Optimisation de code pour application Java haute-performance

Abderrahmane Nassim Halli

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 Août 2006

Présentée par

Abderrahmane Nassim HALLI

Thèse dirigée par **Jean-François MÉHAUT**
et codirigée par **Henri-Pierre CHARLES**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Optimisation de Code pour Application Java Haute-Performance

Thèse soutenue publiquement le **24 octobre 2016**,
devant le jury composé de :

M. William JALBY
Professeur à l'Université de Versailles - St-Quentin-en-Yvelines, Président
M. Gaël THOMAS
Professeur à Télécom SudParis, Rapporteur
M. Denis BARTHOU
Professeur à l'Université de Bordeaux, Rapporteur
M. Sébastien BAYLE
Responsable de l'équipe de développement à Aselta Nanographics, Examinateur
M. Jean-François MÉHAUT
Professeur a l'Université Grenoble Alpes, Directeur de thèse
M. Henri-Pierre CHARLES
Directeur de recherche au CEA LIST, Co-Directeur de thèse
M. Patrick SCHIAVONE
Directeur scientifique à Aselta Nanographics, Invité

Remerciements

Mes remerciements s'adressent tout d'abord à mes directeurs de thèse. Jean-François Méhaut, pour m'avoir offert l'opportunité de préparer cette thèse et Henri-Pierre Charles pour son aide et ses conseils précieux. Merci à tous les deux pour votre encadrement durant ces trois années et demi écoulées.
Ils s'adressent à Sébastien Bayle, pour son soutien et sa confiance, à Julien Nicouleau pour toute l'aide qu'il m'a apporté et à Patrick Schiavone pour l'expérience dont il m'a fait profiter.
Je remercie également Serdar Manakli de m'avoir accordé sa confiance pour mener à bien ce projet ainsi que l'association nationale de la recherche et de la technologie (ANRT) pour sa participation au financement de ce contrat CIFRE.
Merci à Gaël Thomas et Denis Barthou d'avoir rapporter ces travaux. Merci pour leurs retours et pour l'intérêt qu'ils ont témoigné. Un grand merci également à William Jalby pour la présidence du jury de thèse.
Je remercie chaleureusement toute l'équipe d'Aselta Grenoble avec qui j'ai eu le plaisir, et j'ai toujours le plaisir, de partager une grande partie de mon quotidien : Charles, Thomas, Guillaume, Thiago, Vincent, Luc, Pascal, Christophe, Joël, Jean, Mohammed A., Céline, Paolo, Stéphane, Mohammed, Clyde, Alexis, Mathieu, Anne, Alexandre, Bruno, Matthieu et Serguei. Merci pour finir à Brice, Kevin, Naweiluo, Thomas et à tous les membres de l'équipe CORSE.

Résumé

Java est à ce jour l'un des langages, si ce n'est le langage, le plus utilisé toutes catégories de programmation confondues et sa popularité concernant le développement d'applications scientifiques n'est plus à démontrer. Néanmoins son utilisation dans le domaine du Calcul Haute Performance (HPC) reste marginale même si elle s'inscrit au cœur de la stratégie de certaine entreprise comme Aselta Nanographics, éditeur de l'application Inscale pour la modélisation des processus de lithographie par faisceaux d'électron, instigateur et partenaire industriel de cette thèse.

Et pour cause, sa définition haut-niveau et machine-indépendante, reposant sur un environnement d'exécution, parait peu compatible avec le besoin de contrôle bas-niveau nécessaire pour exploiter de manière optimale des architectures de microprocesseurs de plus en plus complexes comme les architectures Intel64 (implémentation Intel de l'architecture x86-64). Cette responsabilité est entièrement déléguée à l'environnement d'exécution, notamment par le biais de la compilation dynamique, chargée de générer du code binaire applicatif à la volée. C'est le cas de la JVM HotSpot, au centre de cette étude, qui s'est imposée comme l'environnement de référence pour l'exécution d'applications Java en production.

Cette thèse propose, dans ce contexte, de répondre à la problématique suivante : comment optimiser les performances de code séquentiel Java plus particulièrement dans un environnement HotSpot/Intel64 ?

Pour tenter d'y répondre, trois axes principaux ont été explorés. Le premier axe est l'analyse des performances du polymorphisme, mécanisme Java haut-niveau omniprésent dans les applications, dans le lequel on tente de mesurer l'impact du polymorphisme sur les performances du code et d'évaluer des alternatives possibles. Le second axe est l'intégration de code natif au sein des applications - afin de bénéficier d'optimisations natives - avec prise en compte du compromis coût d'intégration/qualité du code. Enfin le troisième axe est l'extension du compilateur dynamique pour des méthodes applicatives afin, là encore, de bénéficier d'optimisations natives tout en s'affranchissant du surcout inhérent à l'intégration de code natif.

Ces trois axes couvrent différentes pistes exploitables dans un contexte de production qui doit intégrer certaines contraintes comme le temps de développement ou encore la maintenabilité du code. Ces pistes ont permis d'obtenir des gains de performances significatifs sur des sections de code applicatif qui demeuraient jusqu'alors très critiques.

Sommaire

Chapitre 1

Introduction

1.1 Problématique

Les limitations physiques comme la dissipation thermique causées par la miniaturisation des transistors et l'augmentation de la fréquence de cadencement des composants ont emmenées les concepteurs de micro-processeurs à se focaliser sur l'augmentation du parallélisme de cœur (i.e. l'augmentation du nombre de cœurs sur une puce) afin de maintenir l'évolution de la puissance de calcul prédit depuis le début des années 70 jusqu'à aujourd'hui par la loi de Moore [39]. L'exploitation du parallélisme de cœur s'est ainsi retrouvée au centre des problématiques du HPC.
Dans le même temps, la diminution toujours effective de la finesse de gravure a permis d'intégrer au sein des cœurs de calcul de nouvelles unités dédiées exploitable par de nouveaux jeux d'instruction ainsi que des composants matériels plus sophistiqués pour accélérer l'exécution des instructions parmi lesquels : le moteur d'exécution out-of-order (i.e. dans le désordre), les caches mémoire ou l'unité de prédiction de branchement. La Table 1.1 montre l'évolution (selon le modèle tick-tock d'Intel [153]) des micro-architectures Intel64 et de la finesse de gravure ainsi que l'apparition des différentes extensions du jeu d'instruction x86-64[1].
On peut résumer cette évolution comme étant une augmentation de l'expressivité du langage cible comme, dans les deux cas, les optimisations peuvent être mises en œuvre au niveau du code machine. L'amélioration des performances sur un cœur de calcul (ou optimisation séquentielle) est une autre problématique au centre du HPC, complémentaire du parallélisme, et celle considérée dans cette étude.

L'évolution des architectures s'accompagne parallèlement d'une augmentation de la complexité des applications associée à l'explosion des besoins informatiques [190]. Cette évolution a vu l'émergence de nouveaux paradigmes de programmation ainsi que l'essor

[1] Aussi nommé x64. La dénomination x86 désigne à la fois x86-32 (version 32-bits) et x86-64 (version 64-bits)
[2] AVX3 est un alias pour AVX-512

Lancement	2008	2011	2012	2013	2014	2015	2016
Micro-architecture	Nehalem	Westmere	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake
Finesse de Gravure (nanomètre)	45 nm	32 nm		22 nm		14 nm	
Extensions introduites	SSE4.2		AVX		AVX2, FMA, BMI		AVX3[2]

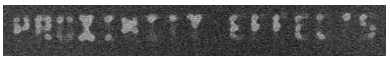
TAB. 1.1: Évolution des micro-architectures Intel64 selon le modèle tick-tock. Chaque génération de processeur est composé de deux cycles : le cycle tock marque le lancement une nouvelle micro-architecture et le cycle tick marque une réduction de la finesse de gravure ou *die-shrink*. Chaque génération s'accompagne d'une extension du jeu d'instruction x86-64 (les die-shrinks peuvent également s'accompagner d'une extension du jeu d'instructions mais généralement minime). L'arrivée du 5 nm est prévue aux alentours de 2020

de nouveaux langages facilitant et augmentant la productivité de la programmation notamment concernant la programmation parallèle. Java incarne cette évolution avec une sémantique simple à prendre en main et le support de nombreuses fonctionnalités haut-niveau comme le polymorphisme.
La tendance des langages de programmation vers l'abstraction et le haut-niveau entre en contradiction avec les besoins de performance rencontrées en HPC qui implique des optimisations bas-niveau et une forte connaissance de l'architecture sous-jacente.
Cet effort d'optimisation est délégué aux environnements d'exécution plus précisément par le biais du compilateur dynamique en charge de compiler et d'optimiser le code applicatif à la volée. Cependant les problèmes d'expressivité du langage et les limitations du compilateur peuvent conduire à un code sous-optimal. Cette étude adresse cette problématique et propose des solutions pour l'optimisation de code Java.

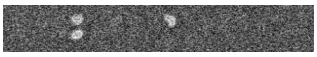
1.2 Contexte de la thèse

Cette thèse, financée par une convention industrielle de formation par la recherche (CIFRE), s'est déroulée en partenariat entre la société Aselta Nanographics et le Laboratoire d'Informatique de Grenoble (LIG).
La partie académique de cette thèse s'est plus précisément déroulée au sein de l'équipe CORSE (Compiler Optimization and Runtime Systems) dont les domaines d'expertises concernent l'optimisation de code à la compilation ainsi que les environnements d'exécution. Ses objectifs de recherches ciblent l'amélioration des performances et la réduction de la consommation énergétique aussi bien pour des systèmes HPC que pour des systèmes embarqués. Les domaines d'application visés sont divers allant de la modélisation numérique au traitement du signal. La thèse s'est par ailleurs déroulée en étroite collaboration avec le laboratoire CEA-LIST.
L'activité principale de la société Aselta Nanographics est l'édition du logiciel Inscale. Ins-

cale est utilisé dans l'industrie des semi-conducteurs et intervient dans le pré-traitement des données utilisées dans les procédés de lithographie électronique.
La lithographie électronique est le procédé d'impression de motifs sur une résine photosensible via des faisceaux d'électrons. Les machines effectuant cette opération le font à partir d'un design d'entrée numérisé contenant les motifs à imprimer appelé *layout*. Il s'agit d'une des étapes les plus critiques de la fabrication des circuits intégrés comme la miniaturisation des transistors voit des phénomènes physiques (les effets de proximité), tel que la diffusion d'électrons, impacter négativement et de manière significative la résolution des motifs en sortie.
Inscale intervient dans la modélisation de ces phénomènes et la simulation des procédés de lithographie électronique afin notamment de corriger les designs d'entrée pour garantir une résolution optimale sur les motifs en sortie. La Figure 1.1 montre l'amélioration de la résolution après une gravure par lithographie grâce au pré-traitement des données par Inscale.



(a) Dimension 80 nm sans traitement



(b) Dimension 40 nm sans traitement

(c) Dimension 80 nm après traitement par Inscale

(d) Dimension 40 nm après traitement par Inscale

FIG. 1.1: Amélioration de la résolution de gravure grâce au traitement d'Inscale pour des finesses de gravure par faisceaux d'électrons de 80 et 40 nanomètres

Le logiciel est écrit en Java et déployé sur des grappes de calcul afin de respecter les délais de traitement requis en production. Sa bonne scalabilité lui permet de réduire ces délais sur d'importantes charges de calcul en augmentant la puissance de calcul.
Compte tenu de la bonne scalabilité du logiciel, l'optimisation de code constituait une piste d'amélioration plus intéressante comme de surcroit son potentiel de gain était difficilement estimable et difficilement exploitable du fait du haut niveau d'abstraction du Java.
Le sujet de thèse s'est ainsi focalisé sur l'optimisation de code Java dans l'objectif de trouver des solutions pour améliorer les performances de code critique au sein d'Inscale. En dehors d'Aselta, la problématique d'optimisation de code se retrouve chez de nombreux éditeurs de logiciel Java. Une étude réalisée en 2015 auprès d'un échantillon de développeurs Java a montré que plus de 50% des efforts d'optimisation concernaient du code inefficace [103].

1.3 Structure de la thèse

La thèse est divisée en 8 chapitres dont, en plus de l'introduction et la conclusion, 1 background, 4 chapitres de contribution et 1 état de l'art :

- Le Chapitre 2 propose un background sur les différents éléments du contexte, à savoir la micro-architecture, l'environnement d'exécution et les noyaux de calcul considérés. Il fixe également la problématique, les motivations et annonce le plan des contributions (cf. Section 2.5) ;
- Les Chapitres 3, 4, 5 et 6 constituent les contributions et sont annoncés plus précisément Section 2.5 Chapitre 2 ;
- Le Chapitre 7 propose un état de l'art sur les techniques d'optimisations de code Java. Il positionne et évalue les techniques investiguées dans cette étude selon une grille de critères précisée dans l'introduction du chapitre ;
- Le Chapitre 8 résume les résultats obtenus ainsi que les conclusions apportées et propose une ouverture sur les points intéressant à investiguer dans la continuité de cette étude.

Chapitre 2

Contexte et positionnement du problème

Ce chapitre fixe le contexte d'étude de la thèse. Quatre éléments indépendants entrent en jeu pour fixer le contexte d'étude. Ces éléments peuvent être classés selon leur niveau d'abstraction par rapport à l'architecture. Chaque section du chapitre s'attache à décrire un de ces éléments dans le cadre de l'étude menée.

1. **Architecture**. L'application Inscale (cf. Section 1.2) cible des systèmes HPC généralistes majoritairement composés de processeurs x86-64. La Section 2.1 décrit ce type d'architecture et en particulier la micro-architecture Sandy Bridge considérée pour les tests de performance. Elle définit également des notions autour de l'optimisation de code ainsi que les mesures de performance utilisées dans cet exercice ;
2. **Environnement d'exécution**[1]. La Section 2.3 propose une description de la machine virtuelle HotSpot, environnement de référence pour l'exécution d'applications Java en production et utilisée pour l'exécution d'Inscale ;
3. **Langage**. La Section 2.2 décrit le langage Java ainsi que ses caractéristiques motivant la problématique d'étude. Ses spécificités ainsi que ses forces et faiblesses pour le HPC sont exposées ;
4. **Algorithmes**. La Section 2.4 décrit les différents noyaux de calcul ou patterns de programmation sujets aux problèmes de performance considérés dans cette étude.

Pour finir, la Section 2.5 formalise le problème traité au regard des différents points exposés et annonce le plan des contributions.

2.1 Architecture et optimisation de code

Comme rappelé en introduction, les architectures évoluent vers plus de parallélisme et d'hétérogénéité. Les études menées concernant l'amélioration des performances des appli-

[1] Ou compilateur dans le cas de langages compilés statiquement

cations s'appuient sur la prise en compte de ces évolutions. On distingue deux modalités d'amélioration des performances, indépendantes et complémentaires, qui sont le parallélisme et l'optimisation de code. Le parallélisme se focalise sur l'exploitation de l'ensemble des cœurs de calcul disponibles à partir du parallélisme exprimé dans la tâche à effectuer[2]. La bonne exploitation du parallélisme repose sur l'amélioration de la scalabilité à savoir la diminution de la criticité des parties séquentielles de l'application. L'optimisation de code, elle, se focalise sur l'amélioration des performances du code critique exécuté sur un cœur de calcul. Cette étude portant sur la langage Java, on s'intéresse à l'optimisation de code qui dépend à la fois du langage et de l'environnement d'exécution. La Section 2.1.2 définit les concepts utilisés autour de l'optimisation de code.

La Figure 2.1 montre la représentation en nombre de cœur des différentes architectures dans la classification Top500 [3] des 500 calculateurs les plus puissants au monde. L'architecture x86-64 (ou x64) reste prédominante comme elle compose plus de 59% des calculateurs même si sa représentativité est en baisse par rapport à Novembre 2015 où elle composait plus de 93% des cœurs de calcul. Cette chute est due à l'arrivée en tête du classement du supercalculateur TaihuLight [52] en Juin 2016 qui a largement bouleversé le classement comme il contient à lui seul plus de 10 millions de cœurs de calcul (sur les 40 millions composant le Top500) contenus dans des processeurs Sunway SW2601 soit une part de plus de 26%[3].

Les expérimentations menées autour d'Inscale ont lieu sur des processeurs de la gamme Intel64. En particulier, les résultats de performance fournis dans cette étude ont été mesurés sur un processeur Intel Core i5-2500 qui implémente la micro-architecture Ivy Bridge (die-shrink de Sandy Bridge comme montré Table 1.1). Intel garantit que les optimisations valides sur cette micro-architecture le sont également sur les générations plus récentes comme Hawell et Broadwell. Malgré l'intégration progressive de ces nouvelles générations de processeurs, la génération Sandy Bridge reste la plus représentée en Juin 2016 en nombre de cœur avec une part d'environ 29%. Cette dernière est décrite plus précisément Section 2.1.1.

[2] La complémentarité signifie que les deux optimisations peuvent être combinées et leur effet multiplié

[3] De ce point de vue, on peut dire que la proportion des différentes architectures dans le Top500 n'est pas pleinement représentative du monde du HPC dans son intégralité

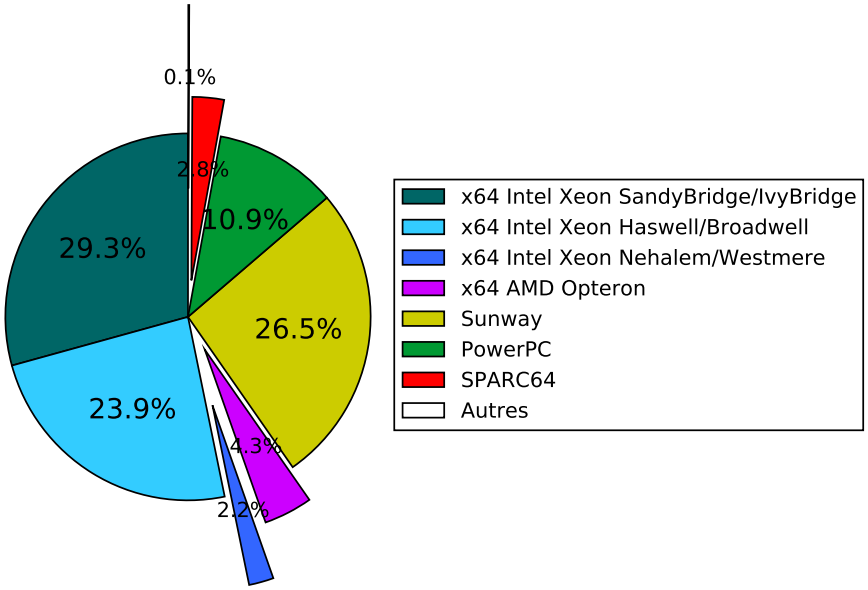


FIG. 2.1: Représentation des différentes architectures en nombre de cœur de calcul sur les 40 millions composant les 500 super-calculateurs les plus puissants au monde en Juin 2016 [3]. Les architectures Intel64 sont regroupées par génération (cf. Table 1.1)

2.1.1 La micro-architecture Sandy Bridge

Sandy Bridge (SB) est une micro-architecture de la famille Intel64 lancée en 2012. SB est gravée en 32 nm et correspond à un cycle de développement *tock* dans le modèle de développement tick-tock d'Intel (cf. Table 1.1). Ivy Bridge (IB) correspond au cycle *tick* à savoir une réduction de la finesse de gravure de SB à 22 nm. Le terme "génération" désigne un cycle complet tick-tock, ainsi la génération SB désigne à la fois SB et IB. La génération SB introduit différentes améliorations pour les performances dont l'une des plus notables est l'intégration d'AVX (Advanced Vector eXtension) pour le calcul vectoriel. Les informations présentées dans cette section sont principalement issues des documents suivants [48, 135].
Une micro-architecture a deux composants : la chaine de traitement (ou pipeline) et la hiérarchie de caches mémoire. La chaine de traitement (ou pipeline) est divisée en 2 parties :

1. Le **front-end** (ou partie avant) dans lequel les instructions sont chargées puis décodées avant d'être consommées par le back-end (cf. Section 2.1.1.1) ;

2. Le **back-end** (ou partie arrière) contenant l'ordonnanceur et les unités d'exécution (cf. Section 2.1.1.2).

La **hiérarchie de caches** permet d'accélérer les transferts entre la RAM et les registres situés. La hiérarchie de cache de SB est décrite Section 2.1.1.3. La Figure 2.2 montre une vue d'ensemble de la micro-architecture décrite plus en détail dans les section qui suivent.

2.1.1.1 Front-end

L'objectif du front-end est d'alimenter back-end en micro-instructions (µops) avec un débit suffisant pour ne pas qu'il y ait de pénuries. Il a notamment deux fonctions : charger les instructions depuis la mémoire (*fetch*) puis les décoder (*decode*).

Prédiction de branchement (PDB) Le chargement se fait depuis le cache d'instruction L1i. Cette étape utilise la PDB, effectuée par l'unité de prédiction de branchement qui détermine à quelle adresse charger les instructions en cas de branchement conditionnel. La PDB est indispensable pour maintenir un débit d'exécution optimal, en particulier sur des micro-architectures avec un long pipeline comme SB. Un branchement peut être conditionnel ou inconditionnel. Il peut également être direct (i.e. déterminé et invariant) ou indirect (i.e. la cible est chargée en mémoire ou dans un registre). En combinant ces deux caractéristiques, il existe donc 4 types de branchement. La PDB prédit ces deux informations à savoir est-ce que le branchement est pris ou non et quelle est sa cible.
SB a renversé la tendance qui allait vers la complexification de la prédiction de branchement en supprimant l'unité de prédiction spécialisée pour les boucles. Le design a dû être remanié avec l'intégration du µop-cache décrit plus loin. Un défaut de prédiction entraîne une latence d'environ 15 cycles.
Pour prédire si un branchement est pris ou non, SB utilise une prédiction adaptative à 2 niveaux avec un historique global sur 32-bits. Un historique global est commun à tous les branchements et permet de prédire des patterns montrant des corrélations entre branchements. L'historique est un registre maintenant l'état réel des 32 derniers branchements exécutés : le i-ème bit de l'historique vaut 1 (resp. 0) si le i-ème branchement exécuté a été (resp. n'a pas été) pris. La valeur de l'historique permet de calculer un index dans la table PHT (*Pattern History Table*) contenant des compteurs (qui sont plus précisément des automates finis) dont la valeur sert à la prédiction. Les compteurs peuvent être combinés afin d'étendre l'ensemble des distributions prédictibles. Ce type de PDB permet de prédire parfaitement (après un délai de quelques période pour la transition des automates) toute distribution périodique de période n+1 ou moins où n est le nombre de bits de l'historique (soit 32 pour SB). Une distribution périodique de période p strictement comprise entre n+1 et 2^n+1 peut être prédite parfaitement si les p plages contiguës de n-bits recouvrant une période sont distinctes.
Les cibles des branchements indirects sont prédites en utilisant le même type de prédiction que pour les branchements conditionnels. Les cibles possibles sont contenues dans la table BTB (*Branch Target Buffer*) et l'historique contient les entrées successives dans la BTB. Le nombre de branchement contenus dans l'historique est moins important que pour prédire si un branchement conditionnel est pris ou non comme dans ce cas plus d'un bit (probablement 6 ou 7) sont nécessaires pour stocker le nombre de cibles possibles. Ce nombre n'est pas connu pour SB, mais il est probablement supérieur à 36 (64 ou 128) qui est la valeur pour la micro-architecture Nehalem qui lui précède.

La PDB dans SB est également limitée par la densité de branchement. Elle peut gérée au maximum 4 instructions `CALL` par mots de 16 bytes. Les branchements conditionnels sont moins bien prédits si leur densité est supérieure à 3 par mots de 16 bytes.
Les instructions `RET` (retour de fonction) sont gérée de manière spécifique. Les adresses de retour sont empilées dans une pile (appelée *stack-buffer*) à chaque exécution d'un `CALL`, la cible d'un `RET` est ensuite chargée depuis la tête de pile. Dans SB, le stack-buffer contient 16 entrées.

Décodage Le décodage a pour fonction de convertir les instructions[4] en micro-instructions (µops). L'usage des µops a deux avantages : d'une part il simplifie le décodage au niveau du back-end[5] et d'autre part il augmente l'efficacité du pipeline et de l'exécution out-of-order. Ces µops sont sauvées dans un cache spécifique appelé µop-cache qui sert ensuite de source au back-end.
Comme illustré Figure 2.2, le décodage est précédé d'un pré-décodage. Son rôle est de fragmenter le flux binaire entrant sous forme d'instructions[6], il repère également certains types d'instruction comme les branchements. Les instructions pré-décodées sont ensuite accumulées dans une une file appelée *instruction-queue* dont la capacité est limitée à 64 bytes (ou 18 instructions). L'instruction-queue permet notamment de couvrir les délais de prédiction de branchement. Le débit du pré-décodeur est limité à 16 bytes/cycle de binaire fragmenté (ce qui équivaut à 6 instructions/cycle au maximum). Les instructions à cheval sur 16 bytes requièrent 1 cycle à elles-seules pour être pré-décodées.
SB contient ensuite 4 décodeurs. Le premier décode les instructions complexes (i.e. composées d'au plus 4 µops). Les 3 autres décodent les instructions simples (i.e. composées d'une µop). Les instructions composées de plus de 5 µops chargent du micro-code et requièrent d'avantage de cycle. Le débit du décodage est limité à 4 µops/cycle indépendamment des combinaisons d'instructions décodées. Afin d'augmenter ce débit deux techniques sont utilisées :

- La macro-fusion : elle consiste à fusionner une instruction de type comparaison entière ou comparaison logique avec un branchement conditionnel lui succédant (typiquement les épilogues de boucle) au sein d'une unique µop ce qui permet d'élever dans certains cas le débit à 5 µops/cycle ;
- La micro-fusion : elle consiste à fusionner un couple de µops au sein d'une unique µop pour certaines combinaisons de µops utilisant deux ports d'exécution distincts et impliquant une µop arithmétique et un transfert mémoire. Il s'agit plus précisément des calculs arithmétiques d'adresse suivis d'une écriture mémoire à cette adresse (e.g.

[4]Par opposition aux µops les instructions classiques peuvent être désignées par macro-instruction
[5]Ces µops s'apparentent aux instructions que l'on retrouve sur les architectures RISC
[6]Contrairement à certaines architectures comme SPARC où les instructions sont encodées sur taille fixe (4 bytes), les instructions x86 ne sont pas encodées sur taille fixe, le début et la fin de chaque instruction doit donc être déterminé

`MOV %rax,(%rbx,%rdx,8)`)[7,8] et des lectures mémoire suivies d'une µop arithmétique lisant la valeur chargée (e.g. `MUL (%rbx),%rax`). La micro-fusion permet en théorie de doubler le débit du front-end dans les meilleurs cas.

SB améliore le débit du front-end par rapport à son prédécesseur Nehalem grâce à l'intégration de deux nouveaux composants :

- Un cache mémoire appelé *µop-cache* pouvant contenir jusqu'à 1536[9] µops. Ce cache sert d'alternative aux décodeurs pour fournir les µops au back-end. Contrairement aux décodeurs ce derniers a un débit non pas limité à 16 bytes/cycle et 4 µops/cycle mais à 32 bytes/cycle et 4 µops/cycle ;
- Une file (*µop-queue*) limitée à 256 bytes ou 28 instructions (512 bytes et 56 dans le cas de IB) alimentée par les décodeurs ou le µop-cache servant d'intermédiaire avec le back-end mais également de tampon de boucle (*loop-buffer*). Pour cela, la file est associée à un composant appelé *Loop Stream Detector* qui maintient les blocs redondants dans la file et évite ainsi le passage par les décodeurs ou le µop-cache.

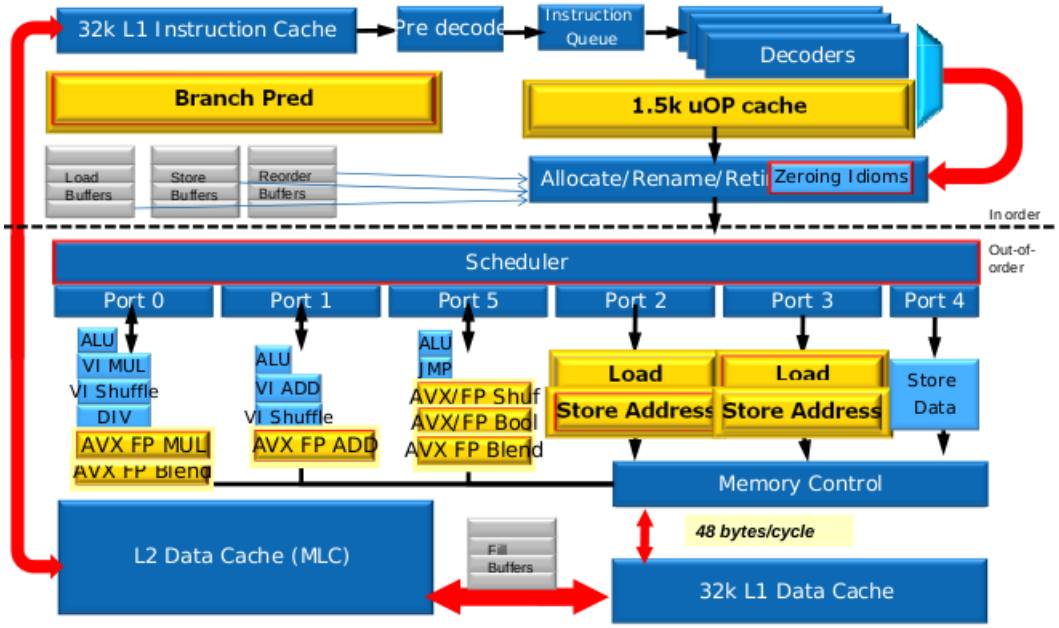


FIG. 2.2: Diagramme simplifié de la micro-architecture Sandy Bridge. Le front-end charge les instructions depuis le cache d'instruction L1, les convertit après décodage en µops puis les stocke dans le µop-cache. Le back-end, alimenté par les décodeurs ou le µop-cache, exécute ensuite les µops dans le désordre. Les composants en jaune soulignent les points d'amélioration par rapport à la micro-architecture précédente Nehalem à savoir la prédiction de branchement, le µop-cache et l'extension AVX. *Crédit image* : [177]

[7] L'ordre des opérandes est : `source, destination`

[8] Notons que les lectures mémoire précédées d'un calcul d'adresse sont prévues pour être exécutées sur une seule unité et correspondent donc à une seule µop

[9] Il s'agit d'un cache de 32 ensemble 8-associatifs contenant jusqu'à 6 µops par ligne

2.1.1.2 Unités d'exécution

Exécution out-of-order Les µops sont chargées depuis le front-end par une unité dont le rôle est d'organiser l'exécution out-of-order. Elle effectue le renommage de registres afin de supprimer les fausses dépendances[10], l'allocation des ressources nécessaires à l'exécution des µops, et la gestion du *Re-Order Buffer* (ROB).
Elle effectue également l'exécution des instructions sans vrai dépendances comme `NOP` (*no-operation*) ou les instructions du type *zeroing* (i.e. qui produisent toujours la valeur zéro comme `XOR %eax,%eax`). Les branchements vers des adresses relatives au compteur ordinal sont également exécutés à cette étape. Sur IB, les instructions de type `MOV` d'un registre à un autre peuvent également être exécutées à cette étape par modification de l'affectation des registres architecturaux (technique est connue sous le nom *0-latency register moves*).
Le ROB est une file circulaire contenant 168 entrées respectant l'ordre séquentiel des µops. Une entrée contient une µop, son statut et ses opérandes. Une µop peut avoir 3 statuts : (i) en attente, (ii) prête à être exécutée ou (iii) exécutée. Si au moins un de ses opérandes d'entrée est indisponible son statut est en attente de la valeur produite. Si ses opérandes d'entrée sont tous disponibles (depuis les registres physiques ou d'autres entrées du ROB) son statut est prête à être exécutée, la µop est alors envoyée dans la station de réservation (RS) pouvant contenir 54 µops. L'ordonnanceur (*scheduler* Figure 2.2) dispatche les µops depuis la RS vers le port d'exécution approprié. Une fois la µop exécutée (statut exécutée), le résultat est sauvé dans l'entrée du ROB associée et les µops en attente de cette valeur sont notifiées et prennent le statut prêtes à être exécutées. Lorsqu'une µop en tête du ROB a le statut exécutée, cette dernière est retirée de la file (phase de *retierement*). La valeur produite peut être sauvée dans le registre correspondant si ce dernier n'a pas été écrasé par une valeur plus récente, auquel cas le registre physique peut être libéré.
Les écritures mémoire sont également effectuées dans la phase de retirement pour maintenir une vision ordonnée de l'exécution. Le modèle mémoire autorise les lectures mémoire d'être réordonnées devant les écritures s'il n'y pas de dépendance entre elles. La technique du *store forwarding* permet, lorsqu'une lecture mémoire dépend d'une écriture mémoire antérieure, d'être exécutée avant que l'écriture ne soit retirée du ROB en maintenant la valeur dans un buffer spécifique.
Chaque port d'exécution peut être alimenté à raison de 1 µop/cycle. Le débit d'exécution de chaque port dépend ensuite de la nature des instructions exécutées. Il est généralement proche de 1 µop/cycle considérant la multitude d'unité d'exécution dans chaque port et le pipelining de ces unités. Les ports 0, 1 et 5 contiennent les unités d'exécution, les ports 2 et 3 les unités de lecture mémoire et de calcul d'adresse enfin le port 5 contient l'unité d'écritures mémoire.

