://doi.acm.org/10.1145/566172.566186.

[74] Leena Unnikrishnan, Scott D. Stoller et Yanhong A. Liu. Automatic Accurate Stack Space and Heap Space Analysis for High-Level Languages. Rapport technique 538, Computer Science Dept., Indiana University, April 2000.

[75] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pomin-

ville et Vijay Sundaresan. Optimizing Java Bytecode Using the Soot Framework : Is It Feasible ? Dans *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 18–34, London, UK, 2000. Springer-Verlag. ISBN 3-540-67263-X. URL http://portal.acm.org/citation.cfm?id=647476.727758.

[76] Alex Villazon, Walter Binder et Philippe Moret. Aspect weaving in standard Java class libraries. Dans *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, PPPJ '08, pages 159–167, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-223-8. URL http://doi.acm.org/10.1145/1411732.1411754.

[77] Alex Villazon, Walter Binder et Philippe Moret. Flexible calling context reification for aspect-oriented programming. Dans *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 63–74, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-442-3. URL http://doi.acm.org/10.1145/1509239.1509249.

[78] John Whaley. A portable sampling-based profiler for Java virtual machines. Dans *Proceedings of the ACM 2000 conference on Java Grande*, Java '00, pages 78–87, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3. URL http://doi.acm.org/10.1145/337449.337483.

[79] Abe White. Serp. URL http://serp.sourceforge.net/.

[80] Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu et Toshio Nakatani. An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers. Dans *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 148–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2021-9. URL http://portal.acm.org/citation.cfm?id=942806.943848.

[81] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen et Michael D. Smith.

System support for automatic profiling and optimization. Dans *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 15–26, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5. URL http://doi.acm.org/10.1145/268998.266640.

[82] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain et Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. Dans *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 263–271, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. URL http://doi.acm.org/10.1145/1133981.1134012.