[10] Par opposition aux vrais dépendances (RaW), les fausses dépendances désignent à la fois les anti-dépendances (WaR) et les dépendances de sortie (WaW)

Extension AVX Par rapport à la génération précédente Nehalem, SB intègre l'extension AVX. Les unités associées sont visibles en jaune Figure 2.2. Celle-ci permet l'exécution d'instructions flottantes SIMD (*Single Instruction Multiple Data*) ou instructions vectoriels. Les instructions SIMD ont pour opérandes des vecteurs ou registres vectoriels dont chaque composante scalaire est un opérande indépendant. AVX expose des registres vectoriels de 256 bits nommés YMM qui permettent donc d'effectuer une opération flottante en 1 cycle sur 4 opérandes scalaires en double-précision et 8 en simple-précision.
La vectorisation est l'étape de micro-programmation consistant à émettre de telles instructions dans le code. Lorsqu'elle est effectuée par un compilateur on parle d'auto-vectorisation. Comme SB contient 2 unités d'exécution vectorielles en parallèle situées sur les ports 0 (multiplication) et 1 (addition), son débit calculatoire maximal théorique, calculé en double précision, est de 8 Flops/cycle (cf. Section 2.1.2.3).

2.1.1.3 Hiérarchie mémoire

Le rôle des caches mémoire est de couvrir les latences d'accès à la RAM en servant de mémoire intermédiaire vers les registres. On parle de hiérarchie comme les caches sont organisés par capacité et latence sachant que, plus la capacité est faible, plus la technologie mémoire utilisée permet des accès rapides. La Figure 2.3 montre la hiérarchie mémoire du processeur utilisé pour les expérimentations. Les caches L1 et L2 sont intégrés au sein d'un cœur SB, leur taille est donc invariante par modèle de processeur SB. Le cache L3 est lui partagé par tous les cœurs et n'est donc pas partie intégrante d'un cœur SB. Les caches L2 et L3 sont unifiés (i.e. les données et les instructions sont mixées) contrairement au cache L1 qui est séparé en deux caches : L1d pour les données et L1i pour les instructions. L'utilisation des caches exploitent le principe de localité dont le prédicat est que des accès mémoires proches temporellement sont localisés géographiquement.

Tous les caches de SB sont de type *write-back* c'est à dire que les modifications d'une ligne de cache sont mises-à-jour au niveau de mémoire inférieur en cas de remplacement (ou éviction) de la ligne de cache. La politique d'éviction utilisée est nommée NRU (*Not-Recently Used*) qui est une variante de LRU (*Least-Recently Used*) plus légère à implémenter. Les caches L1 et L2 utilisent des algorithmes de prefetch avancés afin de prédire quelles zones mémoire vont être accédées et couvrir ainsi les latences d'accès à la RAM.

Les transferts depuis L1d vers les registres se font par mots alignés[11] de 16 bytes. L'accès à des mots mémoire à cheval sur deux lignes de cache entraîne une pénalité importante. L'alignement des données sur 16 bytes permet de bénéficier d'une bande passante optimale dans la plupart des cas. Les ports 2 et 3 peuvent lire depuis L1d avec un débit de 16 bytes/cycle alors que le port 4 peut écrire avec un débit de 16 bytes/cycle. Les 3 ports

[11] Un mot aligné est un mot aligné sur sa taille. Ainsi, une ligne de cache est un mot aligné de 64 bytes et une page, un mot aligné de 4 Kbytes

peuvent fonctionner de manière concurrente offrant un débit mémoire maximal théorique de 48 bytes/cycle. En pratique il est difficile d'atteindre cette valeur considérant le calcul des adresses pour les écritures effectuées sur les ports 2 ou 3 plafonnant le débit maximal à 32 bytes/cycle. Les lignes de cache de 64 bytes sont organisées en 4 banques de 16 bytes. Il n'est pas possible d'effectuer deux lectures dans la même banque dans le même cycle si les lignes de caches sont dans deux ensembles différents (conflit de banque). Les lectures mémoire sont dépendantes des écritures mémoire à des adresses égales modulo 4K, et ne peuvent pas être exécutée simultanément. Ce phénomène est connu sous le nom de 4K-aliasing et provient du fait que seul les 12 premiers bits des adresses sont utilisés pour établir les dépendances mémoire. Le cache L1 est 8-associatifs et indexé virtuellement pour éviter les délais de pagination.

Le cache L2 sert d'intermédiaire entre L3 et L1 (il n'est ainsi ni exclusif ou inclusif vis-à-vis de L1). Tout comme L1 ce dernier est 8-associatif. Il est indexé, tout comme L3, par les adresses physiques, son accès est donc précédé par la pagination. Les accès depuis L2 se font par mots alignés de 32 bytes avec un débit théorique de 32 bytes/cycle. Ainsi l'alignement des donnée sur 32 bytes permet de bénéficier d'un débit optimal. Néanmoins l'alignement sur 32 bytes n'a pas d'impact significatif sur les performances lorsque les données sont alignées sur 16 bytes et que les transferts se font majoritairement dans L1d.

Le cache L3 est le dernier niveau de cache relié à la RAM par un contrôleur mémoire. Les transferts depuis la RAM se font par mots alignés de 64 bytes (les lignes de cache). Sa taille dépend du nombre de cœurs et son associativité varie également selon le modèle de processeur (elle vaut 12 dans le modèle considéré). Ce dernier est découpé en *slices* (tranches) distribués à raison de 1 par cœur. Chaque cœur accède à son slice par mots alignés de 32 bytes avec un débit théorique de 32 bytes/cycle. Les accès aux slices distant se font par l'intermédiaire d'une file circulaire bidirectionnelle, la latence d'accès dépendant de la distance au slice accédé.

Pour accélérer la pagination, un cœur SB possède 2 niveaux de cache TLB (*Translation Look-aside Buffer*). Le cache L1-TLB est 4-associatif et peut indexer 64 pages de 4 Kbytes. Le cache L2-TLB est également 4-associatif et peut indexer 512 pages de 4 Kbytes.

2.1.2 Optimisation de code

L'optimisation de code est l'exercice consistant à, étant donné un bloc de code, fournir par diverses techniques un bloc de code de sémantique équivalente mais ayant de meilleures performances. La métrique de performance peut varier selon les contextes, elle peut être la consommation énergétique, la consommation mémoire ou bien la durée d'exécution.
La durée d'exécution est la métrique considérée dans cette étude. Elle est généralement la métrique la plus regardée concernant les systèmes HPC. La consommation énergétique

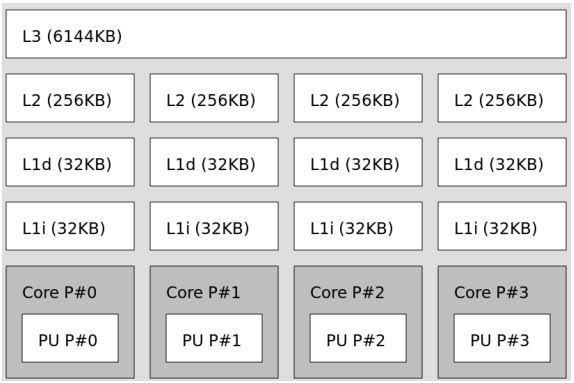


FIG. 2.3: Topologie d'un processeur Intel Core i5-2500 (Ivy Bridge) utilisé pour les expérimentations

est également au centre des problématiques actuelles considérant la forte hausse de l'empreinte énergétique des systèmes informatiques à l'échelle planétaire. La réduction de la durée d'exécution peut induire directement une réduction de la consommation énergétique en exploitant des unités de calcul avec un meilleur rendement énergétique comme les unités de calcul vectoriel.
L'optimisation de code est généralement une phase délicate en programmation informatique d'une part par l'investissement en temps qu'elle demande mais également par les connaissances sur l'architecture sous-jacente qu'elle nécessite pour être bénéfique. Il est ainsi admis que l'effort d'optimisation doit survenir sur un code fonctionnellement figé et après l'avoir précisément identifier comme critique à l'aide de profileurs. Différentes études ont souligné le piège que constituent les optimisations de code prématurées [76].

2.1.2.1 Notion de granularité et criticité

Granularité La granularité est une mesure de la quantité d'instructions. La granularité revêt deux aspects : un aspect statique et un aspect dynamique. La granularité statique correspond au nombre d'instructions dans un bloc de code. La granularité dynamique correspond au nombre d'instructions exécutées entre l'entrée dans le bloc et la sortie du bloc, cette dernière peut donc être variable à chaque exécution du bloc. La granularité dynamique est équivalente à la granularité statique lorsque le bloc ne contient pas de boucle. Dans le reste de l'étude, sauf précision, la granularité désignera la granularité dynamique. Dans le contexte Java on parle ainsi d'une méthode de faible (resp. forte) granularité pour désigner une méthode dont sa forme binaire est de faible (resp. forte) granularité. La granularité statique a une importance majeure concernant le choix des techniques d'optimisations dont l'effort de mise en œuvre dépend de la quantité de code. On pense particulièrement à la programmation assembleur qui est une solution envisageable que pour des blocs de code de faible granularité statique. La granularité dynamique a elle une importance capitale au niveau du coût d'interfaçage utilisée pour relier le code concerné au reste de l'application.

Criticité Pour une instance d'application donnée, on peut affecter à tout bloc de code une criticité (ou température) qui correspond à sa contribution à la durée d'exécution globale de l'instance d'application. La criticité est, à l'instar de la granularité dynamique, une mesure dynamique qui dépend donc du type d'instance d'application exécuté.
Granularité statique et criticité sont reliées par la règle des 80-20 aussi connue comme le principe de Pareto. Cette dernière stipule que 80% du temps d'exécution est consacré à l'exécution de 20% du code total exécuté.
Les efforts d'optimisation se basent sur ce principe en réduisant au maximum la granularité afin de maximiser le ratio criticité/granularité[12] tout en conservant assez de contexte pour avoir une marge de manœuvre conséquente au niveau des optimisations. Les unités d'optimisation sont le plus souvent les fonctions (ou les méthodes dans le contexte Java) qui offrent assez de contexte et limitent les interférences entre optimisations et conception de l'application mais avant les corps de boucle qui constituent la grande partie du code critique. La règle des 80-20 sert par ailleurs de postulat justifiant l'efficacité de la compilation dynamique où seule une partie de l'application peut être compilée pour garantir des bonnes performances (cf. Section 2.3.2.1).
Les méthodes ou fonctions concentrant la plus grande partie du temps d'exécution sont qualifiées de points chauds d'où la notion de température pour exprimer la criticité.

2.1.2.2 Mise en œuvre

Algorithmes et implémentations On peut distinguer deux niveaux d'optimisation qui sont les optimisations algorithmiques et les optimisations d'implémentation.
Théoriquement, les optimisations algorithmiques sont indépendantes du langage de mise en œuvre et précèdent son implémentation. Celles-ci peuvent néanmoins être basées sur des considérations matérielles, comme l'exploitation du principe de localité dans le cas des algorithmes dits cache-oblivious [32].
Les optimisations d'implémentation sont celles considérées dans cette thèse. Elles portent sur l'écriture de l'algorithme dans le langage donné qui peut être fait de différentes manières. Ces optimisations peuvent être basées sur des critères de performance bas-niveaux (comme le déroulage de boucle, l'augmentation du parallélisme d'instruction...) mais également sur des considérations propres au langage. C'est le cas, concernant le langage Java, des optimisations qualifiées de JIT-Friendly (i.e. basée sur la prise en compte des optimisations du compilateur dynamique) explorées dans cette thèse.

Mise en œuvre matérielle et logicielle La mise en œuvre des optimisations peut être matérielle ou logicielle.
Les optimisations matérielles sont celles qui accélèrent l'exécution du code mais qui ne sont pas contrôlées explicitement au niveau du code binaire. Les optimisations matérielles

[12] D'après le théorème de la moyenne on convergerait jusqu'à une instruction

regardées dans cette étude sont la prédiction de branchement, les caches mémoire[13] et l'exécution out-of-order. Leur efficacité varient en fonction du modèle du processeur.
Les optimisations logicielles sont celles exprimées au niveau de la sémantique du code et qui affectent donc le code binaire exécuté comme la vectorisation ou bien l'allocation de registre.
L'exploitation des optimisations matérielles peut nécessiter une mise en œuvre logicielle rendant ambiguë la frontière entre optimisations matérielles et logicielles. Par exemple l'expression du parallélisme d'instruction au bloc niveau d'un bloc de code permet d'exploiter l'exécution out-of-order. D'autre techniques d'optimisation logicielles reposent sur une fine connaissance du fonctionnement des caches ou de la prédiction de branchement. Néanmoins, comme ces optimisations affectent le code exécuté, on parle d'optimisations logicielles.

Compilateur Les optimisations logicielles sont le plus souvent mises en œuvre par les compilateurs qui tentent de maximiser la performance en utilisant des techniques avancées combinées à des informations plus ou moins précises sur l'architecture ciblée. Le compilateur transforme un code exprimé dans un langage source de haut niveau vers le langage machine. De ce point de vue la qualité du code dépend de deux paramètres qui sont l'expressivité du langage source et la qualité du compilateur. Elle dépend également d'un troisième paramètre qui est le degré de spécialisation du code.
Les compilateurs ne garantissent pas néanmoins de produire un code optimal, d'où la nécessité parfois de recourir à des techniques d'optimisations manuelles comme la programmation assembleur. Dans le cas du compilateur dynamique utilisé une des limitations majeures rencontrées concerne la vectorisation du code.

Spécialisation du code Une autre dimension fondamentale dans l'optimisation de code est la spécialisation. La spécialisation consiste à exploiter des informations concernant les données d'entrée afin d'adapter le code pour qu'il soit plus performant. La règle étant que plus un code est spécialisé, plus il est performant. On peut distinguer différents degrés de spécialisation :

0. *Pas de spécialisation.* Il s'agit généralement de code compilé statiquement pour lequel le compilateur statique ne dispose d'aucune information ;
1. *Spécialisation aux constantes dynamiques.* Les constantes dynamiques sont les variables constantes par instance d'application i.e. fixées au démarrage de l'application. En Java il s'agit des variables ayant les attributs `static final`. Ces dernières sont des constantes de compilation du point de vue d'un compilateur dynamique mais des variables du point de vue d'un compilateur statique ;

[13] Dans le cas des caches mémoire il peut y avoir un contrôle au niveau du code avec l'utilisation de *prefecth* explicite. L'utilisation explicite des caches n'est cependant pas considérée dans cette étude

2. *Spécialisation à un site d'appel.* La spécialisation à un site d'appel est typiquement celle mise en œuvre lors de l'inlining par un compilateur dynamique ou bien statique ;
3. *Spécialisation au contexte d'exécution courant.* Il s'agit de la spécialisation la plus forte comme toutes les informations du contexte courant peuvent être exploitées. Ce type de spécialisation est mis en œuvre par les générateurs dynamiques de code mais également par les compilateurs dynamiques.

En pratique chaque degré de spécialisation s'accompagne d'un coût de mise en œuvre rendant sont efficacité dépendante de différents paramètres dont la criticité et la granularité du code en bénéficiant mais également du potentiel de gain par spécialisation.
La spécialisation peut également être mise en œuvre au niveau de la conception des applications. Elle s'oppose dans ce cas au concept de généricité du code qui permet de limiter les efforts de développement mais souvent au détriment des performances. On parle du compromis généricité/spécialisation. La généricité repose en Java sur des mécanismes pris en charge par le langage tels que le polymorphisme (cf. Section 2.2.4) ou bien les types génériques (cf. Section 2.2.3.3).

2.1.2.3 Métriques de performance

Différentes métriques sont utilisées pour quantifier les performances du code. Celles utilisées expriment un débit d'opérations où une opération peut désigner une simple instruction, un type d'instruction ou encore un bloc d'instruction de granularité plus ou moins forte.

Flop/s La principale métrique utilisée concernant les routines numériques effectuant des opérations flottantes est le Flop/s[14] (FLoating point Operation Per Second) qui exprime le débit d'exécution des instructions arithmétiques flottantes de base (i.e. l'addition et la multiplication) contenues dans le bloc de code testé. Cette métrique est utilisée Chapitre 5 pour quantifier les performances de micro-routines calculatoires. Le nombre d'opérations flottantes peut, pour du code régulier[15], être exprimé statiquement par une équation qui dépend de facteurs d'échelle (comme la taille des données). Les Flops/s s'obtiennent alors en divisant ce nombre par la durée d'exécution de la routine. D'autres métriques plus générales existent comme l'IPS (Instruction Per Second) qui mesure le nombre d'instructions exécutées par seconde. Néanmoins cette métrique est plus difficile à exploiter pour comparer les performances sur architectures de type CISC (e.g. x86) et de type RISC (e.g. SPARC, PowerPC)[16] considérant la forte variation de la densité d'instruction entre les

[14] L'acronyme FLOPS est également utilisé
[15] On entend par code régulier, un code dont le nombre d'opérations est calculable statiquement
[16] Les architectures CISC (*Complex Instruction-Set Computer*) possèdent un jeu d'instructions contenant des instructions complexes par rapport aux architectures RISC (*Reduced Instruction-Set Computer*) qui se limitent aux instructions de base. Pour une même sémantique, un code RISC contiendra donc plus d'instructions qu'un code CISC, on dit que sa densité d'instructions est plus élevée. Concernant les performances, les architectures CISC demandent plus d'effort au matériel alors que les architectures RISC

deux. Elle nécessite de plus l'accès à des compteurs matériels pour recueillir le nombre d'instructions exécutées.

Intensité arithmétique À chaque bloc de code ou routine calculatoire, on peut affecter une intensité arithmétique qui est le rapport entre le nombre d'opérations flottantes effectuées et la quantité de donnée lue et écrite en mémoire. Plus précisément, étant donné un bloc de code on peut écrire l'équation :

$$F = AI \times M \tag{2.1}$$

où F est le nombre d'opérations flottantes effectuées (en Flops), M est la quantité de byte lue ou écrite en mémoire (en Bytes) et AI est l'intensité arithmétique (en Flops/Byte).
L'intensité arithmétique peut être constante ou bien variable. Elle est utilisée pour déterminer quel composant matériel est le principal goulot d'étranglement (i.e. celui qui freine les performances) entre les unités d'exécution flottante et l'unité de gestion mémoire [185]. Si le goulot correspond aux unités d'exécution flottante (resp. à la bande passante mémoire) la routine est qualifiée de *CPU-bound* (resp. *memory-bound*).
Pour déterminer la nature de la routine, on considère l'intensité arithmétique à l'équilibre AI_{eq} qui s'écrit :

$$AI_{eq} = \Pi/\beta \tag{2.2}$$

Où Π est la performance crête du processeur en Flops/s et β est la bande passante crête en Bytes/s. Si pour une routine on a $AI \simeq AI_eq$ alors la routine n'a pas de goulot d'étranglement significatif entre les accès mémoires et les calculs flottants. Si $AI >> AI_{eq}$ (resp. $AI << AI_{eq}$) alors la routine est CPU-bound (resp. memory-bound).
Comme exposé Section 2.1.1, sur le processeur utilisé pour les expérimentations Π vaut 8 Flops/cycle (soit environ 29,6 GFlops/s) et β vaut 48 Bytes/cycle (soit environ 177 GBytes/s). AI_{eq} vaut ainsi 1/6 Flops/Byte.
Cette information est utilisée Chapitre 5 pour déterminer la nature des routines considérées et comprendre le comportement des performances lorsque la bande passante mémoire diminue du fait de l'augmentation de la taille des données.

Op/s Les autres métriques utilisées dans l'étude sont exprimées en Op/s (Opération par seconde) et quantifient le débit d'une opération particulière. Dans le contexte Java, une opération est un bloc d'instruction associé à un bytecode, un bloc de bytecode ou encore une méthode. Une opération peut être régulière, auquel cas son chemin d'exécution ne dépend pas des données d'entrée ou bien irrégulière dans le cas contraire. Les performances des opérations irrégulières dépendent de la prédiction de branchement. C'est par exemple le cas du bytecode `invokevirtual` considéré dans l'étude sur le polymorphisme (cf. Chap. 4) ou bien de code contenant des branchements conditionnels (cf. Section 6.2.1 Chap. 6).

demandent plus d'effort au compilateur

Latence et débit réciproque La performance d'une instruction machine est définie par deux métriques qui sont son débit réciproque et sa latence. Le débit réciproque est la latence séparant l'entrée dans l'unité d'exécution de deux instructions similaires mais indépendantes (i.e. l'inverse du débit) alors que la latence correspond à la durée d'exécution globale de l'entrée à la sortie du pipeline. Contrairement à la latence, le débit réciproque prend donc en compte l'exécution out-of-order et le pipelining. Ces deux métriques sont entre autre utilisées par les compilateurs afin de générer un code optimum.
Ces métriques peuvent être étendues pour un bloc de code. Dans les expérimentations menées, les mesures de débit pour des opérations de faible granularité sont privilégiées afin de garantir une bonne exactitude (cf. Chap. 3). Le débit réciproque est obtenu comme l'inverse du débit. Pour des opérations de forte granularité le débit réciproque et la latence convergent (la fenêtre d'exécution out-of-order devenant négligeable devant la quantité de code exécuté). Pour des opérations de faible granularité, cependant, la valeur du débit réciproque mesurée dépend fortement du contexte d'utilisation et est difficile à extrapoler.
Les mesures sont effectuées en utilisant des timers. Ces derniers sont collectés avant et après le bloc de code testé. Sur x86 la collecte est gérée par l'instruction `RDTSC` (*ReaD Time Stamp Counter*) qui charge les registres `edx:eax` avec le nombre de cycle d'horloge écoulés depuis la dernière remise à zéro du processeur (contenu dans le registre TSC). Elle est accompagnée de barrières d'out-of-order (typiquement `CPUID`) pour éviter que des instructions extérieures au bloc testé soient prises en compte ou que des instructions internes ne le soient pas.

2.2 La langage Java

Java est toujours en 2016 le langage le plus utilisé tout domaine de programmation confondu y compris pour le développement d'applications scientifiques [175]. C'est un langage généraliste, concurrent, orienté objet avec une syntaxe proche du C/C++ mais évitant des aspects rendant la programmation complexe et non sécurisée.
Sa popularité est due à la fois à sa simplicité d'usage mais également à la richesse de son écosystème. Sa simplicité repose essentiellement sur l'utilisation d'un environnement d'exécution, la JVM (Java Virtual Machine), qui est en charge d'exécuter l'application sur les différentes plateformes supportées. La JVM permet aux développeurs de s'affranchir, entre autre, des contraintes de portabilité et de gestion mémoire comme précisé Section 2.2.2.
La dénomination Java est néanmoins plus large qu'un simple langage puisqu'elle fait généralement référence à toute la plateforme Java. Celle-ci est décrite brièvement Section 2.2.1. Java souffre néanmoins de limitations pour le calcul haute-performance notamment concernant le sujet traité à savoir l'optimisation de code. Celles ci sont soulignées Section 2.2.3

2.2.1 La plateforme Java

La plateforme Java désigne un standard logiciel pour le développement et l'exécution d'application Java.
Java SE (*Standard Edition*) est la distribution maîtresse de la plate-forme Java, propriété d'Oracle Corporation, elle est destinée aux applications pour poste de travail. Java SE possède différentes versions définies par des spécifications. Ces spécifications sont plus précisément composées des spécifications du langage Java [60] et des spécifications de la machine virtuelle Java [102]. Java SE 8, ou simplement Java 8[17], est la version considérée dans cette études. Java 8 fût lancée en 2014, le lancement de Java 9 est quant à lui prévu courant 2017.

2.2.1.1 Java Development Kit

Le JDK (*Java Développement Kit*) est l'implémentation de référence de Java SE distribuée sous licence BCL (*Binary Code Licence*) par Oracle (ainsi le JDK8 désigne l'implémentation de Java 8, une implémentation du JDK possédant par ailleurs son propre versionnement).
Le JDK se compose du JRE (*Java Runtime Environnement*) et d'outils (compilateur `javac`, debugger, profileur) facilitant le développement des applications Java. Le JRE fournit lui les programmes nécessaires à l'exécution des applications à savoir les librairies (Java et natives) ainsi que la JVM.
De manière stricte, les termes JDK et JRE désignent les implémentations Oracle de Java SE mais sont en pratique utilisés pour d'autres implémentations. La version du JDK 8 considérée dans cette étude est la version 1.8.0_51 (cf. Figure 2.8 Section 2.5).
L'OpenJDK est une implémentation open-source de Java SE maintenue par le projet OpenJDK regroupant de nombreux contributeurs incluant Oracle, IBM, Apple ou encore SAP. À partir de Java 7, l'implémentation OpenJDK est devenue l'implémentation de référence officielle de Java SE. Le JDK est par conséquent en très grande partie similaire à l'OpenJDK avec quelques différences minimes néanmoins. La JVM HotSpot est commune aux deux implémentations.
L'OpenJDK est lui distribué sous licence GPL avec une clause autorisant son utilisation par des applications non GPL.
Le JDK d'IBM (IBM SDK Java TE) est une autre implémentation de Java SE utilisée en production à destination plus particulièrement des plateformes AIX[18]. À eux deux, les JDK d'Oracle et d'IBM permettent de couvrir le spectre des architectures HPC et garantissent la portabilité des applications Java sur ces plateformes.

[17] Plus généralement, et dans la suite du manuscrit, Java X fait implicitement référence à Java SE X
[18] AIX (pour *Advanced Interactive eXecutive*) est un système d'exploitation de type Unix commercialisé par IBM. Il tourne exclusivement sur les processeurs IBM PowerPC

2.2.1.2 Machine Virtuelle Java

La JVM (*Java Virtual Machine*) est l'environnement d'exécution (ou *runtime*) chargé d'exécuter les applications Java sur une architecture donnée. Elle est totalement indépendante du langage Java puisqu'elle reconnait uniquement le format de fichier `class` contenant les instructions à exécuter reconnues par la JVM (le bytecode), une table de symbole et d'autres informations auxiliaires.
Ainsi, la définition d'une application Java ne se restreint pas à "une application développée en Java" mais désigne une application exécutable par la JVM ce qui inclut les applications développées dans un langage ou plusieurs langages pouvant être compilés vers des fichiers exécutables `class`. Ces langages sont parfois qualifiés de langage de la JVM. Il en existe de nombreux parmi lesquels Scala, Groovy ou encore Clojure [184]. Le langage considéré dans cette étude reste néanmoins le Java.
Il existe de nombreuses implémentations de la JVM [183] chacune pouvant mettre l'accent sur une fonctionnalité particulière en fonction du domaine d'application et de l'architecture ciblée. On peut par exemple citer la gamme des JVM distribuées (ou DJVM) [200, 199, 201] destinées à faciliter l'exploitation des grappes de calcul où celle des JVM méta-circulaires destinées principalement à la recherche.
Néanmoins, les machines virtuelles de qualité production et de surcroit distribuées librement sont peu nombreuses. Les deux principales étant HotSpot (cf. Section 2.3) et J9. La JVM Zing [5], basée sur HotSpot, est également une JVM propriétaire de qualité production développée par Azul System. Elle intègre de nombreuses améliorations en particulier concernant la scalabilité et la réduction des latences du ramasse-miette. Elle est livrée au sein d'une distribution libre de l'OpenJDK, également maintenue par Azul System, appelée Zulu JDK. JRockit [85] est une autre JVM de qualité production qui fut développée par la société BEA System. Après son rachat par Oracle en 2008 (suivi par le rachat de Sun), Oracle a intégré les avantages de JRockit au sein de HotSpot. La version de HotSpot contenue dans le JDK8 intègre ainsi différentes fonctionnalités issues de JRockit comme le Flight Recorder qui est un profileur à faible empreinte sur le runtime.
En dehors des machines virtuelles de production on trouve également de nombreuses JVM développées pour la recherche. La JVM Maxine [187] est une machine virtuelle développée par Oracle Labs dite méta-circulaire c'est-à-dire écrite elle même en Java. La méta-circularité est une propriété intéressante pour la recherche car elle facilite les modifications de la JVM ainsi que les interactions avec l'application exécutée. Maxine doit être amorcée par une JVM native comme HotSpot. Elle n'interprète pas le bytecode mais compile systématiquement le bytecode en code natif. Jikes RVM [7] est une autre JVM méta-circulaire destinée à la recherche, similaire à Maxine dans son fonctionnement.

2.2.2 Forces du Java

2.2.2.1 Coût de développement

Le développement d'application de grande échelle implique des coûts de développement importants. Les efforts sont répartis dans différentes tâches comprenant la maintenance, les tests et benchmarks, les améliorations de type *refactoring*, le débogage, les évolutions fonctionnelles, ou encore le support de nouvelles architectures.
La première force de Java réside dans sa capacité à réduire ces coûts. Une étude statistique recueillant les opinions de développeurs issus des communautés Java, C et C++ a révélé, considérant tous les aspects cités précédemment, des durées de développement en moyenne 50% supérieures en C/C++ par rapport au Java [19]. Ce critère s'est avéré central dans le choix du Java pour l'éditeur Aselta Nanographics.
Les gains de productivité proviennent en premier lieu de mécanismes haut-niveau pris en charge par le langage, en particulier le polymorphisme (cf. Section 2.2.4) mais également la gestion des exceptions, le multi-threading ou les types génériques. Ces gains reposent en grande partie sur l'utilisation d'un environnement d'exécution vers lequel sont déléguées les tâches fastidieuses. Les aspects principaux intervenant dans les gains de productivités sont présentés dans les paragraphes suivants.

Gestion mémoire L'environnement d'exécution prend en charge la gestion mémoire qui est plus précisément assurée par le ramasse-miette ou GC (*garbage collector*). Par ailleurs Java ne définit pas de pointeur, ce qui élimine une source importante de bugs.

Portabilité L'environnement d'exécution prend en charge la portabilité. En effet ce dernier concentre les efforts de portabilité comme il doit être déployé nativement sur toutes les architectures supportées. Les applications Java sont elles compilées dans une représentation machine-indépendante du code : le bytecode. Elle peuvent ainsi être déployées sans effort de portabilité sur toutes les architectures possédant un environnement d'exécution Java. Une application Java peut alors être vue comme un opérande de l'environnement d'exécution Java. Cette caractéristique fondamentale du Java est connue sous le sigle WORA (*Write Once Run Anywhere*).
Dans le cas d'Inscale néanmoins la portabilité est un critère secondaire comme le logiciel est uniquement déployé sur des architectures x86-64. Par ailleurs le logiciel est livré avec le JDK de manière monolithique. Ainsi les architectures supportées par Inscale sont celles supportées par le JDK, à savoir x86, SPARC et ARM, ce qui ne couvre pas tout le spectre des architectures représentées en HPC (cf. Figure 2.1). Notons néanmoins qu'il existe un port de l'OpenJDK pour les plateformes Aix/PowerPC [133]. Par ailleurs l'usage de JNI (Java Native Interface) pour interfacer du code natif réduit également la portabilité.

Écosystème L'autre aspect permettant de réduire considérablement les coûts de développement concerne la richesse de l'écosystème Java. On entend par écosystème l'ensemble des outils facilitant le développement des applications ainsi que l'ensemble des librairies Java utilisables directement par les applications. Concernant les outils de développement, on peut citer les environnements de développement Eclipse, NetBeans ou IntelliJ IDEA ; les outils de test unitaire JUnit et TestNG ; le logiciel d'intégration continue Jenkins ou encore les moteurs de production Maven et Ant. La plupart sont utilisés dans le développement d'Inscale.
L'écosystème Java se compose également d'un nombre important de librairies scientifiques. On peut par exemple citer la librairie Apache Commons Math massivement utilisée. Elle couvre de nombreux domaines mathématiques comme les statistiques, l'algèbre linéaire, les éléments finis ou la géométrie.
Par ailleurs JNI permet d'étendre le spectre des librairies utilisables en Java aux librairies natives. Ainsi de nombreuses librairies Java, qualifiées de *bindings*, servent d'interface vers des librairies natives.

Sûreté Les langages traditionnellement utilisés dans le HPC comme le Fortran et le C ont un support limité pour la vérification d'erreur à l'exécution et le debugging qui généralement est long et fastidieux. À l'opposé Java offre lui un support très avancé. L'exemple le plus courant est que lorsqu'une application Java plante, la trace d'appel est enregistrée automatiquement ce qui permet de localiser rapidement la source d'erreur. De plus les spécifications du langage renforcent la sûreté avec différentes opérations de vérification générées implicitement. Enfin, le support avancé et natif des exceptions permet d'améliorer nettement la sûreté des applications.
Par ailleurs, l'environnement d'exécution est en charge d'assurer que la classe chargée vérifie un certains nombre de propriétés garantissant que le code qu'elle contient peut être exécuté sans risque. Cette tâche est effectuée par un vérifieur de bytecode au moment du chargement de classe. Il vérifie par exemple la validité du typage ou l'absence de dépassement de pile. La vérification de bytecode renforce la sûreté des applications et évite certains risques de plantage.

2.2.2.2 Performance

Compte tenu de l'évolution des architectures, l'optimisation de performance peut être séparée en trois approches :

1. L'exploitation du parallélisme ;
2. L'exploitation des cœurs de calcul ;
3. L'exploitation d'accélérateurs matériels en particulier les GPU.

De manière générale, l'utilisation d'un environnement d'exécution favorise les améliorations sur ces trois points considérant les informations résolues dynamiquement exploitables par

les différents composants du runtime (en particulier le compilateur dynamique).
L'exploitation du parallélisme repose en Java sur un support natif et de nombreuses APIs. L'exploitation des cœurs de calcul est elle déléguée au compilateur dynamique. Enfin différents outils existant permettent de faciliter l'exploitation de la puissance de calcul des GPU.

Parallélisme Java est un langage multi-tâches et fournit à ce titre un support avancé pour la gestion des threads et de la synchronisation. Pour garantir la portabilité des applications concurrentes, Java définit son propre modèle mémoire appelé JMM (*Java Memory Model*) [141].
Dans le cas de la JVM HotSpot les threads Java correspondent à des threads natifs. Les latences de synchronisation sont réduites grâce à des techniques comme le *biased locking* [147, 33] où des optimisations du JIT (élimination ou fusion de verrous) [156].
Java 5 a introduit le package `java.util.concurrent` [99] fournissant de nouveaux mécanismes de synchronisation de plus haut niveau que les mécanismes natifs incluant des ordonnanceurs de tâche, des collections concurrentes, des verrous et des barrières.
Le package a été amélioré dans Java 7 avec l'addition de l'API Fork/Join [30] permettant la mise en œuvre du modèle de programmation parallèle éponyme.
Dans Java 8 des améliorations basées sur l'API Fork/Join ont émergé. On peut par exemple citer les méthodes de tri parallèles `parallelSort` dans la classe `java.util.arrays`, ou les méthodes du package `java.util.streams` pour le traitement de données par flux comme les transformations *map-reduce*.
De manière générale les évolutions pour le support du parallélisme sont l'objet en Java d'une amélioration continue considérant les besoins croissant au niveau des applications [127, 130].

Compilation dynamique L'avènement de la compilation dynamique a rendu des langages purement interprété comme Java compétitifs avec des langages compilés statiquement comme le C en terme de performance [25, 152]. La compilation dynamique, a été introduite dans Java 2 et demeure le composant essentiel pour la réalisation des performances.
La compilation dynamique constitue un atout majeur considérant le modèle d'exécution des applications HPC pour deux raisons liées :

1. La prise en compte des informations dynamiques offre un potentiel d'optimisation plus important que la compilation statique ;
2. Le compilateur dynamique se focalisant sur l'optimisation des points chauds applicatifs, dans le cas des applications HPC, l'échelle d'exécution importante garantit la rentabilité des optimisations appliquées.

Les principes de base de la compilation dynamique ainsi que la politique de compilation dynamique utilisée dans HotSpot sont décrits plus précisément Section 2.3.2

Calcul sur GPU L'exploitation de la puissance de calcul des GPU est devenu essentielle pour les applications HPC. Le projet Sumatra de l'OpenJDK [171] a pour objectif de permettre l'exploitation de cette puissance de calcul au sein des applications Java. Le projet vise dans un premier temps une extension de la JVM HotSpot afin qu'elle supporte le calcul sur GPU. Cette extension implique la capacité pour le compilateur dynamique à générer du code pour GPU mais également la capacité pour le GC de gérer la mémoire du GPU. Dans un second la mise en œuvre au niveau des APIs Java sera évaluée.
Un tel support existe déjà dans la version Java 8 du JDK IBM et est décrit dans [81]. Le compilateur dynamique de la machine virtuelle J9 peut affecter certaines tâches à un GPU en se basant sur des heuristiques. Les sections de code impactées sont les boucles du type `parallel().forEach`.
En dehors de la gestion automatisée par le runtime, les outils Rootbeer [139] et Aparapi [6] permettent d'exploiter le calcul sur GPU. Les deux fournissent une API Java ainsi qu'un compilateur statique permettant de générer des noyaux de calcul GPU à partir du code Java. Dans [35], Docampo et Al. propose un étude comparative des différentes API Java existantes pour le calcul sur GPU en considérant à la fois la facilité de programmation et les performances.

2.2.3 Limitations pour le HPC

Les limitations du Java pour le calcul haute performance peuvent être séparées en différents types. Dans cette étude les limitations qui nous intéressent plus particulièrement sont celles concernant l'optimisation de code soulignées Section 2.2.3.1. On peut par ailleurs citer les limitations fonctionnelles ainsi que celles liées à la gestion mémoire. Ces dernières sont soulignées Section 2.2.3.3.

2.2.3.1 Limitations pour l'optimisation de code

Expressivité Les limitations du Java pour l'optimisation de code proviennent principalement de la représentation des programmes sous forme de bytecode. Le bytecode est un format compact (une instruction étant encodée sur un byte) ce qui réduit l'empreinte mémoire des programmes mais limite également le nombre d'instructions encodables à 256. Pour favoriser la portabilité, le bytecode exprime des opérations fonctionnant sur une machine à pile.
La problématique du bytecode concernant l'optimisation de code est son manque d'expressivité vis à vis du langage machine. L'expressivité d'un langage vis à vis d'un langage cible mesure sa capacité à exprimer une sémantique exprimable dans le langage cible. Ce manque d'expressivité s'observe par exemple en comparant le nombre d'instructions bytecode (environ 200 exprimant une sémantique haut-niveau) et le nombre d'instructions machines (plus de 600 pour le jeu d'instructions x86-64).
Ce manque d'expressivité entraîne par exemple l'incapacité d'accéder à des registres spé-

ciaux comme les registres de drapeaux et les compteurs matériels directement depuis du bytecode ou encore l'incapacité d'utiliser certaines instructions qui pourraient permettre d'améliorer les performances. On peut par exemple citer l'instruction `IMUL` effectuant une multiplication entière sur 128 bits considérée dans le Chapitre 6.

Spécifications Le bytecode exprime par ailleurs les spécifications du langage Java qui peuvent s'avérer contraignantes pour les performances. On pense aux vérifications de sécurité générées implicitement. Par exemple le *bound-check* est effectué par les bytecodes du type `<type>a<store/load>`[19] et vérifie que l'index de l'élément accédé est valide. Différentes techniques sont utilisées par les compilateurs dynamiques afin d'éliminer le bound-check [194, 195, 12]. Le *null-check* est lui effectué par différents bytecodes comme `invokevirtual`, utilisé pour l'invocation de méthode, et vérifie que l'objet receveur ne vaut pas `null`. Le null-check est néanmoins masqué dans certain où cas il peut être effectué implicitement (i.e. sans test explicite) en interceptant les exceptions matérielles levées par certaines instructions comme `MOV` [86]. Le null-check implicite est implémenté dans HotSpot (cf. Section 4.2.2.3 Chap. 4). Un autre exemple de contrainte provenant des spécifications est montré Chapitre 6 concernant le bytecode `d2i` effectuant une troncature d'un `double` vers un `int`.

Limitations du compilateur En dehors de l'expressivité du bytecode, l'autre limitation concernant l'optimisation de code provient des capacités du compilateur dynamique. Comme précisé Section 2.3, le compilateur dynamique doit, contrairement à un compilateur statique, géré le compromis qualité du code généré /durée d'optimisation. Cette considération se traduit généralement par une contrainte sur la durée allouée à l'optimisation de code et certaines optimisations comme l'allocation de registre ou la vectorisation privilégient ainsi des algorithmes rapides au détriment de l'optimalité du code généré [98, 186]. Concernant la JVM HotSpot, l'auto-vectorisation a été introduite dans Java 7 mais s'avère, y compris dans Java 8, très limitée comme montré dans le Chapitre 5. De nombreuses évolutions sont à venir avec Java 9 comme par exemple le support de l'extension AVX-512 [123], la vectorisation de l'intrinsic `Math.sqrt` [131] ou la vectorisation des réductions [125].

Intrinsics Une des solutions pour augmenter l'expressivité du langage ou contourner les limitations du compilateur est de faire usage de fonctions dites intrinsèques ou *intrinsics* (également appellées *builtin* dans le contexte de GCC). Ces fonctions permettent dans le cas de GCC d'exprimer directement un grand nombre d'instructions machines. Néanmoins concernant Java, différentes contraintes comme la portabilité limitent la définition de fonctions représentant des instructions machine-spécifiques et donc l'usage d'intrinsics

[19] Il s'agit des bytecodes utilisés pour lire (`load`) ou écrire (`store`) un élément dans un tableau primitif. "*type*" vaut respectivement `a`, `d`, `f`, `l`, `i`, `s`, `b` pour un élément de type `Object`, `double`, `float`, `long`, `int`, `short`, `byte`.

pour faire de l'optimisation de code. Ces problématiques sont abordées plus précisément Chapitre 6.

2.2.3.2 Limitations liées à la gestion mémoire

Les limitations liées à la gestion mémoire sont multiples. Elles peuvent en premier lieu provenir de limitations du ramasse-miette. Des études ont par exemple montré que la gestion mémoire automatique pouvait occasionner dans certains cas des pertes de performance importantes par rapport à la gestion mémoire manuelle [70, 14]. Le ramasse-miette de HotSpot a également montré des faiblesses provenant d'accès distants intensifs lors des collections sur des machines à accès mémoire non uniforme (NUMA) pour des heaps excédant 100 Gbytes (notons que les effets NUMA sont peu significatifs dans le cas de l'application Inscale qui privilégie des instances de JVM dont les heaps sont inférieures à 32 Gbytes, notamment pour bénéficier de pointeurs compressés). Des améliorations ont été apportées dans Java 7 pour ce type d'architectures [121]. Une étude a néanmoins montré que des améliorations significatives pouvaient encore être apportées pour des machines NUMA de 256 à 512 GBytes de mémoire [58].

Le modèle mémoire Java est par ailleurs la source d'une consommation mémoire accrue due à la présence systématique des entêtes d'objet (ou *header*). Sur certaine JVM comme HotSpot l'empreinte des headers est réduite pour les architectures 64-bits grâce à l'utilisation de pointeurs d'objets compressés (cf. Section 2.3.1.2). La sur-consommation mémoire peut être accentuée par la présence de zones de padding. Ces dernières proviennent des contraintes d'alignement dues par exemple aux spécifications Java qui déclarent que tous les objets doivent être alignés sur 8 bytes [60].
La sur-consommation mémoire est d'autant plus significative que la taille de l'objet en mémoire est petite, elle est ainsi importante dans le cas des objets encapsulant des types primitifs. La Table 2.1 montre la sur-consommation mémoire dans HotSpot des objets encapsulant les types primitifs par rapport aux types primitifs.
Cette sur-consommation peut avoir pour effet de bord de réduire la localité mémoire. Certaine technique comme la collocation d'objet ont été étudiées afin de réduire l'empreinte des headers, le principe étant de factoriser un header pour plusieurs objets [198].

Une des contraintes fondamentales du modèle mémoire Java est l'impossibilité de contrôler finement la couche mémoire des structures de donnée à savoir l'organisation et l'emplacement des informations en mémoire. Elle peut avoir pour conséquence une perte de localité mémoire.
Cette perte de localité peut prendre effet au niveau de la hiérarchie mémoire NUMA. Afin d'assurer la localité au niveau NUMA des gestionnaires d'allocation dits "NUMA-Aware" [58, 174] ont été mis au point. C'est par exemple le cas dans la JVM HotSpot.
Cette perte de localité peut également prendre effet au niveau des caches mémoires. Elle

Type objet vs. type primitif	Poids mémoire type objet vs. type primitif [Bytes]	Sur-consommation du type objet
`Long` vs `long` `Double` vs `double`	24 vs. 8	+200%
`Integer` vs `int` `Float` vs `float`	16 vs. 4	+300%
`Short` vs `short`	16 vs. 2	+700%
`Byte` vs `byte`	16 vs. 1	+1500%

TAB. 2.1: Sur-consommation mémoire dans HotSpot entre les objets encapsulant les types primitifs et les types primitifs. Les tailles sont calculées en mode pointeur-compressé (cf. Section 2.3.1.2) et considèrent le padding d'alignement des objets sur 8 bytes

provient en particulier de l'impossibilité en Java de définir des structures de données du type tableaux de structures qui permettent de garantir une la localité (mais également de favoriser la vectorisation) [42]. Le modèle mémoire impacte en corollaire de la localité les performances du code comme il implique de nombreuses indirections pour accéder aux données, ce phénomène est connu sous le nom de *pointer chasing*.
Afin de réduire l'impact du pointer-chasing et d'augmenter la localité, des solutions automatiques comme l'inling dynamique d'objets (ou la collocation dynamique d'objets) ont été étudiées [188].
L'impossibilité de contrôler l'emplacement mémoire des objets peut entraîner des problèmes de *false-sharing* liés à la cohérence de cache ainsi que des problèmes d'alignement des données pouvant occasionner des pertes de performances sur des routines vectorisées (cf. Chap. 5). Dans Java 8 l'introduction de l'annotation `@Contended` [126] permet d'éviter le false-sharing grâce à un support du ramasse-miette et du gestionnaire d'allocation.

Enfin le modèle mémoire Java peut occasionner des allocations en heap excessives provenant du fait que le langage ne supporte pas d'alternatives aux objets en dehors des types primitifs (ces allocations excessives ont par ailleurs pour effet d'alourdir la charge du ramasse miette). Ainsi des valeurs autres que des types primitifs utilisées localement (i.e. sans échapper au scope dans lequel elles sont définies) et qui pourraient donc résider sur la pile ou dans des registres sont nécessairement allouées en heap sous la forme d'objet. Les optimisations du compilateur dynamique telles que l'analyse de localité ou le remplacement scalaire [156] permettent d'éviter les allocations en heap lorsque c'est possible. Par ailleurs l'introduction des *value-types* (prévue pour Java 10) va également favoriser la réduction du coût des allocations dans les applications Java [172].

2.2.3.3 Autres limitations

Limitations fonctionnelles Parmi les limitations fonctionnelles on peut par exemple citer l'indexation des tableaux primitifs avec des entiers 32-bits signés qui limite le nombre d'élément d'un tableau primitif à $2^{31} - 1$ au lieux de $2^{64} - 1$ avec des entiers 64-bits non

signés. On peut également citer certaines spécifications du standard IEEE 754 [18] pour le calcul flottant qui ne sont pas respectées à savoir le support de tous les modes d'arrondis définis avec la capacité de le modifier en cours d'exécution ainsi que la capacité à lever des exceptions telles que *overflow* ou *underflow* lors des dépassements. Enfin on peut citer l'absence de support pour les tableaux primitifs multidimensionnels et les nombres complexes, qui a été pointée du doigt comme une limitation du Java pour le calcul haute performance [109].

Type générique Un des aspects souvent pointé du doigt en Java est l'absence de spécialisation concernant la programmation générique. En C++, la programmation générique via les *templates* permet de programmer des fonctions génériques qui seront par la suite spécialisées par le compilateur en fonction des usages faits dans l'application. Il y aura donc autant de versions qu'il y a d'usages différents. En Java, la méthode ou la classe générique est réellement générique, l'implémentation générique reposant sur la notion d'héritage (cf. Section 2.2.4). Un code générique en Java peut par conséquent être sous-optimal en faisant intensivement usage de polymorphisme et de `checkcast`[20]. Une seconde conséquence de la généricité basée sur l'héritage provient du fait qu'elle s'applique uniquement aux types non primitifs c'est-à-dire aux objets. Le projet OpenJDK *Valhalla* [172] a notamment pour objectif d'intégrer la spécialisation des types génériques en Java (et par conséquent l'extension aux types primitifs). Ces améliorations sont attendues pour Java 10.
Cette limitation ne figure pas dans les limitations pour l'optimisation de code. En effet, comme la généricité se fait au détriment des performances, son gain de productivité n'est pas profitable sur du code critique.

2.2.4 Héritage et polymorphisme

Le polymorphisme est un mécanisme fondamental pour la conception des applications. Il permet de rendre un code générique (en définissant un prototype de fonction pouvant avoir une multitude d'implémentations), permet une évolution fonctionnelle simple (alternative au branchement conditionnel) et permet l'usage de fonction de rappel (*callbacks*). En Java, et plus généralement en Programmation Orientée Objet (POO), le polymorphisme repose sur le concept d'héritage décrit dans le paragraphe suivant.

Héritage en Java Les règles d'héritage sont définies formellement dans les spécifications Java [60]. Un objet possède un unique type concret (défini lorsque l'objet est instancié) qui est soit un tableau soit une classe concrète (par opposition aux classes abstraites). Les types abstraits sont soit les classes abstraites, soit les interfaces. Aucun objet de type abstrait ne peut être instancié.

[20] `checkcast` est le bytecode vérifiant qu'un objet hérite bien d'un certain type, si ce n'est pas le cas il lève l'exception `ClassCastException`

Les classes abstraites sont étendues par des classes appelées classes filles ou classes dérivées. Elles définissent des méthodes abstraites à savoir des prototypes implémentés dans les classes concrètes héritées.
Les interfaces sont des classes abstraites spécifiques ne définissant que des méthodes abstraites (i.e. ni champs d'instance, ni méthodes concrètes). Certaines interfaces sans méthodes, qualifiées d'interface-étiquette (*marker* ou *tag interface*), sont utilisées uniquement pour étiqueter les objets et ne définissent donc pas de prototype à implémenter.
Toute classe exceptée la classe `Object` hérite (ou étend) une et une seule classe appelée classe mère (ou super-classe), qui est par défaut la classe `Object` (ainsi toute classe hérite de la classe `Objet`). Toute classe peut hériter (ou implémenter) plusieurs interfaces. Une interface peut étendre d'autres interfaces. Contrairement au C++, Java ne supporte pas l'héritage multiple qui pose des difficultés de résolution dynamique et complexifie l'usage de l'héritage. Les tableaux héritent de la classe `Object` et des interfaces-étiquettes `Serializable` et `Cloneable`. La règle d'héritage pour les tableaux d'objets est la suivante : le tableau d'éléments de type A hérite du tableau d'éléments de type B si et seulement si A hérite de B.

Polymorphisme On distingue deux types de méthode en Java (et plus généralement en POO) qui sont les méthodes statiques et les méthodes d'instance. Les méthodes d'instance sont les méthodes d'une classe ayant comme premier paramètre une instance de cette classe appelée receveur. Les méthodes statiques sont des fonctions classiques (au sens du C).
La principale différence fonctionnelle entre : utiliser une méthode d'instance et utiliser une méthode statique en lui passant le receveur en premier paramètre est que la méthode d'instance lève implicitement l'exception `NullPointerException` si le receveur est nul, ce qui doit être fait explicitement avec une méthode statique.
Le polymorphisme désigne en Java la capacité de définir des appels de méthode polymorphiques. Un appel polymorphique est un appel dont la méthode ciblée dépend du type du receveur. Le polymorphisme est une caractéristique d'un site d'appel résolue dynamiquement comme un tel site d'appel peut être utilisé de manière monomorphique i.e. avec un unique type de receveur. Ainsi d'un point de vue statique il est plus pertinent de parler d'appel potentiellement polymorphique.
En Java un site d'appel sera polymorphique si le type du receveur est une super-classe ou bien une interface et que ce dernier cible plusieurs méthodes durant son exécution. Les méthodes ciblées étant les implémentations définies dans les classes filles. Le bytecode utilisé lorsque le type du receveur est une interface est `invokeinterface`, et `invokevirtual` lorsque c'est une classe. Un site d'appel polymorphique correspond plus précisément à une opération appelée *dispatch* dont le rôle est d'appeler la bonne méthode en fonction de la valeur des paramètres entrants. En Java le dispatch est dit unitaire comme il ne dépend que du type de receveur. Il s'oppose au dispatch multiple pour lequel la méthode cible dépend de la valeur de plusieurs paramètres (il s'agit d'un mécanisme surtout présent dans

Bytecode d'appel	Type de méthode ciblé	Nature de l'appel
`invokevirtual`	Méthode d'instance non privée	Potentiellement polymorphique
`invokeinterface`	Méthode d'interface	Potentiellement polymorphique
`invokestatique`	Méthode statique	Monomorphique
`invokespecial`	Méthode d'instance privée ou constructeurs	Monomorphique

TAB. 2.2: Appels polymorphiques et monomorphiques en Java

les langage à typage dynamique comme Python).
En dehors des appels polymorphiques, on trouve en Java deux autres types d'appel monomorphiques. Les appels de méthode statiques (ou appels statiques) qui correspondent au bytecode `invokestatic` et les appels spéciaux qui sont soit des appels de méthodes d'instance privées ou des constructeurs et qui correspondent au bytecode `invokespecial` (pour ces derniers le type concret du receveur est nécessairement fixé statiquement). La Table 2.2 résume les différents types d'appel rencontrés en Java.
Les appels de méthodes statiques (bytecode `invokestatic`) sont équivalent au niveau machine à une instruction `call` dont la cible est un opérande immédiat désignant le point d'entrée de la méthode.

2.3 La JVM HotSpot

HotSpot est la JVM, intégrée au JDK Oracle et à l'OpenJDK, considérée dans cette étude. Elle est conçue, implémentée et maintenue par le HotSpot Group. Elle est développée en C++ et contient plus de 1800 fichiers d'en-tête et fichiers sources contenant plus de 700.000 lignes de code. La première version de HotSpot a vu le jour en 1999. Grâce à ses résultats de performance bien supérieurs aux autres JVM, elle fut rachetée par Sun, alors propriétaire de Java (avant son rachat par Oracle), et intégrée par défaut dans Java 3. Son nom fait référence à sa capacité d'optimisation des points chauds (hot spots) compilés dynamiquement.
La JVM possède deux modes d'exécution. Le mode client, utilisé lorsque la puissance de calcul disponible est faible, et le mode serveur qui est le mode par défaut sur les architectures 64-bits.
Le design général de HotSpot est décrit Section 2.3.1. Les principes de la compilation dynamique ainsi que la politique mise en œuvre dans HotSpot sont décrits Section 2.3.2.

2.3.1 Design général

Cette section expose différents éléments fondamentaux du fonctionnement interne de la JVM HotSpot (en dehors de la compilation dynamique, détaillée dans une section à part).

2.3.1.1 Réglages de HotSpot

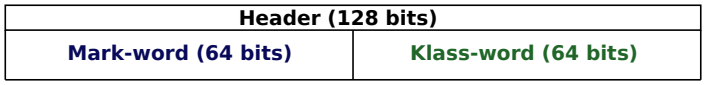
HotSpot possède plus de 800 options[21] permettant de modifier son comportement et affectant les performances des applications exécutées. Le réglage (ou la configuration) de la JVM consiste à fixer ses différentes options afin d'obtenir les effets escomptés. Un aspect fondamental concerne par exemple le régale de la politique de compilation dynamique fixée par l'ensemble des paramètres influant sur la décision du compilateur dynamique (cf. Section 2.3.2.1). Un autre exemple concerne le réglage de la gestion mémoire. La Section 2.8 Chap. 3 expose la configuration de la JVM utilisée dans le cas d'un micro-benchmark. On distingue 3 catégories d'options dans HotSpot : les options standards, non-standards et les options développeurs. Les options standards sont prévues pour être acceptées par toutes les implémentations d'une JVM et restent valides à travers les évolutions (à moins qu'elles ne soient dépréciées). Les options non-standards sont précédées du préfixe `-X:` et peuvent changer d'une évolution à l'autre sans faire l'objet d'une notification. Les options développeurs (préfixées par `-XX:`) sont des options non-standards modifiant le comportement de la JVM mais sans garantie sur l'effet produit et dont la modification est à ce titre fortement déconseillée. Les options standard et non-standards correspondent en arrière plan à des options développeurs. Dans la suite le terme "options" fait référence aux options développeurs.
Chaque option à un type et une valeur par défaut. Les options booléennes sont passées sous la forme `-XX:+<option>` (ou `-XX:<option>=true`) pour être activées et `-XX:-<option>` (ou `-XX:<option>=false`) pour être désactivées. Les options non booléennes sont passées sous la forme `-XX:<option>=<value>`.

2.3.1.2 Représentation des objets

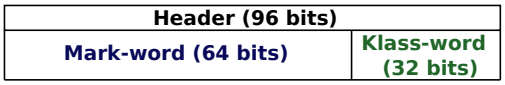
Les objets (ou instances de classe) sont alloués dans la *heap*. La heap ("tas" en français) est l'espace mémoire géré automatiquement par la JVM. Elle est divisée en génération (cf. paragraphe sur le ramasse-miette dans la même section) et contient différentes zones d'allocations comme les TLAB (*Thread Local Allocation Buffer*) [90] dans lesquelles sont alloués les objets à la suite de l'exécution de l'opération `new`. L'utilisation des TLAB permet de diminuer significativement la latence des allocations en évitant l'usage de verrous pour gérer les allocations concurrentes, une allocation en TLAB se réduit alors à un simple incrément de pointeur.
Du point de vue de la JVM (i.e. en C++) les objets Java sont des instances de la classe `oopDesc`. Un pointeur vers une instance de cette classe est qualifié d'oop (*ordinary object pointer*). La classe `instanceKlass` est une classe héritée d'`oopDesc` dont les instances sont les objets représentant les classes (dans lesquelles toutes les informations d'une classe sont sauvées). Les instances de la classe `arrayKlass` représentent elles les types tableaux. Les

[21] Ces options peuvent être visualisées via la commande : `java -XX:+UnlockDiagnosticVMOptions -XX:+UnlockExperimentalVMOptions -XX:+PrintFlagsFinal -version`

oops sont manipulés de manière restreinte dans le code Java (il n'y a par exemple pas la possibilité de faire de l'arithmétique sur les oops).
Les objets ont tous une base commune appelée header (ou entête) qui contient deux champs. Le *klass-word*, contenant l'oop vers la classe de l'objet et le *mark-word* qui contient l'état de l'objet incluant les informations de synchronisation et les informations pour le ramasse-miette (age et génération de l'objet).
Afin de limiter l'empreinte mémoire des headers (cf. Section 2.2.3.2), la JVM HotSpot utilise par défaut sur les architectures 64-bits des pointeurs compressés (option `-XX:+UseCompressedOops`) c'est à dire des oops stockés sur 32 bits au lieu de 64 bits. La Figure 2.4 montre la structure mémoire d'un header d'objet dans HotSpot en mode pointeur-compressé et non-compressé. Les objets étant au moins alignés sur 8 bytes pour garantir des accès alignés, un pointeur compressé correspond à l'adresse d'un mot de 8 bytes dans la heap. Le pointeur vers l'objet est alors calculé par la formule $pointer = base + 8 \times oop$. En effet, la base étant prise à 0, la décompression d'un oop revient à une multiplication par 8 i.e. un décalage de 3 bits vers la gauche ($oop << 3$). Comme 32 bits permettent d'adresser au maximum 4G mots de 8 bytes, le mode pointeur compressé est utilisable pour des heaps n'excédant pas 4×8 Gbytes soit 32 Gbytes.



(a) Mode pointeur non-compressé



(b) Mode pointeur compressé

FIG. 2.4: Structure mémoire d'un header d'objet dans HotSpot sur 64-bits en mode pointeur compressé et non-compressé. Le mode pointeur compressé permet de gagner 25% d'espace sur l'empreinte mémoire des headers

2.3.1.3 Composants du runtime

Une JVM peut être vue comme un ensemble de composants inter-connectés ayant chacun une fonctionnalité précise. Le caractère modulaire de la JVM est d'ailleurs exploité dans certaines librairies servant de connecteur (*glue code*) entre les différents modules et destinées à l'implémentation de machines virtuelles [54, 27]. Tous ces composants affectent les performances de l'application avec une empreinte plus ou moins significative. Les principaux composants de HotSpot sont :

- Le chargeur de classe ;
- L'interpréteur de bytecode ;
- Le ramasse-miette ;
- Le compilateurs dynamique.

Tous ces composants participent à la durée d'exécution globale d'une instance d'application Java t qui peut s'écrire :

$$t = J + n + C + R \tag{2.3}$$

où J est le temps d'interprétation du bytecode applicatif, n est le temps d'exécution du code applicatif compilé dynamiquement, C est le temps d'exécution du code natif applicatif appelé via l'interface native Java (JNI) et R est le temps d'exécution des autres fonctionnalités du runtime (compilateur dynamique, ramasse-miette, chargeur de classe...). Par exemple, cette étude se focalise sur la réduction de la valeur n.

Chargeur de classe Le chargeur de classe (ou CL pour *class-loader*) a pour rôle de charger les fichiers `.class` dans des structures adaptées (comme `instanceKlass` et `arrayKlass` vues Section 2.3.1.2) au sein de la génération permanente de la heap. Il a pour rôle de résoudre les liens symboliques, d'initialiser les classes et les interfaces ainsi que de vérifier la validité du bytecode et du format. Un exemple de résolution de lien symbolique concerne l'opérande des bytecodes `invokestatic` qui est un index de la table des symboles. L'initialisation des classes correspond entre autre à l'exécution des blocs `static` par l'interpréteur. Pour des raisons de performance, HotSpot attend une utilisation de la classe avant de déclencher son chargement. Cette utilisation est soit un accès à un champs statique, un appel de méthode statique ou une instanciation d'objet.
Le chargement de classe peut être initié de différentes manières, soit explicitement via les API Java, soit automatiquement au lancement de la JVM. Il fonctionne par délégation, chaque CL ayant un parent lui déléguant le chargement. Le premier CL actif est le CL dit primordial (*Bootstrap Class-Loader*) qui charge les composants essentiels contenus dans le répertoire d'amorçage (ou `bootpath`) comme le fichier `rt.jar` contenant les classes `java.lang.Object` et `java.lang.Thread`. Le CL primordial délègue ensuite au CL système (*System Class-Loader*) qui est celui chargeant par défaut les classes applicatives (il charge les classes contenues dans le `classpath` dont celle contenant la méthode `main`). Ce dernier délègue ensuite aux différents CL applicatifs. La table de hachage `SystemDictionary` contient toutes les classes chargées et associe à un couple (*identifiant de la classe*, *CL de la classe*) le pointeur de classe correspondant.

Interpréteur La JVM HotSpot utilise un interpréteur qui est dit basé sur des templates (*template-based interpreter*). Le code utilisé par l'interpréteur est généré au démarrage de la JVM à partir d'une table (`TemplateTable`) dont les entrées (les *template*) contiennent des méta-informations ainsi que le code assembleur d'un bytecode donné. L'option `-XX:+PrintInterpreter` permet d'afficher cette table au démarrage de la JVM.
L'avantage de ce design, en dépit de sa consommation mémoire plus forte, est qu'il accélère l'interprétation par rapport à l'utilisation plus classique d'un `switch` dans une boucle. En effet ce dernier, d'une part, effectue de nombreuses comparaisons (en grand O du nombre de bytecode) et, d'autre part, utilise une pile séparée pour la transmission des arguments.

Certains bytecodes, trop complexes pour être implémentés en assembleur, font appel à des fonctions du runtime. L'interpréteur, en plus d'interpréter le bytecode, participe également à la collecte d'informations durant l'exécution destinées à être utilisées par le compilateur dynamique.

Ramasse-miette Différents ramasse-miettes (ou GC pour *Garbage Collector*) sont utilisés par HotSpot. Le GC séquentiel est celui utilisé par défaut dans le mode client. Il stoppe l'application et effectue le ramassage sur un cœur. Les deux autres types de GC utilisés sont le GC concurrent et le GC parallèle. Le GC concurrent n'est pas bloquant et tourne parallèlement au code applicatif ce qui permet de couvrir les latences du ramassage mais limite les ressources disponibles, il est plus intéressant dans le cas d'applications nécessitant un faible temps de réponse. Le GC parallèle lui stoppe l'application, sa latence est minimale comme il peut utiliser toutes les ressources. Il s'agit du GC privilégié dans le cas des applications HPC où le temps de réponse importe moins.
Tous les GC sont basés sur un modèle généalogique [176]. Ce modèle est basé sur le postulat que plus un objet est jeune plus il a de chance d'être collecté. Les efforts de ramassage sont ainsi concentrés en priorité sur la jeune génération.
HotSpot découpe la heap en plusieurs zones appelées générations. Les nouveaux objets sont créés dans la jeune génération puis collectés via un algorithme de type stop-and-copy. Les objets non collectés après plusieurs passages se retrouvent dans la vieille génération. La génération d'un objet est stockée dans le mark-word (cf. Section 2.3.1.2) et actualisée par le GC à chacun de ses passages. Lorsque la vieille génération est pleine le GC utilise un algorithme du type mark-and-compact sur toute la heap. Ce GC étant bloquant il est qualifié de stop-the-world.

Compilateurs dynamiques HotSpot contient deux compilateurs dynamiques : C1 et C2. C1 est optimisé pour la réduction de la durée de compilation mais produit en contrepartie un code de qualité moyenne. C2 est lui focalisé sur l'optimalité du code généré mais induit des latences de compilation bien plus élevées que C1. Dans les versions antérieures à Java 7, C1 est seulement utilisé en mode client alors que C2 est utilisé en mode serveur. Depuis Java 7 les deux compilateurs peuvent être utilisés de manière combinée en mode serveur. Ce mode, appelé compilation étagé (*tiered compilation*), est le mode par défaut dans Java 8. Il est présenté plus en détail Section 2.3.2.3.

2.3.1.4 Safepoints

Les safepoints désignent des points de synchronisation générés à des emplacements précis du code par le compilateur dynamique pour initier des opérations du runtime pouvant avoir des effets de bord. Ces opérations sont par exemples le ramasse-miette dit stop-the-world (cf. paragraphe sur le ramasse-miette) ou certaines formes de désoptimisation (cf. Section 2.3.2.2). Un safepoint peut désigner plus généralement l'état d'exécution d'un

thread n'étant pas bloquant pour le déclenchement d'opérations de la JVM. Les points de synchronisation sont alors désignés comme étant des sondages de safepoint (*safepoint polls*).
En ces points, différentes informations sont connues comme l'emplacement des racines du ramasse-miette ainsi que l'ensemble des informations nécessaires pour basculer de l'exécution de code natif vers l'interpréteur[22]. Par exemple, si un thread effectue une allocation qui initie un passage du ramasse-miette ou bien si un thread invalide le code natif d'une méthode alors les threads voisins bloqueront au niveau d'un safepoints si ces opérations sont bloquantes.
Afin de limiter l'impact des safepoints sur les performances[23], leur fréquence d'apparition dans le code est minimisée tout en garantissant une attente raisonnable pour le déclenchement des opérations bloquantes. En mode interpréteur les safepoints se situent entre chaque bytecode. Dans le code compilé, les safepoints se situent avant les boucles non comptées[24] (excepté si la boucle contient un appel de méthode) ainsi qu'à la fin de chaque méthode.
Le mécanisme de sondage dans HotSpot se fait sur linux/x86 via l'instruction[25] :

- `TEST %eax,offset(%rip)`

La page d'adresse `rip+offset` est rendue inaccessible[26] lorsque une demande d'opération bloquante est initiée par un thread. Comme la page est inaccessible, la JVM peut intercepter un signal de violation de segmentation (SIGSEGV) lorsque qu'un thread atteint un safepoint. Ainsi le runtime peut déterminer que tous les threads ont atteint un safepoint et déclencher l'opération. L'adressage relatif (l'adresse de la page est calculée à partir de son décalage `offset` par rapport à l'adresse de l'instruction courante contenue dans `rip`) permet de réduire la taille d'encodage de l'instruction par rapport à un adressage absolu.
L'option `-XX:+PrintSafepointStatistics` de HotSpot permet de visualiser quelles opérations ont été initiées au niveau des safepoints ainsi que la durée d'attente avant déclenchement.

2.3.2 Compilation dynamique

La Section 2.3.2.1 décrit le principe général de la compilation dynamique et l'objectif de la politique de compilation dynamique puis la Section 2.3.2.2 décrit les différentes techniques utilisées. La Section 2.3.2.3 décrit plus particulièrement la politique de compilation étagée mise en œuvre dans HotSpot. Enfin la Section 2.3.2.4 décrit plus précisément le compilateur C2 intégré dans HotSpot qui est le compilateur considéré dans cet étude.

[22] Ce basculement nécessite une conversion de la pile native vers la pile interprétée puis l'appel à l'interpréteur avec l'index du premier bytecode dans la méthode succédant au point de synchronisation
[23] L'impact des safepoints sur les performances provient également des contraintes sur l'optimisation de code introduites
[24] Les boucles sont dites comptées si le nombre d'itérations est borné par une valeur arbitraire jugée raisonnable. Cette valeur est typiquement la constante `Integer.MAX_VALUE`
[25] La syntaxe assembleur AT&T est utilisée
[26] Typiquement via la fonction POSIX `munmap` (*memory unmap*)

2.3.2.1 Principe général et métrique d'efficacité

le principe de base de la compilation dynamique provient du fait que l'interprétation du bytecode est bien trop lente par rapport à l'exécution de code natif, la compilation dynamique permet alors de réduire cet écart en compilant le bytecode en code natif durant l'exécution. L'exécution natif de ce code entraîne alors un gain de performance important par rapport à son interprétation.

Niveau d'optimisation et citicité du code La compilation dynamique peut être vue comme une optimisation ayant un seuil de rentabilité. Plus précisément, elle introduit une durée supplémentaire au niveau du temps d'exécution global qui dépend de la quantité de code compilé et du niveau d'optimisation appliqué. Ces deux paramètres doivent ainsi être ajustés afin de minimiser le temps d'exécution global composé du temps de compilation plus le temps d'exécution du code compilé.
Parmi les différents niveaux d'exécution et étant donné un bloc de code, il existe un optimum qui dépend de la criticité de ce bloc de code. Plus le bloc de code est critique plus le niveau d'optimisation optimum est élevé. La Figure 2.5 montre à partir d'un modèle simplifié l'accélération provenant du niveau d'optimisation en fonction de la criticité du code[27]
La criticité d'un bloc de code n'est cependant pas connue avant la fin de l'exécution, il est donc impossible de lui affecter statiquement un niveau d'optimisation qui soit optimum. Par ailleurs le choix du bloc de code a aussi une importance capitale comme il détermine le niveau de contexte utilisable par le compilateur ainsi que la durée de compilation.

Politique de compilation et métrique d'efficacité Le rôle de la politique de compilation dynamique est de maximiser l'efficacité de la compilation dynamique en déterminant quand, comment et quel bloc de code compiler [92, 91]. Cette dernière est fixée par tous les paramètres de la JVM impactant ces différents aspects, ainsi toute configuration de la JVM impactant la prise de décision du JIT définie une nouvelle politique de compilation. Soit X une politique de compilation dynamique et $t_A[X]$ la durée d'exécution de l'instance d'application A obtenue avec la politique de compilation dynamique X. L'efficacité $ef_A[X]$ de X pour l'instance d'application A peut s'écrire :

$$ef_A[X] = \frac{t_A[I]}{t_A[X]} \tag{2.4}$$

où I est la politique de compilation de référence qui est typiquement l'interprétation seule. Cette politique est obtenue dans HotSpot via l'option `-XX:-UseCompiler` qui désactive la compilation dynamique. On peut ainsi mesurer l'efficacité de différentes politique de compilation comme la compilation étagée (cf. Section 2.3.2.3) ou encore la politique *compile-the-world* consistant à compiler toutes les méthodes après seulement une occurrence d'exé-

[27] Le modèle utilisé n'est pas forcément rigoureux sur les ordres de grandeur réels mais sert principalement à montrer que le niveau d'optimisation optimal dépend fortement de la criticité du code

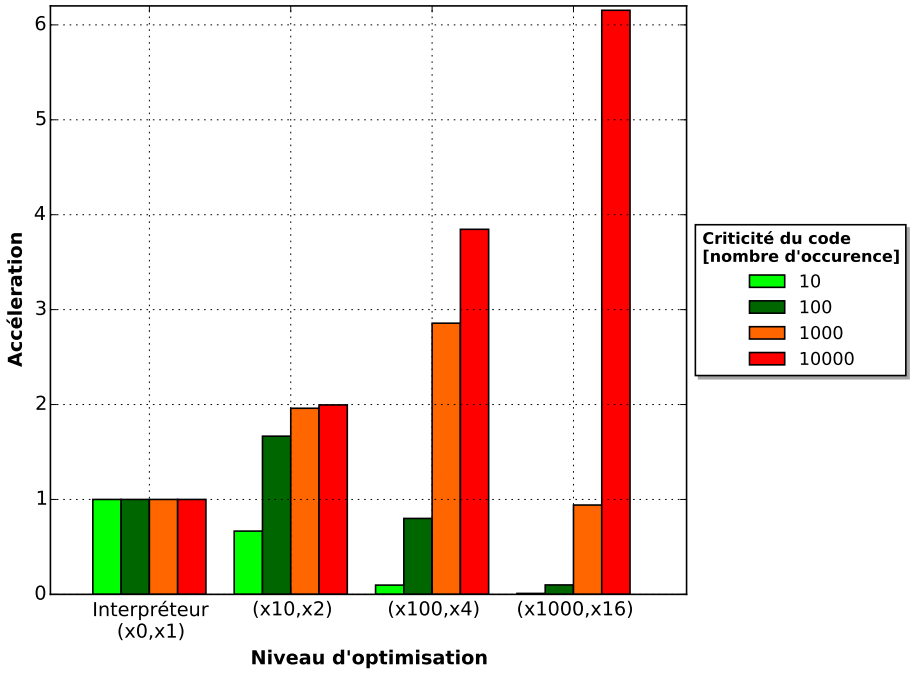


FIG. 2.5: Accélérations provenant de différents niveaux d'optimisation par rapport à l'interpréteur en fonction de la criticité du code compilé. Le niveau d'optimisation est défini par le couple (*coût de compilation*, *accélération*). Le *coût de compilation* correspond au facteur multiplicatif séparant la durée de compilation du code et l'exécution d'une occurrence du code au niveau interpréteur. L'*accélération* désigne l'accélération provenant de l'exécution du code natif par rapport à l'interprétation. En particulier le niveau interpréteur à un coût de compilation de 0 et un facteur d'accélération de 1 ce qui s'écrit (x0, x1). Comme on peut l'observer chaque niveau d'exécution est optimal pour une criticité donnée (par exemple le niveau 1 est optimal pour 100 occurrences). Le second point observable est que plus le code est critique plus le niveau d'optimisation peut avoir un impact significatif sur les performances

cution[28]

2.3.2.2 Techniques utilisées

Détection des points chauds La politique de compilation dynamique exploite le principe des 80/20 qui énonce que 20% du code (les points chauds) représentent 80% du temps d'exécution. La stratégie la plus simple est alors de détecter ces points chauds afin de leurs affecter le niveau d'optimisation le plus élevé.

Cette détection ne doit pas impacter négativement l'exécution du code. Ainsi les techniques sont conçue pour avoir une faible empreinte sur le temps d'exécution.

La détection peut être déléguée aux niveaux d'exécution les plus bas (typiquement l'interpréteur) qui sont alors en charge d'incrémenter des compteurs d'occurrence sur l'exécution

[28]Cette dernière peut être mise en œuvre dans HotSpot avec les options `-XX:-UseTieredCompilation` et `-XX:CompileThreshold=1`

de sections de code [100, 149]. C'est par exemple la technique utilisée dans les JVM HotSpot et J9.
Une autre technique de détection utilisée est l'échantillonnage (*sampling*) [192]. Cette technique est similaire à l'échantillonnage utilisée par les profileurs à savoir que les piles d'appel des différents threads sont inspectées à intervalle de temps régulier par un composant indépendant puis des compteurs d'occurrence sont incrémentés. Cette technique est par exemple utilisée dans les environnements d'exécution Jikes RVM et JRockit. Des techniques de sampling utilisant en plus des compteurs matériels ont également été explorées [21].

Compilation multi-niveaux Les techniques de compilation multi-niveaux permettent d'améliorer l'efficacité globale de la compilation dynamique en associant un niveau de compilation à la température du point chaud (sa contribution à la durée d'exécution globale) [78, 82]. L'idée est d'associer un faible niveau d'optimisation à un point chaud de faible température et un haut niveau d'optimisation à un point chaud de forte température, sachant que plus le niveau d'optimisation est faible, plus la latence de mise en œuvre est également faible.
Les techniques de compilation multi-niveaux utilisent un modèle évolutif : l'état d'exécution d'un bloc de code évolue en fonction de sa température. Ainsi un bloc de température maximale passera potentiellement par tous les états d'exécution intermédiaires avant d'atteindre l'état d'exécution optimal.
En considérant la loi des 80/20, la compilation multi-niveaux permet de réduire la latence précédent l'exécution des points chauds au niveau d'exécution optimal (comme avant d'atteindre ce niveau le point chaud passe par des états intermédiaires plus rapide que l'interpréteur). Ainsi cette technique est souvent caractérisée comme améliorant le temps de démarrage des applications (composé du warmup mais également d'autres opérations comme le chargement de classe) mais peut également permettre d'améliorer les performances asymptotiques.
La compilation multi-niveaux est mise en œuvre dans HotSpot avec le mode étagé combinant deux compilateurs et possédant 5 niveaux d'exécution (cf. Section 2.3.2.3). Elle est mise en œuvre dans J9 qui possède 6 niveaux d'exécution qui sont *interpreter*, *cold*, *warm*, *hot*, *profiling* et *scorching* [162]. Elle est également mise en œuvre dans Jikes RVM qui possède 3 niveaux d'exécution correspondant à 3 degrés d'optimisation du compilateur dynamique (`-O0`, `-O1`, `-O2`).

Unité de compilation et OSR L'unité de compilation désigne la nature du bloc de code transmis au compilateur dynamique pour être compilé. Le compromis posé par le choix des unités de compilation repose sur leur granularité. Plus le bloc de compilation est de granularité élevée, plus les possibilités d'optimisation (du fait de la connaissance du contexte) sont nombreuses. Cependant, plus la granularité est élevée plus la latence de

compilation est élevée et moins la détection des points chauds est précise.
On distingue deux familles de compilateur dynamique. Ceux dont les unités de compilation sont des méthodes (*method-based JIT*) et ceux dont les unités de compilation sont des traces (*trace-based JIT*).
Les traces [11, 67] sont des structures contenant les chemins d'exécution empruntés avec les fréquences associées à chaque bloc. Les traces permettent d'étendre ou de réduire le scope de compilation par rapport à une méthode ce qui permet d'augmenter les possibilités d'optimisations ou de cibler plus précisément les points chauds (en particulier dans le cas où les points chauds sont des boucles contenues dans une méthode). Ces techniques occasionnent cependant un coût de compilation plus élevé provenant principalement de la collecte des traces [77, 69].
La compilation basée sur les méthodes facilite largement la détection des points chauds et offre également assez de possibilités d'optimisation (de surcroît grâce à une politique d'inlining agressive). Elle est ainsi privilégiée dans les JVM de production comme HotSpot et J9.
Un des problèmes de la compilation basée sur les méthodes survient lorsque le code critique est contenu exclusivement dans une méthode en cours d'exécution (i.e. dans une boucle d'itération). Dans ce cas la compilation de la méthode peut être déclenchée, cependant l'exécution de la méthode se poursuit dans le mode d'exécution courant à savoir l'interpréteur. Pour contourner ce problème, la technique de l'OSR (On-Stack Replacement) [44, 159], implémentée dans HotSpot, permet à une méthode en cours d'exécution de basculer du mode interpréteur vers un mode compilé lorsque le runtime détecte une boucle critique. Cette transition se fait en compilant la méthode à partir d'un point d'exécution précis (un safepoint) et en transformant la pile interpréteur en pile d'exécution native. L'OSR est également utilisée dans J9 mais est appelée DLT (Dynamic Loop Transfer) [97].

Optimisations optimistes et désoptimisation L'avantage majeur de la compilation dynamique repose sur les possibilités d'optimisation permises par l'utilisation d'information dynamique. Les informations sont collectées au cours de l'exécution (on parle de profiling) puis sont utilisées par le compilateur dynamique. Les optimisations basées sur les informations de profiling sont qualifiées d'optimisations par profile guidé ou PGO (*Profile Guided Optimisation*). La collecte des informations peut être, tout comme la détection des points chauds, déléguée aux niveaux d'exécution les plus faibles. L'étendue des informations profilées est cependant limitée entre autre par le coût occasionné par les opérations de collecte [94].
Le profiling permet d'appliquer des optimisations dites "optimistes". Une optimisation optimiste est une optimisation dont la validité repose sur l'hypothèse optimiste que, si une condition est vérifiée lors de la phase de profiling, elle sera vrai pour toute l'exécution de l'application.
La désoptimisation [36, 88] est la technique, utilisée dans HotSpot, permettant au code

natif de basculer vers le mode interpréteur si la condition de validité d'une optimisation n'est pas vérifiée pendant l'exécution. La désoptimisation traduit une pile d'exécution native en pile interpréteur, elle peut ainsi être vue comme l'opération inverse de l'OSR qui traduit une pile interpréteur en pile native. La désoptimisation peut avoir différents effets sur la méthode native. Cette dernière peut être invalidée ou non (l'invalidation signifiant que la méthode n'est plus exécutée) et/ou recompilée ou non.

2.3.2.3 Compilation étagée dans HotSpot

La compilation étagée dans HotSpot (*tiered compilation*) [180] introduite dans Java 7 combine les compilateurs dynamiques C1 et C2 afin de réduire la phase de warmup (cf. paragraphe sur la compilation multi-niveaux Section 2.3.2.1).
Le compilateur C1, qui minimise les latences de compilation en contre-partie d'un code généré non-optimal, est utilisé comme niveau d'exécution intermédiaire entre l'interpréteur et le compilateur C2 qui lui est focalisé sur l'optimalité du code généré en contre-partie de latences de compilation plus élevées. Cette stratégie accélère le warmup comme le code généré par C1 est exécuté bien plus rapidement que le code interprété.
Les différents niveaux d'exécution utilisés dans le mode étagé et dans le mode normal (i.e non-étagé) sont montrés Table 2.3.

Niveaux d'exécution		**Mode étagé**	**Mode normal**
0	Interpréteur (compteurs et profiling)	✔	✔
1	C1	✔	✘
2	C1 avec compteurs uniquement	✔	✘
3	C1 avec compteurs et profiling	✔	✘
4	C2	✔	✔

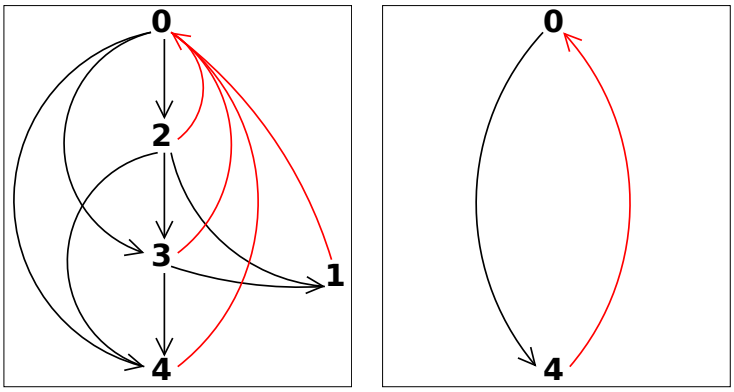
TAB. 2.3: Niveaux d'exécution dans les modes étagé et normal. Dans le mode étagé, le compilateur C1 est utilisé comme niveau d'exécution intermédiaire pour accélérer la phase de warmup entièrement gérée par l'interpréteur dans le mode normal

L'interpréteur est en charge d'incrémenter des compteurs d'occurrence d'exécution en différents points d'encrage dans le bytecode. Ces points sont plus précisément les sites d'appel de méthode et les entrées de boucle. Ces compteurs servent à la détection des points chauds et au déclenchement de la compilation. L'interpréteur est également en charge de collecter des informations sur le profile d'exécution. Ces informations sont de deux sortes : le profiling de type qui recueille les fréquences d'utilisation des différents type d'objet rencontrés et les compteurs de branchements qui recueillent les fréquences d'exécution des branchements conditionnels. Ces informations sont par le suite utilisées par les compilateurs afin de générer un code plus rapide (cf. Section 3.3.2.2 Chap. 3). Le profiling étant plus coûteux que la détection des points chauds, il est effectué uniquement pour les méthodes dont l'occurrence atteint un certain seuil.
En mode étagé, le niveau d'exécution 3 reproduit les fonctions de l'interpréteur à savoir

la collecte des compteurs et le profiling. Le niveau d'exécution 2 se contente de la collecte des compteurs. Les niveaux 0, 2 et 3 notifient régulièrement le runtime de la valeur des compteurs.
L'exécution débute au niveau 0 puis se poursuit au niveau 2 ou au niveau 3. Le choix dépend de la taille de la file d'attente de C2. Comme le niveau d'exécution 3 est environ 30% plus lent que le niveau d'exécution 2 [180], la transition se fait vers le niveau 2 si la taille de la file d'attente de C2 est importante. En effet, le au niveau d'exécution 3 servant à recueillir suffisamment d'information de profiling pour le niveau 4, il vaut mieux effectuer la transition 0->2->3->4 que la transition 0->3->4 si le temps passé au niveau 3 est important du fait de la longue file d'attente de C2 qui freine la transition ->4.
Après la première compilation par C1 aux niveaux 2 ou 3, des informations sur la méthode sont collectées (comme le nombre de bloc de base et le nombre de boucle). Si la méthode est jugée triviale, la transition au niveau 4 est inutile et la méthode transite à la place vers le niveau 1. On peut ainsi avoir les transitions 0->2->1 et 0->3->1.
Si les transitions ->2 et ->3 sont trop lente du fait de la saturation de la file d'attente de C1, la méthode peut être profilée au niveau d'exécution 0. Si le profiling est complété au niveau 0 et que la file de C2 le permet, la méthode peut transiter directement vers le niveau 4, on a alors la transition 0->4. Si la file de C2 est saturée, la méthode transite d'abord vers le niveau 2 avant d'atteindre éventuellement le niveau 4, ce qui donne la transition 0->2->4.
Les méthodes sont dans les files d'attente rangées par ordre de criticité et des filtres sont appliqués régulièrement pour n'y conserver que les méthodes les plus critiques.
Le mode normal fonctionne de manière bien plus simplifiée. L'exécution débute au niveau 0, à partir d'un certain seuil, l'interpréteur débute le profiling. Lorsque la phase de profiling est complétée la transition vers le niveau 4 est effectuée.
Tous les niveaux compilés utilisant des optimisations optimistes, elles peuvent potentiellement être invalidées et donc transiter vers le niveau d'exécution initial 0.
La Figure 2.6 montre les transitions possibles entre les différents niveaux d'exécution dans les modes étagé et normal. Les niveaux évoluent depuis le niveau initial 0 vers les niveaux d'exécution optimaux pour les performances à savoir 1 ou 4 en mode étagé et 4 en mode normal. Ces transitions évolutives sont représentées en noir. Les transitions en rouge sont dues à l'invalidation du code du fait de la non vérification des prédicats optimistes sur les optimisations effectuées.

2.3.2.4 Le compilateur C2

C2 est un compilateur dynamique conçu pour générer un code hautement optimisé [134]. Il est ainsi particulièrement adapté aux applications de type HPC pour lesquelles la phase d'exécution asymptotique est largement dominante par rapport à la phase de démarrage.



(a) Mode étagé (b) Mode normal

FIG. 2.6: Transitions possibles entre les différents niveaux d'exécution dans HotSpot (cf. Table 2.3) en mode étagé et normal. Les niveaux évoluent depuis le niveau initial 0 vers les niveaux d'exécution optimaux pour les performance à savoir 1 ou 4 en mode étagé et 4 en mode normal. Ces transitions évolutives sont représentées en noir. Les transitions en rouge sont dues à l'invalidation du code du fait de la non vérification des prédicats optimistes sur les optimisations effectuées

Représentation intermédiaire L'efficacité de C2 repose en grande partie sur sa représentation intermédiaire (IR) nommée *Ideal* [29] qui favorise de nombreuses optimisations. Ideal forme un graphe de nœuds dans lequel les nœuds représentent des opérations machine-indépendantes. Un nœud est lié à une collection ordonnée de nœuds d'entrée et produit une unique valeur de sortie (pouvant être un n-uplet). Le graphe représente ainsi les liens de dépendance de type producteur-consommateur (*def-use edges*) entre les nœuds. Les dépendances sont de 3 types : les dépendances de donnée, les dépendances de contrôle et les dépendances I/O servant à forcer l'ordre séquentiel de certaines opérations. Les dépendances de donnée sont plus précisément des dépendances entre variables ou des dépendances mémoire. L'IR utilise par conception une représentation SSA (Static Single Assignment). Elle ne définit pas de bloc de base, même si ces derniers sont exprimés via des nœuds spécifiques appelés régions (`RegionNode`).

Phases de compilation Le flow de traitement général se résume en plusieurs étapes qui sont :

1. La traduction de la méthode dans l'IR Ideal (ou *parsing*) ;
2. L'optimisation de code au niveau de l'IR Ideal ;
3. La conversion de l'IR Ideal dans une IR utilisant cette fois des opérations machines (graphe appelé MachNode). Les blocs de base et le graphe de flow de contrôle sont ensuite reconstitués ;
4. L'allocation de registre suivie d'optimisations et enfin de l'émission du code binaire.

La compilation est découpée en phase. Ces phases peuvent être combinées entre elles, imbriquées ou appliquées de manière itérative. On trouve en particulier les phases suivantes[29] :

- **Parser**. Il s'agit de la première phase qui construit le graphe *Ideal* de la méthode. Le parser fonctionne de manière récursive d'abord en traduisant les blocs de base puis les bytecodes le composant. Lorsque le parser rencontre un nœud d'appel il peut prendre la décision d'inliner si les heuristiques d'inlining le permette. À chaque nœud généré, des optimisations locales sont appliquées ;
- **Remove_Useless**. Élimine les nœuds inutiles de l'IR (il s'agit d'une forme d'élimination de code mort) ;
- **Optimistic**. Insère les assertions optimistes en se basant sur les données de profiling afin de spécialiser le graphe ;
- **GVN**. Il s'agit de l'optimisation *Global Value Numbering* [28]. Cette dernière remplace lorsque c'est possible un nœud par un nœud équivalent dans l'IR (i.e. produisant la même sortie) ce qui a pour effet d'éviter les traitements redondants et de réduire l'empreinte mémoire du code. Par exemple, elle permet au graphe de ne posséder au plus qu'une unique copie d'un nœud produisant une constante donnée ;
- **Ideal_Loop**. Cette phase est en charge des optimisations de boucle. Parmi ces optimisations on trouve le déroulage de boucle, l'élimination du *bound-check* et la vectorisation. La vectorisation utilise l'algorithme SLP (*Super-word Level Parallelism*) [98] qui permet de limiter son empreinte sur le temps de compilation ;
- **Ins_Select**. Il s'agit de la phase de sélection d'instructions qui transforme le l'IR Ideal, composée de nœuds représentant des opérations machine-indépendantes, en l'IR MachNode, composée de nœuds représentant des opérations machine. La sélection d'instructions affecte des sous-graphes à des instructions machine en appliquant l'algorithme BURS (*Bottom-Up Rewrite System*) [137] ;
- **CFG**. Il s'agit de la phase reconstituant les blocs de base et le graphe de flow de contrôle (CFG) lors du passage de l'IR Ideal à l'IR MachNode ;
- **Register_Allocation**. Il s'agit de la phase effectuant l'allocation de registre. C2 utilise un algorithme de type coloration de graphe [16] ;
- **Peephole**. Cette phase effectue des optimisations peephole. Les optimisations peephole reconnaissent puis remplacent certains blocs d'instructions machines pouvant être exprimés de manière plus performante avec d'autres instructions.

2.4 Noyaux de calcul

Inscale (cf. Section 1.2 Chap. 1) est une application complexe contenant plus de 8000 déclarations de type et plus de 60000 méthodes. Comme vu en introduction, elle traite des designs, appelés *layout* dans le domaine de la lithographie électronique. Un layout contient

[29] La totalité des phases est énumérée dans l'entête *phase.hpp* dans les sources d'HotSpot. La nomenclature utilisée pour désigner les phases est ici conservée

un dessin géométrique de circuit intégré composés généralement de plusieurs milliards de formes géométriques planes en plus d'informations utilisées lors du procédé de lithographie électronique (comme la matrice contenant les doses d'électron à appliquer en chaque point du circuit). La Figure 2.7 montre un bout de layout visualisé dans l'interface utilisateur d'Inscale.

Ces designs sont encodés dans divers formats de fichier tels que OASIS (*Open Artwork System Interchange Standard*) [96] ou GDSII [17]. Considérant la complexité des designs en terme de nombre de formes géométriques, ces formats sont généralement optimisés afin de réduire la taille des fichiers grâce à diverses techniques de compression. Le traitement rapide de ces fichiers constitue un enjeu pour les performances. Leur poids mémoire peut varier de 1 Gbytes à plusieurs dizaines de Gbytes.

Un flow d'exécution standard dans Inscale se décompose en différentes étapes comprenant (i) le découpage du design d'entrée pour le traitement parallèle (ii) la répartition des tâches sur le serveur de calcul (iii) le traitement du design (iv) la recomposition du design traité dans le format spécifié.

Les traitements appliqués font principalement appel à de la géométrie algorithmique allant d'opérations booléennes simples (unions ou intersections de polygones) à des transformations plus avancées comme du sizing[30]. Les traitements effectuent également de la simulation numérique par exemple en utilisant des produits de convolution. Ces derniers peuvent recourir à des transformées de Fourier rapides (FFT). Des techniques numériques avancées, comme des processus gaussien pour la recherche d'optimum, sont également utilisées dans certains traitements afin de calibrer les modèles de simulation. Les investigations menées se sont d'avantage focalisées sur du code critique de faible granularité et ce pour différentes raisons. D'une part les problèmes d'expressivité du langage Java (cf. Section 2.2.3.1) emmènent à s'interroger d'abord sur de l'optimisation à grain fin afin de rendre la tâche d'optimisation plus aisée. D'autre part l'optimisation de noyaux de calcul de plus forte granularité débouche en premier lieu sur des considérations de nature algorithmiques. Ces dernières sont prises en charge par l'équipe de développement et dépassent le périmètre de cette étude.

Enfin les investigations ont en grande partie portées sur les sites d'appel critiques. Plus précisément les sites d'appel polymorphiques et les sites d'appel natifs. La criticité d'un site d'appel suppose que le code appelé est de faible granularité dans la mesure où il ne couvre pas le coût d'appel.

L'optimisation des sites d'appel polymorphiques constituent le premier type de code investigué dans cette étude. Le polymorphisme, qui est un mécanisme fondamental pour le développement des applications (cf. Section 2.2.4), est intensivement utilisé dans Inscale sans que son impact sur les performances ne soit bien connu. Le second type de code investigué concernent les routines de calcul vectorisables de complexités linéaires. Ces routines

[30] Un sizing est un type de transformation appliqué à un polygone dans laquelle chaque bord du polygone est déplacé dans la direction perpendiculaire à celui-ci

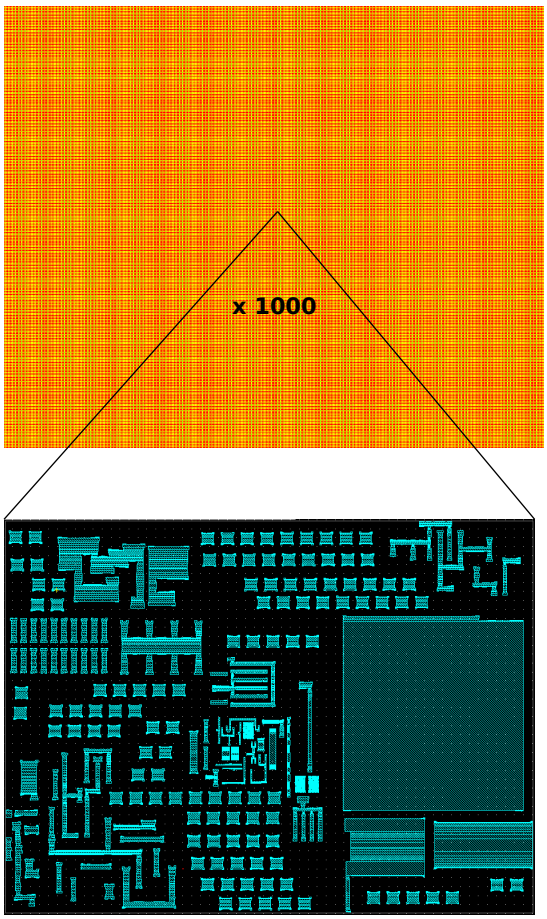


FIG. 2.7: Aperçu d'un layout. L'image du haut est une vue globale du layout caractérisé par sa densité en formes géométriques, la largeur du layout est de l'ordre de 1 mm. L'image du bas est un zoom d'un facteur 1000 au cœur du layout, la largeur de la fenêtre est de l'ordre de 1 µm

de calcul ont été sélectionnées afin d'analyser la qualité du code généré par le compilateur dynamique sachant qu'elles peuvent bénéficier d'optimisations natives comme la vectorisation et de l'exécution out-of-order. Ces routines ont également servies à évaluer l'usage de l'interface native Java (JNI) qui constitue le second type d'appel critique investigué. La complexité linéaire des routines permet de jouer sur la criticité du site d'appel.
Enfin le troisième type de code investigué concerne des routines régulières de faible granularité. Une routine régulière est une routine de granularité est constante (complexité en O(1)), la criticité du site d'appel ne dépend donc pas de facteurs d'échelle modifiant sa granularité.

2.5 Contributions

La problématique précise adressée dans cette étude est donc la suivante : comment optimiser le code de points chauds applicatifs Java de faible granularité s'exécutant dans

un environnement HotSpot/Intel64 ? La configuration utilisée pour les expérimentations est détaillée Figure 2.8.

Étape préalable La première étape, précédant la recherche de solutions, concerne la quantification des performances à savoir comment quantifier les performances de code de faible granularité dans un contexte de compilation dynamique où l'isolation du code critique peut altérer la nature du code testé. Cette mesure de performance servant en particulier à valider des optimisations ou bien sélectionner l'implémentation la plus performante parmi différentes versions. Le Chapitre 3 donne des éléments de réponse en exposant la méthodologie de micro-benchmark utilisée. Cette méthodologie est ensuite mise en œuvre dans les différentes solutions explorées. L'analyse des performances à l'aide de micro-benchmark est qualifiée dans cette étude d'approche analytique par micro-benchmark.

Solutions explorées 3 pistes ont été explorées afin de répondre à la problématique fixée :

1. La première piste, décrite Chapitre 4, concerne les sites d'appel polymorphiques critiques. Elle tente de quantifier le sur-coût induit par l'utilisation du polymorphisme et tente également d'identifier des optimisations pouvant être mises en œuvre au niveau du code Java ou bien au niveau des réglages du compilateur dynamique ;
2. La seconde piste, décrite Chapitre 5, concerne l'utilisation de code natif via l'interface native Java (JNI) afin d'appeler du code natif plus performant que celui généré par le compilateur dynamique C2. La prise en compte du coût d'intégration du code natif en plus de qualité du code appelé définit le compromis qualité du code natif/coût d'intégration qui est l'aspect étudié dans ce chapitre ;
3. Enfin la troisième piste explorée concerne l'ajout d'intrinsic au sein du compilateur dynamique C2 afin de bénéficier d'optimisations natives tout en évitant un coût d'intégration pouvant être bloquant pour l'intégration de code de faible granularité.

Les différentes pistes explorées prennent en considération les contraintes d'intégration au sein d'une application industrielle (ces dernières sont exposées Chapitre 7) et sont caractérisées par leur difficulté de mise en œuvre et leur efficacité.
La première piste se place dans les conditions des développeurs Java qui, de leur point de vue, privilégient les optimisations pouvant être mises en œuvre au niveau Java par refractoring du code. Cette piste propose ainsi des optimisations pouvant être mises en œuvre au niveau du code Java (ou du bytecode) ou bien au niveau des réglages de la JVM. Les optimisations identifiées s'inscrivent dans une approche qualifiée de *JIT-Friendly* (cf. Section 7.1.1 Chap. 7) à savoir une programmation prenant en compte les optimisations du compilateur dynamique afin de produire un code plus performant.
Considérant les limitations du Java exposées Section 2.2.3, la nécessité de faire de la programmation bas-niveau comme de l'assembleur s'est imposée. Les deux secondes pistes

explorées à savoir l'utilisation de JNI et l'extension du compilateur dynamique s'inscrivent dans cette optique. L'utilisation de l'interface native Java s'est présentée en premier lieu comme la piste la plus simple à mettre en œuvre néanmoins, considérant le coût d'intégration du code natif qu'elle introduit, le recourt aux intrinsics a été exploré pour bénéficier d'optimisation native en réduisant le coût d'intégration. La Figure 2.9 résume dans un schéma les 3 pistes explorées.

- Processeur : Intel Core (Ivy Bridge) i5-2500 (@3.30GHz) ;
- Système : CentOS 6.7 (Linux 2.6.32) ;
- Java : java version "1.8.0_51"
  Java(TM) SE Runtime Environment (build 1.8.0_51-b16)
  Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode) ;
- JNI : GCC 4.8.2.

FIG. 2.8: Configuration utilisée pour les expérimentations

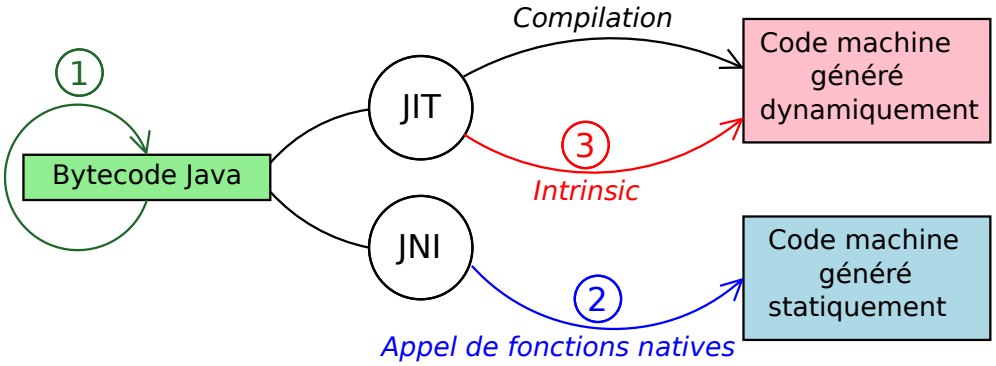


FIG. 2.9: Les 3 pistes explorées correspondent aux 3 passages du code applicatif au code machine. La piste 1 est l'optimisation de bytecode, le code machine est ensuite généré par le JIT suivant le flow de compilation classique. La piste 2 est l'appel de fonctions natives depuis des librairies dynamiques via l'interface native Java (JNI). La piste 3 est l'extension du JIT avec des intrinsics afin de contrôler le code généré

Chapitre 3

Approche analytique par micro-benchmark

La complexité de l'application Java étudiée a nécessité d'isoler des sections de code critiques ou des patterns récurrents, comme l'utilisation de polymorphisme dans des boucles étudiée Chapitre 4, afin dans l'idéal d'établir des règles d'optimisation ou bien d'en quantifier les performances. L'implémentation de micro-benchmarks a ainsi occupée une place centrale afin de quantifier, proposer ou valider des optimisations sur du code critique.
L'approche analytique par micro-benchmark se définit comme l'analyse des performances de bloc code séquentiel par l'utilisation de micro-benchmarks. Ce Chapitre s'attache à détailler la mise en œuvre de micro-benchmark en soulignant les différents aspects à considérer pour obtenir des mesures consistantes.

Les micro-benchmarks sont par exemple utilisée par Agner dans [46] afin de quantifier la latence et le débit d'instruction machine. L'approche est ici similaire mais se place néanmoins a un niveau de granularité supérieure (niveau bytecode) et utilise une méthodologie de mesure ainsi que des métriques différentes. D'autres outils utilisés en calcul haute-performance tel que MAQAO [34] font usage de micro-benchmark afin d'analyser les performances de code. Ils peuvent également êtres utilisés à des fins de rétro-ingénierie comme dans [107] où ils sont combinés à des compteurs matériel afin pour déterminer l'organisation de l'unité de prédiction de branchement.

La Section 3.1 propose une introduction sur les problématiques d'implémentation des micro-benchmarks en Java. La Section 4.1.1 définit l'état du code dans lequel la mesure doit être effectuée, dans un contexte de compilation dynamique où cet état évolue au cours de l'exécution. La Section 3.3 décrit une implémentation générique de micro-benchmark et expose les différents points considérés pour obtenir des mesures consistantes. Enfin la Section 3.4 décrit la configuration de la JVM HotSpot utilisée pour les micro-benchmarks avec les principales options utilisées et leurs effets escomptés.

3.1 Code critique et micro-benchmarks

La Section 3.1.1 propose une introduction sur la détection de code critique au sein des applications Java qui est l'étape précédant l'implémentation de micro-benchmarks. Puis, Section 3.1.2, le rôle des micro-benchmarks ainsi leurs spécificités du Java sont discutés.

3.1.1 Détection de code critique

Préalablement à l'écriture de micro-benchmarks, la détection de code critique est une étape cruciale qui permet d'isoler les méthodes ou patterns problématiques au sein de l'application. Différentes techniques pour détecter les points chauds applicatifs existent. La plus classique repose sur l'utilisation de profileurs Java dont les plus utilisés [103] sont *VisualVM*, *JProfiler*, *Java Mission Control*, ou encore *Yourkit*. Ces derniers échantillonnent les méthodes Java s'exécutant à intervalle de temps régulier (de l'ordre de la milliseconde) puis fournissent une estimation du temps passé à exécuter chacune d'elles.
Les résultats apportés par les profileurs Java doivent cependant être exploités avec précaution et, idéalement, recoupés avec les résultats provenant d'autres profileurs, ces derniers pouvant conduire à des interprétations erronées[1] notamment concernant le niveau de criticité réel des méthodes [113].
Certains profileurs plus avancés, tels que *Oracle Performance Analyzer*, *Intel Vtune*, permettent de combiner profiling bas-niveau et haut-niveau, en distinguant le code de la JVM, le code applicatif interprété ou compilé et le code système.
De manière générale, les profileurs Java n'ont pas de support pour le code généré dynamiquement, ce qui accroît la difficulté d'analyse des performances de l'application.

D'autres approches reposent sur l'utilisation des compteurs matériels qui permettent de recueillir diverses informations sur l'exécution de l'application (défauts de caches, défauts de prédiction de branchement, nombre d'instructions flottantes exécutées...). En Java, leur utilisation n'est pas prise en charge nativement par le langage, ni par la JVM. Leur usage requiert l'appel à des librairies natives via JNI. Différents résultats démontrent l'efficience de l'usage de compteurs matériels afin d'identifier des patterns Java problématiques [80], de détecter les points chauds applicatifs [48, 20, 164] ou encore d'étudier les performances de la JVM [64, 79]. L'avantage des compteurs matériels provient du fait qu'ils permettent de comprendre les résultats de performance et viennent ainsi en complément des mesures de performance.

[1] Ces erreurs d'approximation proviennent de la méthode d'échantillonnage

3.1.2 Micro-benchmarks en Java

Définition et usage Un micro-benchmark peut être définit comme un programme qui fournit une mesure précise de la performance d'un bloc de code [157]. Le préfixe micro[2] fait référence au temps d'exécution du bloc de code de l'ordre de la microseconde, et se distingue ainsi des "macro-benchmarks" exécutant des instances d'application. L'ordre de grandeur de la mesure rend la conception des micro-benchmarks délicate du fait de la sensibilité des mesures aux différents paramètres affectant l'état du code testé.
L'intérêt du micro-benchmark est double. Il permet d'une part de quantifier les performances d'une section de code en isolation par rapport au reste l'application ; mais avant tout, il permet de sélectionner l'implémentation maximisant la performance parmi un ensemble de versions ; le micro-benchmark sert alors de fonction objective à maximiser en proposant des optimisations sur le code.
Ainsi, l'exactitude de la mesure est cruciale afin d'exploiter les résultats pour l'amélioration des performances.

Particularités du Java L'utilisation de micro-benchmarks pour du code Java est souvent sollicitée du fait du haut niveau d'abstraction du langage qui, combiné à la compilation dynamique, rend les performances sensibles à de nombreux paramètres et difficiles à prédire.
Dans l'idéal, l'étape préalable à l'écriture d'un micro-benchmark consiste à identifier les conditions d'exécution réels afin de les reproduire dans un micro-benchmark. Ainsi les conclusions prises seront valident dans les cas d'exécution réels. Cependant ces conditions d'exécution sont parfois impossibles à identifier et donc à reproduire.
Malgré ces précautions, l'implémentation d'un micro-benchmark en Java demeure un exercice délicat. Différentes études ont soulignées la complexité de l'écriture de micro-benchmark Java [73, 59, 55]. On distingue deux raisons principales :

- L'état d'exécution d'une méthode peut évoluer au cours de l'exécution du programme ;
- Le compilateur dynamique peut altérer la nature initiale du code testé en le spécialisant selon le profile d'exécution.

Le premier point concerne la politique de compilation de la JVM qui rend le code évolutif au cours de l'exécution ce qui nécessite de garantir son état d'exécution au moment des mesures.
Le second point concerne les optimisations effectuées par le JIT, comme l'élimination de code mort qui peuvent altérer l'état du code testé et rendre les résultats inconsistants et inexploitables. Un autre exemple concerne la spécialisation du code aux données d'entrée qui peut aussi en altérer l'état du code.
Par ailleurs les mesures peuvent être perturbées par d'autres composants de JVM pouvant

[2] Les termes nano-benchmark ou macro-benchmark sont également employés pour désigner respectivement des benchmarks de l'ordre de la nanoseconde ou d'une application grandeur nature.

interférer avec du code applicatif pendant l'exécution (compilation dynamique, ramasse-miette, chargeur de classe...). Par exemple si le code évalué effectue des allocations d'objet avec l'opérateur `new`, les mesures pourront être perturbées par le GC et le résultat dépendra de la taille du tas allouée au lancement de la JVM.
Différents outils de micro-benchmark existent en Java mais ils sont peu nombreux. L'outil de référence est JMH (Java Micro-benchmark Harness) [154]. Basé sur des annotations, il facilite l'écriture de micro-benchmarks et offre des fonctionnalités avancées pour maîtriser la consistance des mesures.
Pour les tests effectués, l'écriture à la main des micro-benchmarks a été privilégiée, en premier lieu pour garantir un contrôle optimal sur l'état du code testé.
La validité des mesures permet ensuite d'extrapoler les résultats au contexte applicatif. Ces résultats sont à relativiser. D'une part compte tenu de la criticité du code au sein de l'application qui doit être avérée. Et d'autre part compte tenu du contexte d'utilisation dans l'application, notamment la dépendance aux données réelles pouvant varier.

3.2 État du code testé

Dans la Section 3.2.1, la notion d'état d'exécution, utilisée dans ce Chapitre et les Chapitres suivants, est définie ainsi que les différents niveaux d'exécution possibles dans la JVM HotSpot. Dans la Section 3.2.2, l'état du code considéré lors des mesures de performance est fixé en considérant les transitions possibles des états d'exécution.

3.2.1 États et niveaux d'exécution

L'état du code désigne l'ensemble des instructions qui le composent. En Java, il peut être du bytecode interprété, du code natif généré par compilation du bytecode ou du cote natif appelé depuis une libraire dynamique via JNI.
Les niveaux d'exécutions sont eux définis par le runtime et déterminent l'état du code. Inversement chaque état possède un niveau d'exécution. Le niveau d'exécution d'une méthode peut évoluer au cours de l'exécution de l'application et détermine le niveau de performance du code. On peut arbitrairement parler d'état ou de niveau d'exécution pour faire référence à l'état du code dans lequel il s'exécute.
L'évolution du niveau d'exécution nécessite de définir quel état considérer lors des mesures de performance.
La JVM HotSpot possède deux modes d'exécution, le mode étagé (cf. Section 2.3.2.3 Chap. 2), activé par défaut, et le mode normal obtenu en désactivant le mode étagé. Les différents niveaux d'exécution possibles sont résumés pour les deux modes Table 2.3 (cf. Section 2.3.2.3 Chap. 2).

3.2.2 État valide pour les mesures

Les micro-benchmarks implémentés mesurent les performances du code testé lorsqu'il est exécuté dans l'état compilé par C2 qui correspond au niveau d'exécution 4 de HotSpot (cf. Table 2.3 Section 2.3.2.3 Chap. 2). Cet état est, d'une part, l'état optimum pour les performances et, d'autre part, le niveau d'exécution asymptotiquement dominant comme, par conception, tous les points chauds applicatifs parviennent à ce niveau d'exécution après la phase de warmup. Par conséquent, si un point chaud applicatif doit être optimisé, l'état du code à considérer est celui généré par C2.
L'implémentation des micro-benchmarks doit donc garantir que les mesures sont effectuées lorsque le code testé est dans l'état compilé par C2. Comme vue Section 2.3.1 Chapitre 2, les unités de compilation dans HotSpot sont les méthodes. Ainsi garantir que le code testé est dans le bonne état au moment des mesures revient à garantir la méthode encapuslant ce code est dans cet état.

Transitions des états Lorsque le code a atteint l'état compilé par C2, il peut quand même transiter vers des niveaux d'exécution inférieurs. Ces transitions surviennent lorsque le code compilé par C2 est invalidé ce qui a lieu si les hypothèses de compilation sont fausses à l'exécution. Les transitions sont montrées Figure 2.6 Section 2.3.2.3 Chapitre 2. Ces transitions doivent être prises en compte pour la validité des mesures. En effet, il est possible que le code soit dans un niveau d'exécution inférieur au moment des mesures, soit parce qu'elles sont effectuées trop tôt (i.e. pendant le warmup) ou bien parce que le code compilé par C2 a été invalidé, entrainant un biais plus ou moins important sur les résultats. Ainsi, le micro-benchmark doit, d'une part, garantir que la mesure retournée a bien été collectée après le warmup (le nombre d'occurrence d'exécution d'une méthode avant qu'elle ne soit compilée par C2 est par défaut de 10 000 en mode normal, ce seuil pouvant être ajusté) et d'autre part, que la désoptimisation na pas affectée les mesures, ce qui est traité Section 3.3.2.

Choix du mode de compilation Les micro-benchmarks implémentés utilisent le mode normal au lieu du mode étagé qui est le mode par défaut (cf. Section 2.3.2.3 Chap. 2). Le mode étagé vise principalement à réduire la durée du warmup, son activation n'est donc pas nécessaire ni pertinente dans le cas d'un micro-benchmark qui mesure les performances dans l'état compilé par C2 et pour lequel la quantité de code critique compilé est très réduite. De surcroît, le mode normal facilite amplement l'analyse des logs de compilation et le contrôle du déclenchement de la compilation au niveau de l'implémentation du micro-benchmark. Il facilite le réglage des seuils de compilation et des paramètres d'itération du micro-benchmark avec des valeurs pertinentes, à savoir des valeurs qui garantissent que l'état valide pour les mesures est atteint tout en limitant la durée d'exécution du micro-benchmark.

3.3 Implémentation d'un micro-benchmark

Cette section décrit différents points intervenant dans l'implémentation d'un micro-benchmark afin d'obtenir des mesures de performance consistantes avec la meilleure exactitude possible. L'exactitude d'une mesure se compose de deux métriques qui sont sa fidélité et sa justesse. Étant donnée une suite de mesures, la fidélité est estimée par l'écart-type de la série et la justesse est estimée par l'écart entre la moyenne de la série et la valeur de référence.
Dans la section précédente, l'état valide du code pour les mesures à été établis et justifié. Il correspond au niveau d'exécution asymptotiquement dominant pour les points chauds applicatifs. Cette section s'attache à décrire la méthodologie permettant d'obtenir des mesures consistantes.
La Section 3.3.1 propose une implémentation générique de micro-benchmark, avec pour chaque méthode définie, son rôle et son impact sur les mesures. La Section 3.3.2 expose les différentes optimisations du JIT intervenant pouvant altérer l'état du code testé ainsi que les solutions employées pour y remédier. Enfin la Section 3.3.3 mesure l'exactitude du timer[3] utilisé pour les micro-benchmarks afin d'établir son impact sur les mesures de performances effectuées.

3.3.1 Squelette du micro-benchmark

Cette section expose une implémentation minimale et générique de micro-benchmark dont la base est reprise dans les tests de performance effectués dans les chapitres suivants. L'implémentation proposée n'utilise typiquement que deux méthodes, avec un niveau d'itération en guise de warmup :

1. La méthode mesurant la durée d'exécution. Dénommée `compute`, elle encapsule le code testé et retourne une durée d'exécution. Elle est décrite Section 3.3.1.1 ;
2. La méthode chargée du warmup. Dénommée `getBestTime`, elle itère sur `compute` et retourne la durée d'exécution utilisée pour le calcul des performance. Elle est décrite Section 3.3.1.2.

3.3.1.1 Méthode mesurant la durée d'exécution

Les mesures sont effectuées en utilisant des timers dans la méthode `compute` (cf. Table 3.1) qui encapsule le code à tester. Son rôle est de retourner avec le plus d'exactitude possible le temps d'exécution du code testé.
Le timer utilisé est la méthode `System.nanoTime` du JDK dont le fonctionnement et l'exactitude sont détaillés Section 3.3.3.
Lorsque la durée d'exécution du code est du même ordre de grandeur que l'incertitude

[3]Le terme "timer" est un anglicisme désignant les compteurs de cycles d'horloge utilisant l'unité matériel dédiée

sur la mesure, à savoir de l'ordre de la dizaine de nano-seconde (cf. Section 3.3.3.2), il est nécessaire d'itérer sur le code via une boucle afin de réduire l'erreur relative répercutée sur la mesure. Un nombre important d'itération permet d'augmenter la justesse de la mesure. La boucle d'itération ajoute néanmoins des opérations à chaque itération (une addition, une comparaison et un branchement conditionnel) qui diminuent la justesse de la mesure en ajoutant un biais. L'erreur relative due à ce biais diminue quand la durée d'exécution du code testé augmente. À ces opérations peuvent s'ajouter des opérations de chargement et déchargement des registres, dont la présence dépend du code testé[4]. Enfin la boucle ajoute également des opérations en entrée de boucle mais dont le coût relatif est négligeable car amorti rapidement par les itérations de boucle.

`compute` avec boucle d'itération	`compute` sans boucle d'itération

`static long compute(...){`
`  long t0 = System.nanoTime();`
`  for(int i = 0; i < IT; ++i){`
`  // Code a mesurer`
`  }`
`  long t1 = System.nanoTime();`
`  return (t1 - t0);`
`}`
	`static long compute(...){`
	`  long t0 = System.nanoTime();`
	`  // Code a mesurer`
	`  long t1 = System.nanoTime();`
	`  return (t1 - t0);`
	`}`

TAB. 3.1: Méthode `compute` retournant la durée d'exécution du bloc de code étudié. La boucle d'itération est nécessaire lorsque la durée d'exécution du code à mesurer est du même ordre de grandeur que l'incertitude sur la mesure fournie par `System.nanoTime`. Un grand nombre d'itération peut permettre d'augmenter la justesse de la mesure

3.3.1.2 Méthode chargée du warmup

La performance mesurée par le micro-benchmark est calculée à partir de la valeur retournée par la méthode `getBestTime` (cf. Figure 3.1). Cette dernière renvoie la durée minimale d'exécution de `compute` parmi `WU` itérations (cf. Figure 3.1 pour le paramètre `WU`) autrement dit la durée qui maximise la performance.
La méthode `getBestTime` permet de garantir via une multitude d'itérations (valant plus exactement `WU`) que la méthode `compute` qui contient le code testé sera compilée par C2. Elle est donc en charge du warmup. Par ailleurs, les durées d'exécution passées dans les états transitoires ne sont pas prises en compte dans le résultat final étant donné que la durée d'exécution minimale est considérée. Elle permet également d'atteindre, tant que possible, un état stable en considérant différents paramètres matériels tels que les effets de cache lorsque le code lit ou écrit des données en mémoire et la prédiction de branchement lorsque le code testé contient des branchements conditionnels.

[4]Par exemple lorsque le code testé est un appel de méthode nécessitant la libération puis la restauration de registres pour respecter les conventions d'appel

Variance sur les mesures Le code testé et ses données d'entrée étant invariants à chaque exécution, les variations de mesure sur la série d'itération de `compute` sont considérées comme du bruit. Ce bruit correspond à la variance sur la valeur retournée et son importance varie suivant les causes. La variance provenant du timer reste négligeable. Elle est plus significative lorsqu'elle est causée par de la prédiction de branchement ou bien des effets de cache. Elle peut être due également à des valeurs aberrantes ponctuelles dans la suite des mesures pouvant provenir des interactions avec le système. Ces dernières requièrent un traitement statistique plus approfondi pour être éliminées. Considérer la durée minimale parmi une multitude d'itérations (typiquement plusieurs milliers) comme fait par `getBestTime` (cf. Figure 3.1) permet d'éliminer ce bruit. On obtient ainsi une valeur représentative de ce que l'on veut mesurer, à savoir un débit d'exécution lorsque le chemins d'exécution emprunté et les données sont constants. Le nombre d'itération (fixé par le paramètre `WU`) ne doit cependant pas être trop élevé ce qui peut amener dans certains cas avec une forte variance à considérer une valeur exceptionnellement grande et donc non représentative.

Une variation des mesures peut également apparaître lorsque le benchmark est relancé dans une nouvelle JVM (i.e. dans un nouveau processus). Son importance dépend du contenu du code testé dont la performance peut avoir une forte sensibilité aux conditions initiales (alignement du code, alignement des données, état du système...)[5]. Comme le code exécuté et les données d'entrée sont invariants, on admet que cette variance implique une sensibilité aux conditions initiales. Dans ce cas, l'écart-type sur la série de mesures produites en itérant les instances de benchmark est pris en considération et le résultat de performance correspond à la moyenne de la série de mesure.

```
static long getBestTime(...){
    long bt = Long.MAX_VALUE;
    for(int i = 0; i < WU; ++i){
        long t = compute(...);
        if(t < bt)
            bt = t;
    }
    return bt;
}
```

FIG. 3.1: Méthode `getBestTime` retournant la durée minimale d'exécution de `compute`. Cette dernière sert de warmup. par ailleurs, la prise en compte du minimum permet de réduire la variance sur les mesures. La durée retournée est utilisée pour le calcul de la performance

[5]Cette observation a été faite sur du code contenant une forte dispersion au niveau des branchements conditionnels dans l'étude du polymorphisme Chapitre 4.

Inlining de `compute` La méthode `getBestTime` n'est typiquement exécutée qu'une fois par instance de benchmark. Cependant elle peut être compilée par OSR (cf. Section 2.3.2.2 Chap. 2) si le paramètre `WU` est suffisamment grand (cf. Figure 3.1). Si cette dernière est compilée par OSR, `compute` peut être inlinée ce qui peut biaiser les mesures en entrelaçant des opérations entre les timers ou en permettant des optimisations modifiant l'état du code. Ainsi la configuration des microbenchmarks garantit que la méthode `compute` n'est pas inlinée par le JIT.

Métrique de performance La performance $perf$, exprimée en opérations par seconde (Ops/s), est calculée par la formule $perf = nbop \times IT/dt$ ou $nbop$ est le nombre d'opérations exécutées par le code (la nature d'une opération est arbitraire cf. Section 2.1.2.3 Chap. 2), IT le nombre d'itérations effectuées sur le code entre les timers (cf. Table 3.1) et dt la durée minimale en seconde (s) retournée par `compute` pour exécuter les IT itérations.

3.3.2 Prise en compte des optimisations du JIT

Pour un même bloc de code Java, le code machine généré par le JIT dépendra de différents paramètres parmi lesquels les données collectées avant compilation, la configuration de la JVM ou le site d'exécution du code testé (à savoir la méthode `compute` dans l'implémentation générique proposée).
L'isolation du code testé depuis l'application vers un micro-benchmark peut entraîner une modification substantielle de son état d'exécution (par rapport son état d'exécution dans les conditions réelles) si les optimisations effectuées par le JIT ne sont pas prises en compte et ainsi, fournir des résultats erronés menant à de mauvaises décisions.
Les optimisations affectant l'état du code ainsi que les techniques de contournement sont résumées Table 3.2 et détaillées dans les suivantes. La Section 3.3.2.1 considère l'élimination de code mort, La Section 3.3.2.2 la spécialisation du code et la Section 3.3.2.3 les optimisations de boucle.

3.3.2.1 Élimination de code mort

Une méthode produit différentes valeurs de sortie (valeur retournée par `return`, valeur écrite en mémoire...). Une valeur produite dans une méthode est dite consommée lorsqu'au moins une de ses valeurs de sortie en dépend. Lorsque les valeurs produites par un bloc de code ne sont jamais consommées où bien que le bloc de code n'est jamais exécuté, ce dernier est qualifié de code mort. Le JIT peut alors l'éliminer du fait son inutilité, on parle alors d'élimination de code mort.
Pour éviter que tout ou partie du code testé ne soit du code mort, les données de sorties produites doivent impérativement être consommées par la méthode l'encapsulant comme montré Figure 3.2. Consommer les valeurs produites ajoute nécessairement des opérations

Optimisation du JIT	Technique de contournement
Élimination de code mort	Les sorties produites par le code testé doivent être consommées i.e. retournée par `compute` ou écrite en mémoire.
Propagation de constante	Les entrées du code testé sont passées en paramètres de `compute`.
Optimisation de boucle	L'impact des optimisations de boucle doit être analysé au cas par cas selon le code testé. Le code peut être encapsulé dans une méthode pour laquelle l'inlining est désactivé (ce qui ajoute un biais de mesure provenant du coût d'appel). Le facteur de déroulage de boucle peut être contrôlé dans la configuration de la JVM
Spécialisation du code au profiling	Les données d'entrée sont maintenues constantes par instance de benchmark. Le profiling de type peut être désactivé pour obtenir une implémentation générique (cf. Table 3.7, Section 3.4).

TAB. 3.2: Optimisations du JIT pouvant affecter l'état du code testé avec, pour chacune d'elles, la technique employée dans le micro-benchmark afin de la contourner

au code testé (une addition dans l'exemple donné Figure 3.2) et détériore donc la justesse de la mesure.
Une alternative pour éviter l'élimination de code mort est d'encapsuler le code testé dans une méthode appelée par `compute` et de désactiver l'inlining de cette méthode (comme montré Figure 3.2). Dans ce cas, la justesse de la mesure est également détériorée par le coût d'appel de la méthode.

3.3.2.2 Spécialisation du code

Propagation de constante Lorsque les entrées sont des constantes de compilation, le JIT applique de la propagation de constante consistant à remplacer une expression constante par sa valeur et à spécialiser le code pour cette valeur de qui meut mener à de l'élimination de code mort si le code contient des branchements conditionnels. Pour éviter la propagation de constante, les paramètres d'entrée du code testé sont systématiquement passés en paramètres de `compute` comme montré Figure 3.3.

Spécialisation du code aux données profilées Les optimisations provenant du profiling sont à prendre en considération car elles permettent de spécialiser le code au profil d'exécution.
Le profiling permet au JIT de faire des optimisations optimistes. Ces optimisations supposent que les informations collectées avant compilation seront valides pour le reste de l'exécution. Le JIT génère des branches de désoptimisation (aussi qualifiées de *uncommon traps*) qui basculent en mode interpréteur pour poursuivre l'exécution des chemins invalidant les hypothèses de compilation.
Lorsque les occurrences d'exécution des *uncommon traps* franchissent un certain seuil (cf.

```
static double CONSUME;
static long compute(double[] a){
    double consume = 0;
    long t0 = System.nanoTime();
    for(int i = 0; i < IT; ++i){
        consume += sum(a); // inlining de sum
    }
    long t1 = System.nanoTime();
    CONSUME = consume;
    return (t1 - t0);
}
```

Version de `compute` avec inlining de `sum` activé

```
static long compute(double[] a){
    long t0 = System.nanoTime();
    for(int i = 0; i < IT; ++i){
        sum(a); // sum n'est pas inlinee
    }
    long t1 = System.nanoTime();
    return (t1 - t0);
}
```

Version de `compute` avec inlining de `sum` desactivé

```
static double sum(double[] a){
    return a[0]+a[1];
}
```

Méthode `sum` dont le contenu est testé

FIG. 3.2: Méthode `compute` avec boucle d'itération mesurant la performance du code contenu dans `sum`. La boucle d'itération est necessaire pour garantir une bonne justesse de mesure étant donnée la faible granularité de `sum`. Dans la version avec inlining, la valeur retournée est consommée par `compute` en écrivant le champ statique `CONSUME`, ce qui garantit que le corps de la méthode ne sera pas éliminé après inlining mais détériore la justesse de la mesure. La seconde version suppose que `sum` n'est pas inlinée. Dans ce cas le code n'est pas éliminé mais le coût d'appel de la méthode détériore également la justesse de la mesure

Table 4.3 Section 3.4), la méthode est invalidée et bascule au niveau d'exécution 0 (interpréteur). Plus précisément, lors de l'invalidation, l'entrée de la méthode est écrasée par un saut vers une routine runtime chargée de modifier la cible de l'appel. Le code de la méthode peut être effacé du code-cache seulement lorsque la méthode n'est plus active c'est-à-dire que tous les sites d'appel la ciblant ont été mis à jour, la méthode est alors qualifiée de zombie. La visualisation des logs de compilation permet de s'assurer que le code n'est pas invalidée au cours de l'exécution[6].
Pour garantir la consistance des mesures, les données d'entrée doivent être invariantes par instance de micro-benchmark et le résultat final associé au jeux de données correspondant.

[6]Dans les logs de compilation du JIT, l'information `made not entrant` annonce qu'une méthode compilée est invalidée, l'information `made zombie` annonce qu'une méthode n'est plus active et peut être effacée

```
static double CONSUME;
static long compute(){
    double a = 4.;
    long t0 = System.nanoTime();
    CONSUME = Math.exp(a);
    long t1 = System.nanoTime();
    return (t1 - t0);
}
```

Version de `compute` avec avec propagation de constante

```
static long compute(double a){
    long t0 = System.nanoTime();
    CONSUME = Math.exp(a);
    long t1 = System.nanoTime();
    return (t1 - t0);
}
```

Version de `compute` sans propagation de constante

FIG. 3.3: Méthode `compute` mesurant la performance du code contenu dans la méthode `Math.Exp`. Notons que la méthode `compute` ne possède pas de boucle d'itération comme la granularité de `Math.Exp` est assez grande pour obtenir une bonne justesse dans les mesures. Dans les deux versions la méthode `Math.Exp` est inlinée. Dans la première version, il y a propagation de constante. L'expression `Math.Exp(a)` est par conséquent évaluée à la compilation et, durant l'exécution, seule l'affectation de la variable `CONSUME` (servant à éviter l'élimination de code mort) est mesurée ce qui produit des résultats de performance anormalement élevées (d'un facteur $\times 4$ dans ce cas précis). Ainsi pour éviter la propagation de constante, les paramètres du code testé sont systématiquement passés en paramètre de `compute` comme montré dans la seconde version de `compute`

Deux types de profiling affectant l'état du code généré par le JIT sont effectués au runtime :

- Les compteurs de branchement ;
- Le profiling de type.

Les compteurs de branchements sont collectés au cours de l'exécution afin de mesurer les fréquences d'exécution des blocs de base. Cette information permet ensuite au JIT de réorganiser les blocs de base afin de minimiser le nombre de branchements générés, d'améliorer la localité du code ou encore d'ordonner les branchements pour éviter les tests inutiles. Ces informations sont également utilisées afin d'optimiser le code des intrinsics du compilateur dynamique.
Le profiling de type permet de recueillir de l'information sur l'opérande de certains bytecodes. Les emplacements dans le code des différents bytecodes profilés sont appelés points de profiling. L'information recueillie aux points de profiling est utilisée par le JIT pour générer un code spécialisé, et donc plus rapide, pour le bytecode en question. Les bytecodes bénéficiant du profiling de type sont listés Table 3.3. En mode non-étagé, le profiling de type est effectué uniquement au niveau 0 d'exécution (interpréteur) ; en mode étagé le niveau d'exécution 3 effectue également du profiling (cf. Table 2.3).

L'alternative utilisée pour empêcher la spécialisation du code aux profiling de type est de le déactiver depuis les paramètres de la JVM (cf. Section 3.4). On obtient ainsi un code compilé générique. Ce code compilé générique correspond à celui généré lorsque le profiling de type ne dégage pas d'information permettant de faire des optimisations. Un autre moyen est de sélectionner un jeu de donné qui ne permette pas la spécialisation par son caractère aléatoire.

Bytecode	**Pile d'opérandes**	**Opérande-objet dont le type est profilé**	**Bref description**
`invokevirtual` `invokeinterface`	*argN* ... *arg2* *arg1* ***rcv***	***rcv***	Appel la méthode encodée associée au type du receveur ***rcv*** avec les arguments *arg1*, *arg2*..., *argN*.
`checkcast` `instanceof`	***obj***	***obj***	Permet de tester l'égalité entre le type de l'objet ***obj*** et le type constant encodé.
`aastore`	***obj*** *index* *tab*	***obj***	Insère l'objet ***obj*** dans le tableau *tab* à l'index *index* en vérifiant que le type de ***obj*** correspond au type des éléments de *tab*.

TAB. 3.3: Les différents bytecodes bénéficiant du profiling dans HotSpot. En un point d'exécution de ces bytecodes, le code généré dépendra de la distribution des types profilée pour l'opérande-objet

3.3.2.3 Optimisations de boucle

Les optimisations de boucle effectuées par le JIT sont diverses [134, 73]. Dans l'implémentation du micro-benchmark proposée, elles peuvent apparaître dans la version de `compute` avec boucle d'itération (cf. Table 3.1). Rappelons que l'utilisation de la boucle d'itération permet d'augmenter la justesse de la mesure.

Déroulage de boucle Il consiste à dupliquer le corps de boucle qui exécute alors plusieurs itérations en une itération déroulée. Le déroulage de boucle n'est, à priori, pas une mauvaise chose comme il permet d'augmenter la justesse de la mesure en diminuant le sur-coût de la boucle. Néanmoins, il peut entraîner une modification de l'état du code testé en permettant, d'une part, de factoriser des opérations communes aux itérations déroulées et en permettant, d'autre part, la vectorisation du corps de boucle (cf. Section 2.1.1.2 Chap. 2). Le déroulage de boucle peut cependant être désactivé en réglant l'option `-XX:LoopMaxUnroll=1` qui fixe le nombre maximum d'itérations pouvant être déroulées. Néanmoins toutes les boucles sont affectées y compris celles contenues dans le code testé.

Déplacement des invariants de boucle Le déplacement des invariants de boucle, comme son nom l'indique, déplace les opérations invariantes par itération en dehors de la boucle. Cette optimisation modifie ainsi l'état du code testé en factorisant un bloc d'instruction dont la durée d'exécution est dissipée par le nombre d'itération. La Figure 3.4 montre deux versions de `compute` dont l'une est affectée par le déplacement d'invariant de boucle. La solution retenue pour éviter les optimisations de boucle est l'encapsulation du

```
static double CONSUME;
static long compute(double a, double b){
    double consume = 0;
    long t0 = System.nanoTime();
    for(int i = 0; i < IT; ++i){
        consume += add(a,b); // inlining de add
    }
    long t1 = System.nanoTime();
    CONSUME = consume;
    return (t1 - t0);
}
```

Version de `compute` avec inlining de `add` activé

```
static long compute(double a, double b){
    long t0 = System.nanoTime();
    for(int i = 0; i < IT; ++i){
        add(a,b); // add n'est pas inlinee
    }
    long t1 = System.nanoTime();
    return (t1 - t0);
}
```

Version de `compute` avec inlining de `add` desactivé

```
static double add(double a, double b){
    return a+b;
}
```

Méthode `add` dont le contenu est testé

FIG. 3.4: Méthode `compute` avec boucle d'itération mesurant la performance du contenu de `add`. La boucle d'itération est necessaire pour garantir une bonne justesse de mesure étant donnée la faible granularité de `add`. Dans la version avec inlining, la valeur retournée est consommée par `compute` en écrivant le champ statique `CONSUME`, ce qui garantit que le corps de la méthode ne sera pas éliminé après inlining. Cependant, l'expression `add(a,b)` étant un invariant de boucle après inlining, cette dernière est déplacée en dehors de la boucle. Ainsi, seul l'incrémentation de `consume` est mesurée. La seconde version suppose que la méthode `add` n'est pas inlinée. Dans ce cas, l'expression n'est pas déplacée en dehors de la boucle comme son contenu n'est pas connu

code testé dans une méthode suivi de la désactivation de l'inlining pour cette méthode. La boucle permet une bonne justesse dans les mesures et le JIT n'effectue pas d'optimisation de boucle. L'ajout d'un appel de méthode diminue néanmoins la justesse de la mesure avec un impact qui dépend de la granularité du code testé.

3.3.3 Exactitude du timer utilisé

Une des composantes essentielles déterminant la qualité d'un micro-benchmark repose sur le timer mesurant les durées d'exécution. Le fonctionnement du timer utilisé est brièvement décrit Section 3.3.3.1. Afin d'interpréter correctement la durée retournée par le timer il est important de connaitre à la fois sa fidélité et sa justesse. Ces deux attributs sont estimés Section 3.3.3.2.

3.3.3.1 Fonctionnement interne

La méthode statique `nanoTime` de la classe `System` et du package JDK `java.lang` est utilisée pour les mesures de durée. Elle retourne la valeur courante d'un compteur de temps avec une résolution à la nano-seconde (ns). Ce compteur de temps n'a pas de signification absolue. Il est linéairement indexé sur le signal d'horloge CPU et est adapté pour mesurer des durées courtes. Son implémentation est architecture-dépendante. Dans la JVM HotSpot, la méthode `System.nanoTime` est une méthode native et un *intrinsic* du compilateur dynamique. Sous linux elle fait appel à la routine `os::javaTimeNanos` de la JVM qui elle fait appel à la routine système `clock_gettime`. Cette dernière est utilisée en mode `CLOCK_MONOTONIC` rendant le timer insensible aux variations de fréquence du processeur.

3.3.3.2 Estimation de l'exactitude

Afin d'évaluer l'exactitude de `System.nanoTime`, on utilise les mesures retournées par la méthode `emptyNanoTime`. La valeur de référence est 0 elle correspond à la valeur théorique attendue comme la méthode mesure la durée d'exécution d'un bloc de code vide.
Comme rappelé dans l'introduction du Chapitre 3.3, l'exactitude de la mesure correspond à sa fidélité et sa justesse. Étant donnée une suite de mesures, la fidélité est estimée par l'écart-type de la série et la justesse est estimée par l'écart entre la moyenne de la série et la valeur de référence.
Dans le cas considéré, la valeur de référence étant 0, la justesse est donc estimée par la moyenne de la série. La Figure 3.6 montre une distribution de 40 000 mesures retournées par `emptyNanoTime` à travers différentes instances de benchmark.
Avec l'approche utilisée, l'erreur de justesse due à `System.nanoTime` est estimée à environ

```
static long emptyNanoTime(){
    long t0 = System.nanoTime();
    long t1 = System.nanoTime();
    return (t1 - t0);
}
```

FIG. 3.5: Méthode `emptyNanoTime` mesurant du code à vide et permettant d'évaluer la justesse et la fidélité des mesures effectuées. La mesure de référence pour le calcul du biais étant 0

27 ns pour un écart type de 3 ns environ. Ainsi, les mesures effectuées seront entachées de cette erreur de justesse. L'estimation correspond à la moyenne d'une série de 40 000 mesures. Chaque mesure correspond à l'exécution d'une instance de benchmark retournant la durée minimale obtenue sur 100 000 itérations de `emptyNanoTime`.

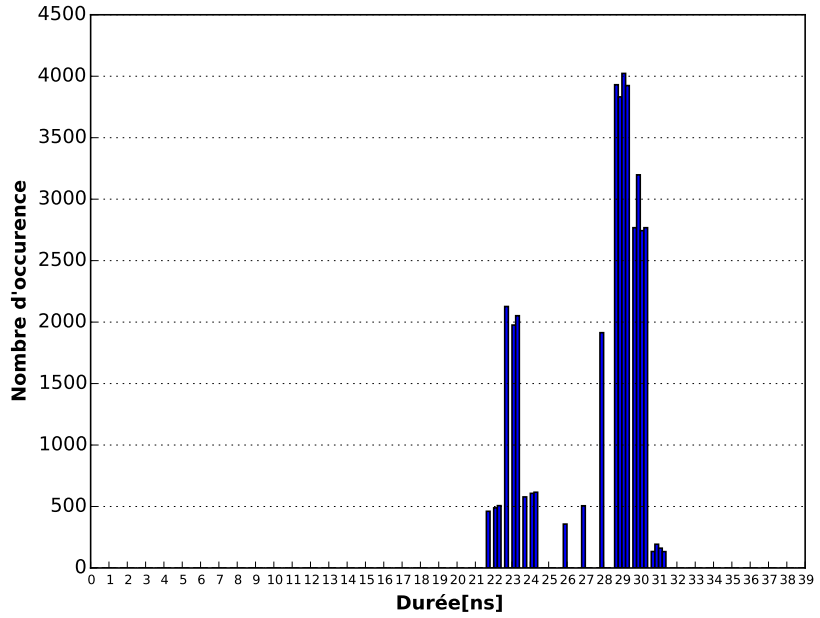


FIG. 3.6: Distribution d'un échantillon de 40 000 mesures retournées par `emptyNanoTime` (cf. Figure 3.5). La *fidélité*, qui est l'écart-type de la distribution, vaut environ 3 ns. La *justesse*, qui est l'écart entre la moyenne et la valeur de référence (ici 0), vaut environ 27 ns

3.4 Configuration de la JVM pour les micro-benchmarks

La JVM expose de nombreux paramètres permettant de contrôler son exécution. L'ensemble de ses paramètres et les valeurs associées définissent la **configuration de la JVM**. Comme détaillé dans la section précédente (cf. Section 3.3.2), les micro-benchmarks requiert un contrôle précis des paramètres d'exécution pour garantir la consistance des mesures. Cette section dresse la liste des différents paramètres impliqués dans la configuration de la JVM HotSpot lors de l'exécution d'un micro-benchmark, avec pour chacun l'effet escompté.

Les paramètres listés sont les paramètres servant de base à la configuration des micro-benchmarks. Ils ne constituent pas une liste exhaustive des paramètres pouvant être utilisés pour contrôler le comportement de la JVM et l'état du code généré. Ils se déclinent en plusieurs catégories :

1. Réglage de la politique de compilation (cf. Table 3.4) ;
2. Visualisation de l'état d'exécution des méthodes (cf. Table 3.5) ;

3. Directives au JIT (cf. Table 3.6) ;
4. Profiling et désoptimisation (cf. Table 3.7).

Les catégories 1, 2 et 3 permettent de garantir et favoriser le contrôle du bon état d'exécution du code au moment des mesures (cf. Table 3.8). La catégories 4 regroupe les paramètres de base permettant de contrôler la spécialisation du code due au profiling ainsi que la désoptimisation (cf. Table 3.2).

Paramètre	**Effet**
`-XX:-TieredCompilation`	Désactive le mode étagé, actif par défaut, pour le mode non-étagé (cf Section 2.3.2.3 Chap. 2).
`-XX:-BackgroundCompilation` `-XX:CICompilerCount=1`	Désactive la compilation en arrière plan. Fixe le nombre de threads du JIT à 1 ce qui garantit que deux méthodes distinctes ne seront pas compilées de manière concurrente. Ces deux paramètres sont utilisés de manière combinée pour garantir la correspondance entre l'ordre des évènements affichés sur la sortie standard (cf. Table 3.5) et l'ordre d'exécution. Ils permettent ainsi de garantir l'état du code.
`-XX:-UseOnStackReplacement`	Désactive l'OSR. Offre un meilleur contrôle sur la compilation. Permet d'empêcher la compilation OSR de `getBestTime` pouvant inliner `compute` et perturber les résultats. D'autres paramètres peuvent être utilisés pour empêcher l'inlining.
`-XX:CompileThreshold=10000`	En mode étagé et sans OSR, correspond au nombre d'occurrences d'exécution d'une méthode avant qu'elle ne soit compilée par C2. Par exemple, lorsque le paramètre vaut 10 000, la méthode ne sera plus interprétée mais exécutée avec le code généré par C2 à partir de la 10 001 ième occurrence. Ainsi, pour garantir que la méthode `compute` soit compilée, le paramètre `WU` utilisé par `getBestTime` (cf. Figure 3.1) doit être supérieur à ce seuil.

TAB. 3.4: Configuration de la JVM HotSpot utilisée pour les micro-benchmarks : paramètres permettant de régler la politique de compilation (sous fond **bleu** figurent les paramètres dont les valeurs diffèrent de celles par défaut et servant de base fixe à la configuration)

Paramètre	Effet
`-XX:+PrintCompilation` `-XX:+PrintInlining`[7]	Affichent les logs du JIT sur la sortie standard lorsqu'une méthode est compilée, invalidée ou rendue inactive. Lorsqu'une méthode est compilée les méthodes inlinées sont également affichées. Sont utilisées conjointement aux options Table 3.4 pour garantir l'état du code au moment des mesures. Peuvent-être désactivées lorsque les seuils sont bien ajustés.

TAB. 3.5: Configuration de la JVM HotSpot utilisée pour les micro-benchmarks : paramètres permettant de contrôler et de garantir le niveau d'exécution du code (sous fond **bleu** figurent les paramètres dont les valeurs diffèrent de celles par défaut et servant de base fixe à la configuration)

Paramètre	Effet
`-XX:CompileCommand=...` `-XX:CompileCommandFile=...`	Permet, pour une méthode donnée, de passer des directives au JIT. Parmi ces directives on a : • `dontinline` : elle permet d'empêcher l'inlining d'une méthode. Elle est utilisée dans notre cas pour empêcher l'inlining de `compute` (cf. Table 3.8) ou du code testé lorsqu'il est encapsulé dans une méthode ; • `dontcompile` : elle permet d'empêcher la compilation d'une méthode. Elle peut être utilisée pour empêcher la compilation OSR de `getBestTime` lorsque l'OSR est activé pour respecté le bon état lors des mesures (cf. Table 3.8) ; • `print` : elle permet d'afficher le code assembleur généré par le JIT pour une méthode donnée. Elle permet de scruter le code généré par le JIT pour l'analyse des performances. Notons que cette option ne peut pas être utilisée pour visualiser le code des méthodes natives (i.e. le code servant d'interface et appelant la fonction native depuis la libraire dynamique). Pour cela il faut utiliser l'option `-XX:+PrintNativeNMethods` qui est cependant globale à toutes les méthodes natives.

TAB. 3.6: Configuration de la JVM HotSpot utilisée pour les micro-benchmarks : paramètres permettant de passer des directives au JIT (sous fond **bleu** figurent les paramètres dont les valeurs diffèrent de celles par défaut et servant de base fixe à la configuration)

Paramètre	Effet
`-XX:-UseTypeProfile`	Désactive le profiling de type. Permet de garantir que le JIT produira un code générique (non spécialisé aux données d'entrée) pour le code mesuré.
`-XX:InterpreterProfilePercentage=33`	En mode non-étagé et sans OSR, correspond au nombre d'occurrences d'exécution d'une méthode (en pourcentage de la valeur fixée par `-XX:CompileThreshold`) avant que l'interpréteur ne commence à recueillir les données de profiling pour cette méthode. Ainsi, lorsque `-XX:CompileThreshold=10000`, l'interpréteur commencera à profiler la méthode `compute` à partir de l'itération 3 300 durant l'exécution de `getBestTime`.
`-XX:PerBytecodeTrapLimit=4` `-XX:PerMethodTrapLimit=100`	Nombre d'exécution minimal d'une branche de désoptimisation, par bytecode (cf. Table 3.3) et par méthode, avant que la méthode compilée ne soit invalidée ; son exécution bascule alors au niveau interpréteur. Le paramètre `-XX:+PrintCompilation` permet de visualiser s'il y a invalidation.
`-XX:-BlockLayoutByFrequency`	Désactive la réorganisation des blocs de base en fonction des fréquences profilées.

TAB. 3.7: Paramètres principaux permettant de contrôler le profiling et la désoptimisation

État de `getBestTime`	État de `compute`
Interpréteur	Interpréteur
Interpréteur	C2-OSR
C2-OSR	Inlinée
C2-OSR	C2
Interpréteur	C2

TAB. 3.8: États d'exécution possibles en mode non-étagé lors des mesures de performance lorsque le résultat est fourni par la première itération de `getBestTime`. L'état d'exécution dépend de la valeur des paramètres `WU` (cf. Figure 3.1) et `IT` (cf. Table 3.1) ainsi que de l'activation ou non de l'OSR. L'état en **vert** est celui garanti par l'implémentation et la configuration du micro-benchmark lors des mesures. Les états en **rouge** sont ceux pouvant fournir des mesures différentes

3.5 Conclusion

Les micro-benchmarks occupent une place centrale dans le domaine de l'optimisation de code. Il servent principalement à comparer des implémentations, valider des optimisations et mesurer des performances. En Java, ils sont particulièrement plébiscités considérant le haut-niveau du langage et la compilation dynamique qui rendent les performances difficiles à prédire. Différentes études ont soulignées la complexité de l'écriture de micro-benchmark Java qui nécessite une bonne connaissance de l'environnement d'exécution sous-jacent (cf. Section 3.1.2).

Ce chapitre expose une implémentation minimale et générique d'un micro-benchmark dont les bases sont reprises dans les tests de performance effectués dans les Chapitres 4, 5 et 6. Les différents points affectant l'exactitude des mesures, comme l'usage des timers, sont exposés ainsi que les différents paramètres impactant l'état du code testé dans un contexte de compilation dynamique.

La configuration de la JVM HotSpot permettant d'obtenir une mesure consistante est décrite avec pour chaque option l'effet escompté. Ces précautions ont pour but principal de maîtriser la consistance des mesures afin de tirer les bonnes conclusions, valides dans des conditions d'exécution réelles.

L'évolution de la méthodologie de micro-benchmark va de paire avec celle de la compilation dynamique. L'arrivée dans Java 9 du contrôle de la compilation (cf. Section 7.2.2.2 Chap. 7) va faciliter mais également solliciter d'avantage l'usage de micro-benchmark.

Chapitre 4

Polymorphisme : analyse pour l'optimisation des performances

Dans ce chapitre, on s'intéresse à l'amélioration des performances pour du code utilisant intensivement du polymorphisme. Le chapitre précédent (Chap. 3) expose la conception d'un micro-benchmark afin de mesurer les performances de code Java en considérant les optimisations effectuées par le JIT et les différents paramètres pouvant impacter la consistance des mesures. Cette approche est ici appliquée à l'étude des performances du polymorphisme afin d'une part d'évaluer son impact sur les performances du code et d'autre part d'identifier des opportunités d'optimisation, notamment en direction des développeurs Java.
La motivation principale provient du niveau d'abstraction du langage Java qui définit des instructions haut-niveau (i.e. les bytecodes) manipulées par les développeurs mais dont le coût réel et l'impact sur les performances est mal connu. Parmi eux, les bytecodes `invokevirtual` et `invokeinterface` qui servent à la mise en œuvre du polymorphisme.
La métrique quantifiant les performances du polymorphisme est définie et évaluée au travers d'un micro-benchmark générique, indépendant d'un traitement applicatif spécifique. Son contenu est décrit Section 4.1.
Afin d'identifier des opportunités d'optimisations, les différents états possibles du code généré par le compilateur C2 sont exposés Section 4.2.
Les performances des différents états mesurés par micro-benchmark sont exposées et analysées Section 4.3. Enfin Section 4.4 différentes alternatives pour l'amélioration des performances du polymorphisme sont proposées et évaluées.

4.1 Conception du micro-benchmark

Le micro-benchmark implémenté pour mesurer les performances du polymorphisme possède le même squelette que le micro-benchmark générique présenté Chapitre 3 ; la méthode appelée `compute` y tient le même rôle à savoir l'encapsulation du code testé et la

mesure de sa durée d'exécution.
La méthode `compute` implémente ici le pattern *appel polymorphique dans une boucle*. La Figure 4.1 montre une version avec et sans timer de `compute`. Le pattern *appel polymorphique dans une boucle* est un exemple épuré d'utilisation intensive du polymorphisme. Les performances du polymorphisme sont définies comme la performance mesurées au travers de la méthode `compute`.
Ce pattern est omniprésent dans les applications Java mais ne ressort pas nécessairement avec autant de clarté dans le code applicatif. Le site d'appel peut être entrelacé avec d'autres opérations ou bien encapsulé dans une méthode inlinée.
Le bon choix du contenu de la méthode virtuelle `method` ciblée par le site d'appel est

```
static void compute(AbstractType[] data){
    int n = data.length;
    for (int i = 0; i < n; ++i)
        data[i].method();
}
```

Version sans timers

```
static long compute(AbstractType[] data){
    long t0 = System.nanoTime();
    int n = data.length;
    for (int i = 0; i < n; ++i)
        data[i].method();
    long t1 = System.nanoTime();
    long dt = (t1 - t0);
    return dt;
}
```

Version avec timers

FIG. 4.1: Méthode `compute` implémentant le pattern *appel polymorphique dans une boucle*. La seconde version est la version avec timer mesurant la durée d'exécution servant au calcul de la performance `n/dt`

fondamental afin de, un, garantir la généricité du benchmark (i.e. son indépendance vis à vis d'un traitement spécifique), deux, se placer dans un cas d'utilisation du polymorphisme critique et, trois, avoir une mesure qui soit juste. Une description détaillée des méthodes intervenant est faite Section 4.1.1.

4.1.1 Contenu du code mesuré

Cette section détaille le contenu des méthode impliquées dans le micro-benchmark (montré Figure 4.1) à savoir la méthode `method` Section 4.1.1.1 et la méthode `compute` Section 4.1.1.2.

4.1.1.1 Méthodes virtuelles ciblées

Dans ce type de pattern, l'utilisation du polymorphisme ne peut être remise en question que si le coût relatif de l'appel n'est pas négligeable par rapport à celui des méthodes ciblées. Ainsi une condition suffisante[1] pour que le polymorphisme soit problématique est que le coût d'appel ne soit pas entièrement masqué par le coût des méthodes ciblées[2]. Pour satisfaire cette condition, on se place dans un cas extrême où toutes les versions de la méthode `method` sont vides. Le coût relatif de l'appel par rapport aux méthodes appelées est alors théoriquement maximal.
Une des conséquences de l'utilisation de méthodes vides est que l'inlining doit être désactivé. Autrement, la boucle et le site d'appel dans `compute` seraient éliminés à la compilation et la mesure perdrait son sens. On peut considérer que la désactivation de l'inlining n'est pas représentative d'un cas ou le polymorphisme est problématique, car les méthodes ciblées (supposées de faible complexité relativement au site d'appel), seraient probablement inlinées par le JIT. Ainsi, pour que le benchmark soit représentatif, on doit s'assurer que le site d'appel avec ou sans inlining est similaire. Cette similarité signifie que le site d'appel sans inlining est équivalent au site d'appel avec inlining excepté que le corps des méthodes inlinées est remplacé par une instruction d'appel ; le dispatch demeurant similaire. Cette hypothèse est vraie, excepté dans le cas bimorphique esposé plus tard où l'inlining peut modifier l'état du site d'appel (cf. Figure 4.9).
Une autre approche peut être plus judicieuse, aurait été d'effectuer les tests en activant l'inlining mais avec des méthodes cibles effectuant une opération élémentaire comme incrémenter un champ de classe statique pour éviter l'élimination de code mort. L'avantage étant que le coût de la méthode aurait été moins important. En effet, une méthode vide contient des opérations constantes comprenant la création et la restauration du contexte sur la pile ainsi qu'une barrière de synchronisation (ou safepoint, cf. Section 2.3.1.4) avant retour. Le contenu d'une méthode vide est montré Figure 4.2.

```
mov   %eax,-max_offset(%rsp) # verifie le depassement de pile
push  %rbp                   # sauve le frame pointer
sub   $frame_size,%rsp       # alloue la frame
...
add   $frame_size,%rsp       # desalloue la frame
pop   %rbp                   # restore le frame pointer
test  %eax,0xb5ea71f(%rip)   # safepoint
ret
```

FIG. 4.2: Code généré par C2 pour une méthode d'instance vide

[1] Cette condition est suffisante car les problèmes de prédiction de branchement dues au polymorphisme persistent indépendamment de cette condition.
[2] Si il y a une grande disparité entre les différents coût des méthodes ciblées, il est judicieux de considérer un coût moyen

4.1.1.2 Méthode contenant les timers

La méthode `compute` itère sur un tableau d'objets de type `AbstractType` passé en argument (`data`). Le type `AbstractType` désigne soit une interface, dans ce cas le bytecode d'appel correspondant est `invokeinterface`, ou bien une classe et dans ce cas le bytecode d'appel correspondant est `invokevirtual`.
A chaque itération, un objet est chargé depuis le tableau `data` et la méthode `method` est appelée avec cet objet comme receveur. Les performances sont mesurées en Mop/s (méga-opérations par seconde) ou une opération désigne l'exécution du site d'appel. Le temps d'exécution des appels est obtenu en ajoutant des timers en entrée et sortie de méthode (cf. Section 3.3).
La méthode `compute` est un exemple d'utilisation intensive du polymorphisme, l'appel polymorphique étant la seule opération effectuée dans le corps de boucle.
Le paramètre `data` est utilisé pour faire varier la distribution des types qui est la principale variable dont dépend la performance comme détaillé Section 4.1.2.

Opérations mesurées par `compute` L'opération d'intérêt mesurée par `compute` est la durée d'exécution du site d'appel. Les autres opérations comme la durée d'exécution de la méthode ciblée sont considérée comme du bruit. L'objectif de l'implémentation est de maintenir l'intensité de ce bruit minimal, on se place ainsi and le cas extrême où le coût du polymorphisme (i.e. du site d'appel) est maximal. Cette approche permet d'obtenir une borne supérieure concernant l'impact d'une optimisation ou d'un paramètre sur le coût du polymorphisme.
Parmi ces opérations, on a vu dans la section précédente le contenu des méthodes ciblées qui sont maintenues vides pour limiter leur contribution au bruit. Les autres opérations s'ajoutant au bruit sont sont : le chargement du receveur depuis la mémoire ; le coût de la boucle et des timers ; des opérations de déchargement (*spilling*) et de restauration (*filling*) des registres de part et d'autre du site d'appel. Le coût de la boucle qui correspond à un test et un branchement à chaque itération est considéré négligeable. Il peut être amorti en déroulant la boucle (ce qui n'est pas effectué par le JIT pour des boucles contenant des appels de méthode). Le coût des timers étant constant, il est amorti par la taille de la boucle (cf. Section 3.3.3, Chap. 3). Pour réduire le coût des accès mémoires, la taille de `data` est fixée à 2048 bytes afin qu'il puisse loger dans le cache L1d. Sa taille doit par ailleurs être assez grande pour qu'une distribution de type aléatoire puisse mettre à mal la prédiction de branchement. La localité des données est augmentée en allouant un unique objet par type considéré et en réutilisant sa référence dans `data`.

4.1.2 Paramètres impactant les performances

Un des enjeux du benchmark est de mesurer l'impact de différents paramètres sur les performances du polymorphisme. Les deux paramètres dont la performance résulte sont :

- L'état du site d'appel i.e. le code généré par C2 pour les bytecodes `invokevirtual` ou `invokeinterface` ;
- La prédiction de branchements gérée au niveau matériel.

L'état du site d'appel et de la prédiction de branchement dépendent tout deux de la **distribution** des types dans `data` i.e. les différents types mis en jeux et leur ordonnancement (cf. Section 4.3.1.1) qui caractérise le jeu de données d'entrée. La Figure 4.3 montre la chaîne de dépendance entre la performance résultante et les différentes variables impliquées.
Dans la suite, on dit que la distribution X est exécutée pour dire que le micro-benchmark est lancé avec la distribution X en entrée. L'état du code, plus précisément du site d'appel, dépend de la distribution exécutée lors de la phase de profiling. Lorsque l'appel est polymorphique, il dépend en plus de la nature de ce dernier à savoir `invokevirtual` ou `invokeinterface` (paramètre **Application**). Les différents états du site d'appel sont analysés Section 4.2.

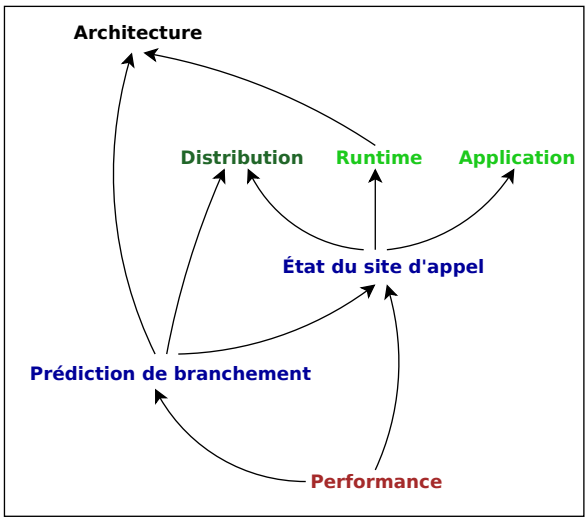


FIG. 4.3: Lien de dépendance entre la performance résultante et les différentes variables. La principale variable considérée pour les tests de performance est **Distribution** à savoir la distribution des types en entrée qui affecte à la fois l'état du site d'appel et la prédiction de branchement. Le paramètre **Application** correspond au code applicatif et est donc modifié lorsque le code source l'est (avec du dispatch explicite par exemple). Le paramètre **Runtime** est quant à lui modifié lorsque la configuration de HotSpot est modifiée. Ces deux paramètres impact l'état du site d'appel. Les paramètres **Architecture** et **Prédiction de branchement** sont fixés

La prédiction de branchement est gérée par le CPU (cf. Section 2.1.1.1). Ni le compilateur, ni le développeur n'ont donc la main dessus. Des techniques avancées d'optimisation

[107] de code existent mais requièrent une connaissance poussée du fonctionnement interne de l'unité de prédiction de branchement et varie en fonction des architectures. Elles sont donc très délicates à mettre en œuvre, à la fois côté développeur Java mais aussi côté développeur du runtime Java.
Sans connaître son fonctionnement interne, on s'attend néanmoins à ce que la prédiction de branchement soit efficace pour des distributions de nature prédictible, comme les distributions périodiques, et inefficace dans le cas de distributions aléatoires ou pseudo-aléatoires. Ces deux types de distribution sont utilisées dans les tests de performance. Les optimisations de C2 reposent sur le compromis suivant : favoriser les appels directs (notamment pour bénéficier de l'inlining) mais limiter le nombre de branchements (i.e. le coût du dispatch) avec l'utilisation d'appel indirect (via une table de méthodes).

4.2 États d'un site d'appel polymorphique

Le code généré par le compilateur C2 pour un site d'appel peut avoir différents états. Ces états sont générés selon l'information profilée (cf. Section 3.3.2) afin d'obtenir les meilleures performances. Deux types de nœuds au niveau de la représentation intermédiaire de C2 sont utilisés par le compilateur pour émettre le code binaire. Le nœud `CallStaticJavaNode` émet un appel dont la cible est connue à la compilation. Le nœud `CallDynamiqueJavaNode` émet un dispatch via une table de méthode lorsque la cible dépend du type du receveur. Ces nœuds sont combinés afin de produire le meilleur site d'appel possible.

La Section 4.2.1 détaille les différents états d'un site d'appel polymorphique pouvant être générés par C2 puis la Section 4.2.2 analyse le code binaire correspondant à chacun de ces états.

4.2.1 États et transitions

Le benchmark implémenté (décrit Section 4.1) a permis d'identifier les différents états possibles du site d'appel générés par C2 en jouant sur les différents paramètres impactant l'état du code (cf. Figure 4.3). Ces différents états sont les suivants :

1. **Monomorphique-AHC** (généré seulement pour les appels virtuels) ;
2. **Monomorphique** ;
3. **Bimorphique** ;
4. **Polymorphique-domination**[3] ;
5. Etats polymorphiques[3] : **polymorphique-CI**, **dispatch-virtuel** ou **dispatch-d'interface**.

[3]Polymorphique, signifiant "plusieurs formes", inclut normalement le cas particulier de deux formes à savoir bimorphique. Dans cette étude le terme polymorphique exlus le cas bimorphique comme ce dernier est un cas spécifique. Pour lever cette ambiguïté le terme mégamorphique est parfois utilisé à la place de polymorphique pour signifier "strictement plus de deux formes".

L'état 1 utilise l'AHC (Analyse de Hiérarchie de Classe) [31] pour déterminer si le site d'appel est monomorphique. C'est le cas si une unique implémentation de la méthode virtuelle est chargée par le chargeur de classe[4].
Les états 2, 3, 4 sont des états optimisés utilisant les résultats du profiling sur le type du receveur. Plus précisément, l'état 2 survient lorsque l'AHC a déterminé plusieurs méthodes cibles potentielles mais que le profiling ne montre qu'un seul type de receveur impliqué au niveau du site d'appel. L'état 3 survient lorsque le profiling a détecté deux types de receveur impliqués et l'état 4 lorsqu'il a détecté plus de deux types impliqués, mais que l'un domine en terme de fréquence. Le profiling du type du receveur est activé par défaut et contrôlé via le paramètre `-XX:UseTypeProfile`. Contrairement à l'AHC qui est globale à tous les sites d'appels, le profiling est local à chacun d'entre eux.
L'état 5 est l'état polymorphique généré lorsque le profiling ne peut optimiser le site d'appel ou bien qu'il est désactivé. Il possède deux sous-états. Un premier, polymorphique-CI, utilisant un cache d'inline (CI) [71] et un second utilisant du dispatch et qui diffère dans le cas d'un appel virtuel (dispatch-virtuel) ou d'un appel d'interface (dispatch-d'interface).

4.2.1.1 États finaux

Les états finaux sont les états pour lesquels l'exécution ne déclenche pas de transition vers un autre état[5]. La Figure 4.4 illustre les différents états et leurs transitions. L'état interprété est l'état initial. Les transitions après compilation vers l'état interprété surviennent lorsque le code du site d'appel est invalidé. Il y a deux raisons qui causent l'invalidation du site d'appel, soit par désoptimisations successives lorsque le code utilise les hypothèses de profiling, soit par le chargeur de classe lorsque le code est généré par AHC.
Le diagramme état-transition est considéré en mode non-étagé (cf. Section 3.2.2). Dans ce cas le profiling a lieu uniquement en mode interpréteur et l'utilisation des informations se fait à la compilation du code par C2. En mode étagé, des états intermédiaires entre les états compilés et interpréteur apparaîtraient avant d'atteindre les états compilés par C2 (cf. Section 2.3.2.3 Chap. 2). On admet que pour l'exécution d'une même distribution le code résultant après warmup en non-étagé et étagé est identique.

4.2.1.2 Prise en compte de l'inlining

Comme justifié dans la mise en œuvre du benchmark mesurant les performances du polymorphisme (cf. Section 4.1), l'inlining n'est pas activé pour les tests. Les états considérés et énumérés précédemment sont donc ceux obtenus lorsque l'inlining est désactivé. Les sites d'appel générés lorsque l'inlining est désactivé sont cependant similaires à ceux lorsque ce dernier est activé. Cette similarité signifie que le site d'appel sans inlining est

[4]Notons que l'AHC fonctionne dans ce cas par méthode et non par classe. Plus précisément si une classe fille est chargée mais qu'elle ne surcharge pas la méthode

[5]Une méthode contenant un site d'appel dans un état final peut néanmoins être invalidée pour d'autres raisons que le site d'appel

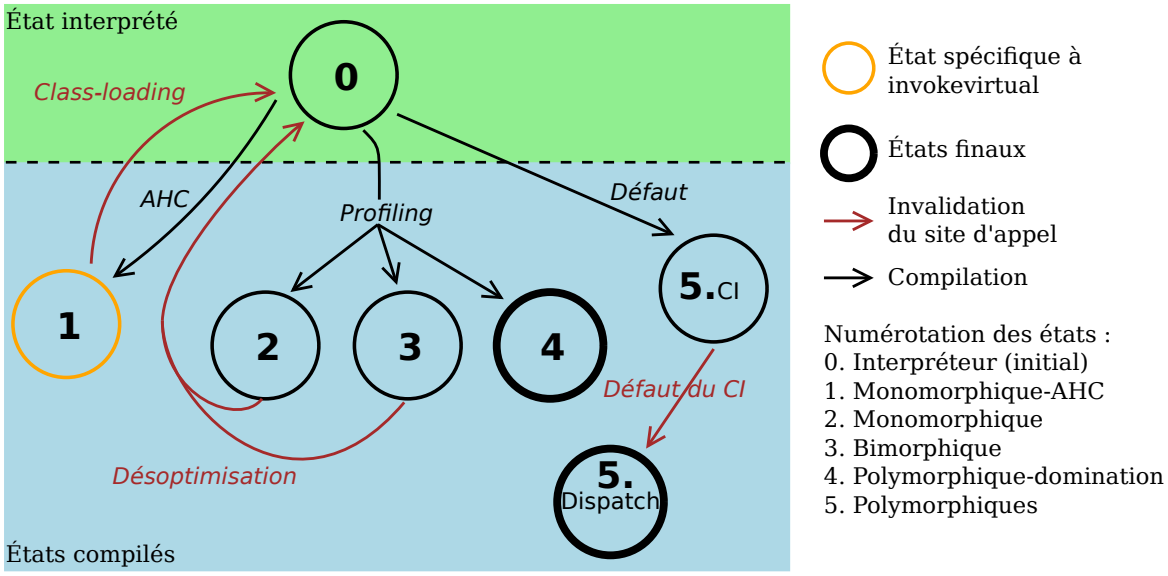


FIG. 4.4: Diagramme des états/transitions d'un site d'appel `invokevirtual` ou `invokeinterface` en mode non-étagé

équivalent au site d'appel avec inlining excepté que le corps des méthodes inlinées est remplacé par l'instruction d'appel de la méthode. La Table 4.1 résume l'effet de l'inlining sur le site d'appel lorsqu'il est activé. Considérant uniquement le coût des méthodes à vide (cf. Figure 4.2), l'inlining s'avère crucial pour garantir de bonne performance pour des appels de méthodes de faible complexité ; indépendamment des optimisations de contexte, en considérant uniquement l'élimination coup d'appel à vide. Ainsi l'écart de performance entre les sites d'appel avec inlining et sans doit être plus important dans un benchmark ou l'inlining est activé.

État du site d'appel	**Effet de l'inlining**
Monomorphique-AHC Monomorphique	Le JIT tente d'inliner l'unique méthode rencontrée
Bimorphique	Les JIT tente d'inliner les deux méthodes rencontrées
Polymorphique-domination	Le JIT tente d'inliner la méthode associée au type dominant
États polymorphiques (avec CI ou dispatch)	Aucun effet

TAB. 4.1: Effet de l'inlining sur les sites d'appel lorsqu'il est activé. En **vert** les états bénéficiant de l'inlining et en **rouge** les états invariant avec l'activation de l'inlining. Le code généré pour les sites d'appels inlinés diffère par le fait que l'instruction d'appel de méthode est remplacée par le corps de méthode ; le dispatch étant inchangé

4.2.2 Analyse de code des différents états

Pour comprendre le fonctionnement interne du polymorphisme et les résultats du benchmark, le code machine correspondant à chaque état généré par C2 est analysé puis décrit dans les sections suivantes. La section code assembleur, extraite et commentée provient de la compilation de la méthode `caller` (cf. Figure 4.5) qui encapsule uniquement le bytecode d'appel.
L'état du code reste cependant dépendant du contexte, en particulier au niveau des registres utilisés, la nature des instructions restant similaire.

```
static void caller(AbstractType rcv){
    rcv.method();
}
```

FIG. 4.5: Méthode `caller` implémentée pour analyser le code assembleur d'un site d'appel

4.2.2.1 Considérations préalables

La syntaxe assembleur est celle utilisée par défaut par le dés-assembleur de la JVM HotSpot, à savoir la syntaxe AT&T[6]. Les adresses absolues utilisées dans le code sont remplacées, pour plus de clarté, par des symboles (ou label) dont le nom est explicite. Les constantes numériques sont conservées dans leurs bases décimales ou hexadécimales (dans ce cas la valeur est préfixée par `0x`). L'ordre en mémoire des instructions est conservé, cependant les instructions utilisées pour l'alignement (e.g. `nop`) sont supprimées.

Quelques pré-requis concernant les structures internes de la JVM sont nécessaires à la bonne compréhension du code. Ceux-ci sont exposées Section 2.3.1.2 Chap. 2 et rappelés ci-dessous. Chaque objet Java possède dans son entête un champ nommé *klass word* qui contient un oop appelé pointeur de classe et noté `instanceKlass*` qui définit le type de l'objet. Ainsi deux objets ayant la même valeur de pointeur de classe ont un type identique. Cet oop pointe vers une instance de la classe `instanceKlass` correspondant au type de l'objet et stockée dans la génération permanente de la heap. Elle contient toutes les informations sur le type de l'objet requises pour l'exécution de l'application, en particulier la table des méthodes.
Dans les commentaires de code, `rcv` désigne l'oop du receveur et `rcvKlass` désigne son pointeur de classe. Plus généralement, les symboles finissant par `Klass` désignent des pointeurs de classe. Par volonté de simplification, le symbole d'un pointeur de classe reste inchangée dans sa forme compressée et non-compressée (`rcvKlass` est utilisé pour désigner à la fois le pointeur de classe du receveur dans sa forme compressé et décompressé).
Certains états du code ayant fait l'objet d'optimisations optimistes possèdent une branche

[6]La syntaxe Intel peut être utilisée via le paramètre `-XX:PrintAssemblyOptions=intel` de HotSpot

exceptionnelle de désoptimisation. Cette dernière est désignée par le label `uncommon_trap` dans le code assembleur. Elle est exécutée dans des cas jugés rares par le profiling et peut entraîner une invalidation du site d'appel et donc de la méthode compilée contenant le site d'appel (cf. Section 3.3.2).

4.2.2.2 État monomorphique-AHC

Différents cas spécifiques, n'entrant pas dans le cadre du benchmark, permettent au compilateur de déterminer de manière certaine que le site d'appel est monomorphique. Par exemple si le type concret du receveur est connu et invariant à la compilation (c'est notamment le cas si l'allocation de l'objet à lieu dans le même contexte que l'appel ou encore si le receveur est `this` dans le cas d'appels imbriqués). Ou encore si le receveur est un tableau qui ne peut donc pas surcharger certaines méthodes virtuelles, comme les méthodes de la classe `Object`. En dehors de ces cas spécifiques, le compilateur utilise l'AHC pour déterminer si le site d'appel est monomorphique. C'est en particulier le cas si la méthode appelée ou le type du receveur sont déclarés `final`.
L'état monomorphique-AHC est exclusive à `invokevirtual` et n'est pas généré pour les appels d'interfaces `invokeinterface` [124]. L'AHC connaît l'arbre de hiérarchie de classe à un instant donné. Cette hiérarchie peut évoluer au cours de l'exécution. Le compilateur enregistre donc une dépendance au niveau du chargeur de classe. Le code de la méthode contenant le site d'appel optimisé sera invalidé (i.e. rendu non entrant) par le chargeur de classe si ce dernier charge une classe surchargeant la méthode appelée. L'état futur du site d'appel est alors déterminé par le profiling (cf. Figure 4.4).
La section de code assembleur générée par le JIT correspondant à l'état monomorphique-AHC est commenté Figure 4.6. Contrairement à un appel statique, ce dernier nécessite un *null-check* explicite à savoir un test explicite pour savoir si le pointeur du receveur, désigné par `rcv`, vaut `null` ; auquel cas, le code jette l'exception `NullPointerException` (repéré par l'adresse portant le label `exception` dans le code). Sinon, l'unique méthode ciblée est appelée[7].

```
                    # rcv est dans r11d
test   %r11d,%r11d # Teste rcv
je     exception    # Branche vers l'exception si rcv est nul
call   method       # Appel la methode si rcv non nul
```

FIG. 4.6: Section d'assembleur généré par C2 pour l'état monomorphique-AHC

[7]La méthode est appelée à son point d'entrée vérifié ou VEP (*Verified Entry Point*) par opposition au point d'entrée non vérifié ou UEP (*Unverified Entry Point*) utilisé dans l'état polymorphique-CI

4.2.2.3 État monomorphique

Lorsque le profiling ne dénombre qu'un seul type de receveur impliqué au niveau du site d'appel, le code généré par C2 sera dans l'état monomorphique. Le code généré par C2 pour cet état est détaillé Figure 4.7. Il fait l'hypothèse que le receveur sera du même type que celui profilé et teste donc l'égalité des types. Si le test réussie, la méthode attendue (`type1Klass.method`) est appelée. Si le test échoue, le code branche vers `uncommon_trap` où la méthode est désoptimisée. Le test d'égalité fait intervenir le pointeur de classe du receveur (`rcvKlass`) situé à un décalage constant (8 bytes) du pointeur du receveur (`rcv`). Le pointeur du type profilé (`type1Klass`) est une constante et un opérande immédiat dans le code généré. Contrairement à l'état monomorphique-AHC qui effectue un test explicite pour savoir si le receveur est nul, ici le teste est implicite. Il a lieu au moment du chargement de `rcvKlass` dans un registre, la JVM intercepte alors un signal matériel émit si `rcv` est nul puis jette l'exception `NullPointerException`. Cet état n'est pas un état final, les des-optimisations successives au niveau du site d'appel déclenchent une invalidation de la méthode appelante, qui sera ensuite interprétée, jusqu'à une nouvelle compilation et un nouvel état du site d'appel (cf. Figure 4.4).

```
                        # rcv est dans r10
mov 0x8(%r10),%r8d      # Charge rcvKlass dans r8d, null-check implicite
cmp type1Klass,%r8d     # Compare type1Klass et rcvKlass
jne uncommon_trap       # Desoptimise s'ils sont non-egaux
call  type1Klass.method           # Appel la methode attendue sinon
```

FIG. 4.7: Section d'assembleur généré par C2 pour l'état monomorphique

4.2.2.4 État bimorphique

Lorsque le profiling dénombre deux types de receveur impliqués au niveau du site d'appel le code par C2 sera dans l'état bimorphique[8]. L'activation ou non de cet état est contrôlée par le paramètre `-XX:UseBimorphicInlining` de HotSpot. La section de code générée par C2 pour cet état est détaillée Figure 4.8. Tout comme pour l'état monomorphique, `rcv` désigne le pointeur du receveur, et `rcvKlass` le pointeur vers la classe du receveur situé à un décalage constant de `rcv`. `type1Klass` désigne le pointeur de classe du type le plus fréquemment rencontré et `type2Klass` le pointeur de classe du second type rencontré. Le code teste en premier lieu si le type du receveur est égal au type le plus fréquemment rencontré. Si tel est le cas, le code branche vers l'adresse de label `rcvType1` qui appellera la version de la méthode correspondante (`type1Klass.method`). Si le premier test échoue, le code teste dans un second temps si le type du receveur est égal au second type rencontré. Si l'égalité est fausse, le code branche vers `uncommon_trap` et la méthode est des-optimisée. Si l'égalité est vrai, alors la méthode du second type (`type2Klass.method`) est appelée.

[8]Comme expliqué plus bas, en supposant que l'option `-XX:UseOnlyInlinedBimorphic` est désactivée.

Cet état n'est pas un état final, les des-optimisations successives au niveau du site d'appel déclenchent une invalidation de la méthode appelante, qui sera ensuite interprétée, jusqu'à une nouvelle compilation et un nouvel état du site d'appel.

Lorsque deux types sont profilés, la génération de l'état bimorphique dépend des décisions

```
                        # rcv est dans rsi
mov 0x8(%rsi),%r10d     # Charge rcvKlass dans r10d, null-check implicite
cmp type1Klass,%r10d    # Compare type1Klass avec rcvKlass
je  rcvType1            # Branche vers rcvType1 s'ils sont egaux
cmp type2Klass,%r10d    # Compare type2Klass avec rcvKlass
jne uncommon_trap       # Branche vers uncommon_trap si non-egaux
call  type2Klass.method  # Appel type2Klass.method
...
rcvType1:
call   type1Klass.method # Appel type1Klass.method
```

FIG. 4.8: Section d'assembleur généré par C2 pour l'état bimorphique

d'inlining. Lorsque l'option `-XX:UseOnlyInlinedBimorphic` est désactivée (ce qui n'est pas le cas par défaut), l'état généré est toujours l'état bimorphique. Par défaut, l'état généré peut être soit dans l'état polymorphique-domination, soit dans l'état bimorphique. La Figure 4.9 illustre les décisions du JIT. Lorsque l'inlining est activé et qu'au moins l'une des deux méthodes rencontrées lors du profiling est inlinée, le site d'appel sera dans l'état bimorphique. Si aucunes des deux méthodes n'est inlinées (ce qui est le cas lorsque l'inlining est désactivé), le code sera soit dans l'état polymorphique-domination si l'un des deux types domine soit dans l'état bimorphique sinon.

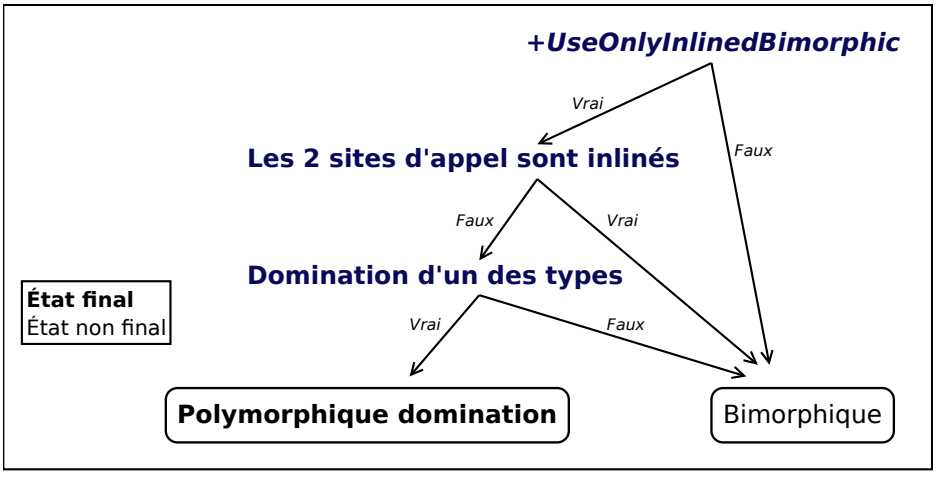


FIG. 4.9: Arbre de décision du JIT lorsque l'interpréteur exécute une distribution bimorphique au niveau du site d'appel (i.e. le profiling détecte deux types impliqués)

4.2.2.5 État polymorphique-domination

Lorsque le profiling dénombre plus de deux types de receveur[9] impliqués au niveau du site d'appel, et que la fréquence d'apparition d'un type est supérieure à un certain seuil, le code généré par C2 est dans l'état polymorphique-domination. Ce seuil de fréquence est déterminé par le paramètre `-XX:TypeProfileMajorReceiverPercent` de HotSpot dont la valeur par défaut est 90. Dans ce cas, cet état est généré si un type survient avec une fréquence supérieure ou égale à 90%. Le code correspondant à cet état est similaire à celui de l'état monomorphique excepté que, si le type du receveur est différent du type dominant (dont le pointeur de classe constant est désigné `domKlass`), le code branche vers un site d'appel polymorphique (repéré par le label `dispatchStub`). L'état polymorphique-domination est un état final.

```
                        # rcv est dans rsi
mov   0x8(%rsi),%r11d   # Charge rcvKlass dans r11d, null-check implicite
cmp   domKlass,%r11d    # Compare domKlass et rcvKlass
jne   dispatchStub      # Branche vers dispatch si different
call   domKlass.method  # Appel domKlass.method
```

FIG. 4.10: Section d'assembleur généré par C2 pour l'état polymorphique-domination

4.2.2.6 États polymorphiques

Lorsque le profiling dénombre strictement plus de deux types de receveur impliqués au niveau du site d'appel, mais qu'aucun type n'est dominant au niveau des fréquences, le code généré par C2 est dans l'état polymorphique. Les différents états possibles lors de l'exécution sont :

- polymorphique-CI : état polymorphique utilisant un cache d'inline ;
- dispatch-virtuel (dans le cas d'un appel virtuel `invokevirtual`) ou dispatch-d'interface (dans le cas d'un appel d'interface `invokeinterface`) : état polymorphique utilisant du dispatch.

Polymorphique-CI L'état polymorphique-CI correspond à l'état initial d'un site d'appel polymorphique. Le code associé est détaillé Figure 4.11. Une fois le code généré par C2, le site d'appel cible le runtime. À la première exécution du site d'appel, le runtime inscrit le pointeur de classe du premier receveur rencontré (nommé `1stKlass`) dans le CI puis modifie la cible de la méthode vers la méthode correspondant à ce receveur. `1stKlass` est donc inscrit dans le CI (qui désigne l'opérande ré-inscriptible dans le code binaire), puis est chargé dans le registre dédié `rax`. L'appel cible ensuite l'UEP (*Unverified Entry Point*) de la méthode associée au type inscrit dans le CI (notée `1st.method`). Chaque méthode

[9] Lorsque l'option `-XX:UseOnlyInlinedBimorphic` est activée, cet état peut aussi être généré si deux types sont impliqués mais que l'un domine (voir état bimorphique).

d'instance possède un UEP précédent son point d'entrée normal. Ce point d'entrée est spécialement prévu pour l'état avec CI. Il vérifie préalablement que le pointeur de classe correspondant au type du receveur (`rcvKlass`) est identique à celui inscrit dans le CI (`1stKlass`). Si tel est le cas, le code poursuit son exécution vers le point d'entrée de la méthode. Sinon, le code appel le runtime qui modifie le site d'appel vers l'état de dispatch (cf. Figure 4.4).
Pour que l'état avec CI soit bénéfique, il faut que le nombre de receveurs successifs, ayant le même type que le premier receveur rencontré après la compilation, soit suffisant. Autrement dis, il faut que le site d'appel exécute une distribution monomorphique post-compilation et une distribution polymorphique pré-compilation. C'est le cas si la méthode contenant le site d'appel possède plusieurs contextes d'appel rendant le site d'appel polymorphique mais est inlinée dans un contexte ou elle exécutera une distribution monomorphique.

```
                        # rcv est dans rsi
movabs 1stKlass,%rax    # Charge 1stKlass dans rax
call   1st.method.UEP   # Appel l'UEP de 1st.method
...
1st.method.UEP:
mov  0x8(%rsi),%r10d # Charge rcvKlass dans r10d, null-check implicite
shl  $0x3,%r10       # Decompresse rcvKlass dans r10
cmp  %r10,%rax       # Compare rcvKlass et 1stKlass
jne  runtime         # Branche vers le runtime si non egaux
1st.method:          # Entre dans 1st.methode si egaux
```

FIG. 4.11: Section d'assembleur généré par C2 correspondant l'état polymorphique-CI

États polymorphiques avec dispatch L'état polymorphique avec dispatch diffère selon la nature de l'appel, à savoir un appel virtuel `invokevirtual` ou bien appel d'interface `invokeinterface`. Cette différence provient du fait qu'une classe ne peut hériter que d'une seule super-classe, alors qu'elle peut hériter de plusieurs interfaces (cf. Section 2.2.4 Chap. 2).

Dispatch-virtuel Le code correspondant à l'état dispatch-virtuel est commenté Figure 4.12. Dans cet état, le CI n'est pas utilisé et contient l'entier signé -1. L'appel cible alors la section de code `vtableStub` qui effectue le dispatch et branche vers la bonne méthode. Le dispatch s'effectue en utilisant la table de méthodes virtuelles ou *vtable*. La *vtable* est située à un décalage constant dans la structure `instanceKlass` quelque soit la classe. Ses entrées sont des pointeurs de méthode (`method*`). Chaque méthode non-statique d'une classe possède une entrée unique dans cette table. La *vtable* d'une classe étend la *vtable* de sa superclasse en ajoutant de nouvelles entrées à la suite. Ainsi une méthode virtuelle est repérée par un indice unique, indépendamment du type qui l'implémente, qui correspond à son entrée dans la *vtable*. Plus largement, comme la *vtable* est situé à décalage constant et que l'index de son entrée est constant, le décalage vers le pointeur de méthode

est également constant. Dans l'exemple donné, le décalage vers le pointeur de méthode vaut 480 bytes (`0x1e0` en hexadécimal). Une fois le pointeur de méthode chargé, le code branche vers le point d'entrée de la méthode. Ce dernier est chargé depuis le champ nommé `from_compiled`, situé dans la structure `method` à un décalage constant de 64 bytes (`0x40` en hexadécimal), et qui contient l'entrée de la méthode à appeler depuis du code compilé[10].

```
                          # rcv est dans rsi
movabs $-1,%rax           # Cache d'inline inactif
call vtableStub           # Appel vtableStub
...
vtableStub:
mov    0x8(%rsi),%eax     # Charge rcvKlass dans eax
shl    $0x3,%rax          # Decompresse rcvKlass dans rax
mov    0x1e0(%rax),%rbx   # Charge le pointeur de methode dans rbx
jmp    *0x40(%rbx)        # Branche vers l'entree de la methode
```

FIG. 4.12: Section d'assembleur généré par C2 pour l'état dispatch-virtuel

Dispatch-d'interface Le code correspondant à l'état dispatch-d'interface est commenté Figure 4.13. Le pointeur d'interface est chargé dans un registre puis l'appel cible la section de code runtime `itableStub` chargée d'effectuer le dispatch. Tout comme pour une méthode virtuelle, une méthode d'interface correspond à un index unique dans la *vtable* de l'interface. La section `itableStub` est constante pour un index donné. La *vtable* de l'interface est identique pour toutes les classes implémentant l'interface. Cependant, à la différence d'un appel de méthode virtuelle avec dispatch, une classe peut hériter de plusieurs interfaces. Ainsi l'emplacement de la *vtable* de l'interface dans la structure `instanceKlass` de la classe du receveur n'est pas constant et doit être trouvé. Pour cela l'`instanceKlass` contient une table d'interfaces, ou *itable*, dont le nombre d'entrée correspond au nombre d'interfaces qu'elle implémente.
La *itable* est écrite en mémoire après la *vtable* de la classe. Comme cette dernière est de taille variable en fonction du type du receveur, sa taille est inscrite dans un champ (`vtableSize`) dont le décalage dans l'`instanceKlass` est constant. Cette dernière est ainsi chargée et utilisée pour localiser l'adresse de la *itable* dans l'`instanceKlass` (notée `itable*`).
Une entrée de la *itable* contient deux mots de 8 bytes. Le premier mot (i.e. l'index) de type `instanceKlass*` permet de déterminer à quelle interface l'entrée correspond. Le second mot contient le décalage en bytes auquel est situé sa *vtable* dans l'`instanceKlass`.
Pour trouver l'entrée de l'interface dans la *itable*, l'`instanceKlass*` de l'interface qui émet l'appel (`intKlass`) est écrit dans le CI. En premier lieu, l'interface contenu dans la pre-

[10]Ce champ contient soit le point d'entrée de la méthode compilée, soit un adaptateur (`comp_2_int`) permettant de basculer vers l'interpréteur. De manière similaire, un champ `from_interpreted` contient l'entrée de la méthode lorsque l'appelant est l'interpréteur, qui peut être un adaptateur vers du code compilé (`int_to_comp`).

mière entrée est comparée avec `intKlass` contenu dans `rax`. S'il s'agit de la bonne entrée, le code branche vers `found`. Sinon le code rentre dans `lookup` et itère sur le reste des entrées tant que `intKlass` n'est pas trouvée. Un index nul signal la fin de la *itable*. Si l'entrée est trouvée, l'adresse de la *vtable* (`intVtable`), située dans le second mot, est chargée et le dispatch à lieu comme pour un appel virtuel. Si cette dernière n'est pas trouvée (i.e. si l'index nul) le code branche vers `notFound` et lève une exception.
Le code contenant une boucle sur la *itable*, la performance du dispatch dépend donc de l'index de l'interface dans la *itable*. Le développeur peut contrôler sa valeur puisqu'il correspond à l'ordre de définition des interfaces implémentées dans le prototype de la classe.

```
                        # rcv est dans rsi
mov  intKlass,%rax      # Charge intKlass dans rax
call itableStub         # Appel itableStub
...
itableStub:
mov    0x8(%rsi),%r10d   # Charge rcvKlass dans r10d
shl    $0x3,%r10         # Decompresse rcvKlass dans r10
mov    0x120(%r10),%r11d        # Charge vtableSize dans r11d
lea    0x1b8(%r10,%r11,8),%r11d # Charge itable* dans r11d (pointeur d'entree)
lea    0x8(%r10),%r10    # Ajout du decalage de la method
mov    (%r11),%rbx       # Charge la 1ere entree dans rbx
cmp    %rbx,%rax         # Test la 1ere entree
je     found
lookup:                  # Boucle sur les autres entrees
test   %rbx,%rbx         # Test l'index nul indiquant la fin de la itable
je     notFound          # Si vrai l'interface n'est pas trouvee
add    $0x10,%r11        # Incremente le pointeur vers l'entree suivante
mov    (%r11),%rbx
cmp    %rbx,%rax
jne    lookup
found:
mov    0x8(%r11),%r11d    # Charge intVtable dans r11d
mov    (%r10,%r11,1),%rbx # Charge la method dans rbx
jmp    *0x40(%rbx)        # Branche vers le point d'entree
...
notFound:
jmp    runtime      # Appel le runtime qui lance une erreur
```

FIG. 4.13: Section d'assembleur généré par C2 pour l'état dispatch-d'interface

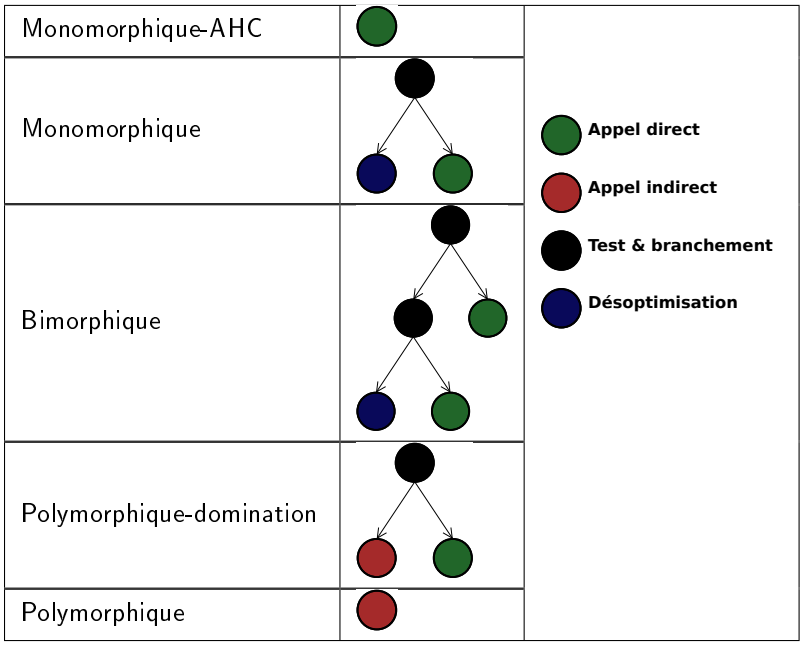
4.2.2.7 Résumé

Les différents états décrits dans les sections précédentes sont résumés Table 4.2 à l'aide d'arbres descriptifs de leur fonctionnement. Ces arbres se composent de 4 types de nœud :

- Les **tests/branchements** (en noir) qui testent le type du receveur puis branchent en fonction du résultat du test ;
- Les **appels directs** (en vert) qui peuvent bénéficier de l'inlining ;
- Les **appels indirects** (en rouge) qui passent par une table de méthode ;

- Les **nœuds de désoptimisation** (en bleu).

Les opérations de décompression des pointeurs[11] ou de null-check[12] ne sont pas représentées. L'état polymorphique-CI n'est pas non plus représenté, il est équivalent à l'état monomorphique excepté que le test est effectué au niveau de la méthode cible. Comme montré Figure 4.4, les états polymorphique-domination et polymorphique sont des états finaux qui ne possèdent donc pas de nœud de désoptimisation. Concernant l'état monomorphique-AHC, ce dernier n'est pas final mais ne possède pas de nœud de désoptimisation car dans son cas la désoptimisation est déclenchée par le chargeur de classe.



TAB. 4.2: Représentations abstraites des différents états générés par C2

4.3 Résultats et analyse des performances

Les résultats mesurés représentent la performance pour un état du site d'appel de la méthode `compute` (cf. Figure 4.1), exprimée en méga-opérations par seconde (Mop/s) ou une opération désigne l'exécution du site d'appel.
Les performances sont mesurées en fonction de la distribution des types dans `data`; on parle par la suite de la performance de l'exécution d'une distribution.
La taille de `data` (cf. Figure 4.1) fixée pour les tests est de 2048, ainsi la structure loge dans le cache L1d et `compute` possède suffisamment d'itérations pour masquer les coûts constants. La méthodologie de mesure est décrite Section 3.3.

[11] Afin d'accéder au klass-word
[12] Dans le cas de l'état monomorphique-AHC

4.3.1 Distributions et états stables

Cette section définit une distribution (cf. Section 4.3.1.1) ainsi que la notion d'état stable pour une distribution (cf. Section 4.3.1.2)

4.3.1.1 Distributions

Les possibilités de distribution sont infinies, deux types sont considérés ici : les distributions pseudo-aléatoires et les distributions périodiques. Une distribution est d'abord caractérisée par la **grandeur du polymorphisme** à savoir le nombre de type différents qui la compose. Pour 1 type, la distribution est monomorphique, pour 2, la distribution est bimorphique et pour N types, avec N strictement supérieur à 2, la distribution est N-polymorphique.
Les distributions aléatoires sont caractérisées par la fréquence d'apparition de chaque type (exprimée en pourcentage). Par exemple la distribution Al. 40-60 est une distribution bimorphique aléatoirement répartie pour laquelle la proportion du type 0 est d'environ 40% et celle du type 1 est d'environ 60%. Les distributions aléatoires sont obtenues grâce à la classe `Random` du JDK qui est un générateur congruentiel linéaire à partir d'une graine passée en paramètre.
Les distributions périodiques sont caractérisées par la taille des paquets, un paquet étant le nombre d'enchaînement d'un type dans une période, une période étant l'enchaînement des paquets de chaque type impliqué. La distribution Per. 1 désigne une distribution avec une taille de paquet de 1.
La Table 4.3 illustre différentes distributions.

Monomorphique	0000000000000000000000000000000000000000000
Bimorphique Al. 90-10	1001000000000000000000000000000000000100000
Bimorphique Per. 1	0101010101010101010101010101010101010101010
3-Polymorphique Per. 2	0011220011220011220011220011220011220011220001
4-Polymorphique Al. 25-25-25-25	3103101011311313223121212022313133112213103 10

TAB. 4.3: Exemples de distributions. Les types sont arbitrairement numérotés à partir de 0. Les 45 premières valeurs sont affichées. Les distributions aléatoires sont générées avec la classe `Random` avec une graine invariante selon les distributions

4.3.1.2 État stable pour une distribution

Les résultats regroupent les performances calculées pour différentes grandeurs du polymorphisme.
Les états sélectionnés pour les mesures sont ceux qualifiés de *stables*. Un état est *stable* pour une distribution donnée si son exécution sur cette distribution ne provoque pas de transition du site d'appel vers un autre état. Un état *stable* pour une distribution peut être un état *adapté* ou un état *final*.

Un état *adapté* à une distribution est un état dont le code est spécialisé pour cette distribution. Un état adapté suppose que les données collectées lors de la phase de profiling et utilisées pour la compilation décrivent cette distribution. En mode non-étagé, cela implique que l'interpréteur, chargé du profiling, a exécuté cette distribution.
Un état *final* pour un site d'appel est un état dont l'exécution ne provoque pas de transition vers un autre état (cf. Figure 3.8).
Par exemple, les états monomorphique, bimorphique, polymorphique-domination et polymorphique-CI sont des états stables d'une distribution monomorphique ; l'état monomorphique étant adapté, l'état polymorphique-domination final et les états polymorphique-CI et bimorphique simplement stables.
L'analyse des performances pour un état non-adapté permet de quantifier la diminution de performance lorsque la distribution sur laquelle est basée la compilation diffère de la distribution sur laquelle l'appel est exécuté après compilation.

4.3.2 Résultats d'exécution d'une distribution monomorphique

La Figure 4.14 montre les résultats de performance des états stables pour une distribution monomorphique. Tous les états du site d'appel générés par le JIT sont stables pour les distributions monomorphiques (cf. Figure 4.4).
Les états adaptés aux distributions monomorphiques sont bien gérés par le JIT comme le montre les faibles écarts entre l'état *statique* et les états adaptés. Les écarts sont résumés Table 4.4.
L'écart de performance entre l'état monomorphique-AHC et monomorphique s'explique par la présence d'un test sur le type du receveur dans le second cas (cf. Figure 4.7). Le léger écart entre l'état *statique* et l'état monomorphique-AHC provient lui, de la présence d'un *null-check* explicite dans le second cas (cf. Figure 4.6).
L'état bimorphique (cf. Section 4.2.2.4) a une performance similaire aux états adaptés lorsque le type composant la distribution correspond au premier test du dispatch (*première branche*, cf. Figure 4.14). Lorsque le type composant la distribution correspond au second type testé, le second branchement est emprunté (*seconde branche*, cf. Figure 4.14) ce qui entraîne un écart de performance de 12%.
De la même manière que l'état bimorphique, l'état polymorphique-domination (cf. Section 4.2.2.5) a une performance similaire aux états adaptés lorsque le type dominant profilé est identique au type de la distribution monomorphique (*branche type dominant*, cf. Figure 4.14). Lorsqu'ils sont différents, la performance plonge (-60%), la branche exécutée étant la branche de dispatch. Dans le cas mesuré, le dispatch est virtuel (l'état polymorphique-domination est généré pour le bytecode `invokevirtual`). On constate que les performances sont sensiblement inférieures à l'état dispatch-virtuel. Cet écart provient du test d'égalité entre le type du receveur et le type dominant (cf. Figure 4.10). De manière similaire, dans le cas d'un dispatch d'interface, on s'attend à ce que les performances soient légèrement inférieures à l'état dispatch-d'interface.

Les autres états non-adaptés (dispatch-virtuel et dispatch-d'interface-0[13]) souffrent d'un écart de performance supérieur à 50% par rapport aux états adaptés.
L'état polymorphique-CI (cf. Section 4.2.2.6) a une performance environ 15% inférieure aux états adaptés. Cet état est particulier dans le sens où il ne s'agit pas d'un état adapté mais d'un état stable uniquement pour les distributions monomorphiques et destiné à améliorer la performance de leurs exécution.
Il permet de gérer les cas où la méthode encapsulant le bytecode profilé (`invokevirtual` ou `invokeinterface`) est exécutée pour différents sites d'appels. Le profiling peut alors déterminer que la distribution globale considérant tous les sites d'appel est polymorphique. Cependant, le bytecode profilé peut être inliné dans un contexte pour lequel la distribution est monomorphique. On parle alors de pollution du profiling pour désigner les interférences d'informations issues de différents sites d'appels et entrainant une perte d'informations au niveau d'un site d'appel.
Dans le cas d'un code séquentiel possédant un unique site d'appel pour la méthode encapsulant le bytecode d'appel, la stabilité de l'état polymorphique-CI est exceptionnelle car elle suppose un changement de régime au niveau de la distribution, passant de polymorphique à monomorphique après compilation par C2.

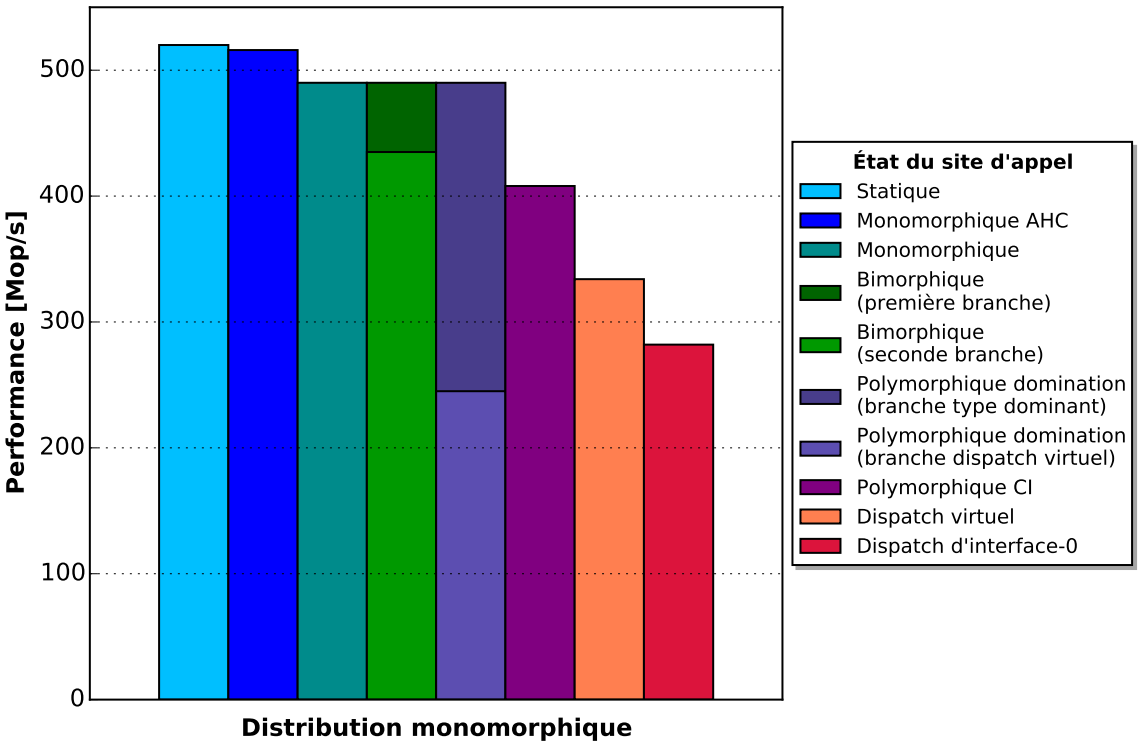


FIG. 4.14: Performance des états stables d'une distribution monomorphique. Les états monomorphique-AHC, monomorphique sont les états adaptés. L'état *statique* est utilisé comme référence

[13]Le suffixe "-<i>" indique que l'index de l'interface dans la *itable* est i (cf. Section 4.2.2.6).

État du site d'appel	**Écart de performance (en %) par rapport à l'état *Statique***
Statique	0%
Monomorphique-AHC	-1%
Monomorphique	-6%
Bimorphique (première branche)	-6%
Polymorphique-domination (branche type dominant)	-6%
Bimorphique (seconde branche)	-16%
Polymorphique-CI	-21%
Dispatch-virtuel	-35%
Dispatch-d'interface-0	-45%
Polymorphique-domination (branche dispatch-virtuel)	-52%

TAB. 4.4: Écart de performance des états stables d'une distribution monomorphique par rapport à l'état *Statique*. En *vert* figurent les états adaptés, en rouge les états finaux et en *bleu* les autres états stables

4.3.3 Résultats d'exécution d'une distribution bimorphique

La Figure 4.15 montre les résultats de performance des états stables pour une distribution bimorphique. Tous les états sont stables pour les distributions bimorphiques excepté les états adaptés aux distributions monomorphiques (monomorphique-AHC et monomorphique) et l'état polymorphique-CI (cf. Figure 4.4).
L'exécution des distributions bimorphiques est optimisée par le JIT grâce à l'état adapté bimorphique (cf. Section 4.2.2.4). Ce dernier donne de meilleures performances pour les distributions bimorphiques considérées.
Pour tout les états, la performance dépend du type de distribution qui impact la qualité de la prédiction de branchement. Celle-ci peut être observée en comparant les performance de l'état bimorphique pour les distributions *Al. 50-50* (aléatoire avec une répartition équilibrée) et *Per. 1* (distribution alternée entre 2 types). Dans les deux cas la proportion des types est identique, ce qui implique que la quantité d'instructions exécutées dans les deux cas est la même. On observe cependant un écart de performance de 30% en faveur du cas périodique qui correspond à un pattern prédictible pour la prédiction de branchement.
La légère différence de performance (4%) entre de la distribution *Al. 10-90* par rapport à la distribution *Al. 90-10* pour la distribution bimorphique vient du fait que pour la première itération la première branche du dispatch est prise 10% du temps dans le premier cas contre 90% du temps dans le second (cf. Figure 4.8).
Pour les états non-adaptés, les écarts de performance en moyenne sur les distributions considérées par rapport à l'état bimorphique sont résumés Table 4.5.

L'état polymorphique-domination a une performance similaire à l'état bimorphique pour la distribution *Al. 90-10* car le type dominant correspond alors au type présent à 90% dans la distribution, ainsi le premier branchement est pris 90% du temps. À l'inverse, pour la distribution *Al. 10-90*, la branche n'est empruntée que 10% du temps, le reste exécutant un dispatch virtuel, on observe alors une diminution de la performance de 27% par rapport à la distribution *Al. 90-10*.

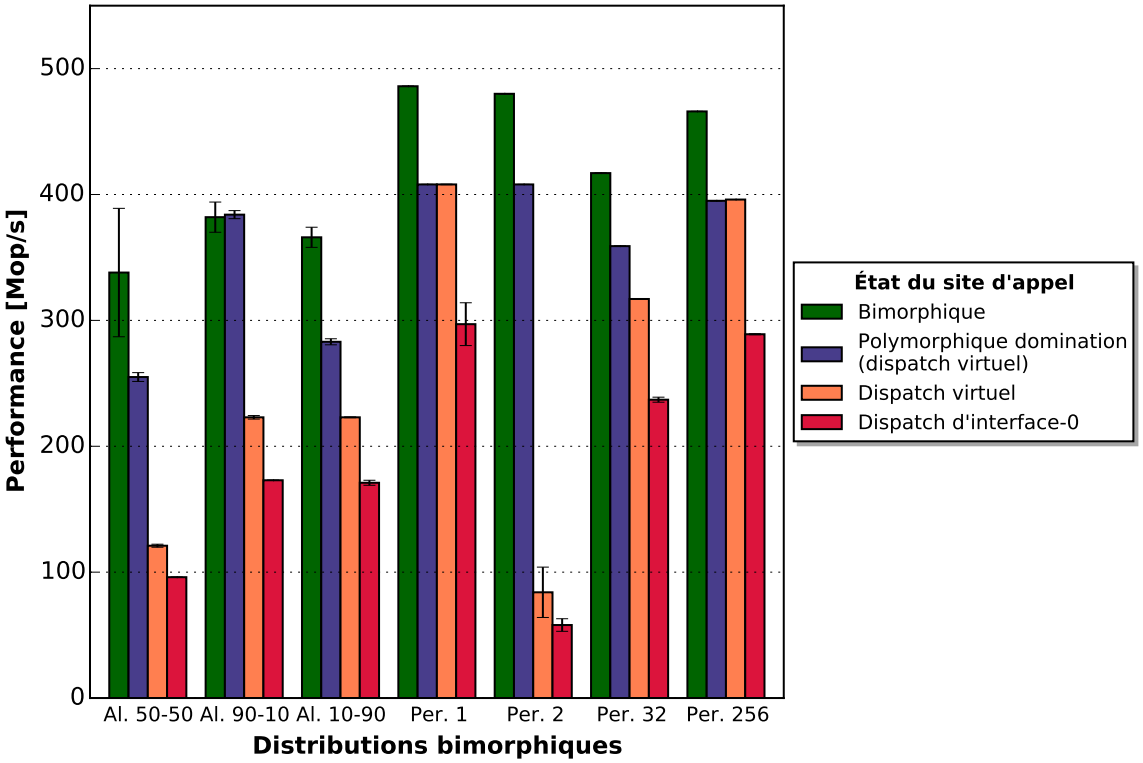


FIG. 4.15: Performances des états stables d'une distribution bimorphique pour différentes distributions bimorphiques. L'état bimorphique est l'état adapté, les autres états étant des états finaux

États non-adaptés	**Écart moyen de performance (en %) par rapport à l'état bimorphique**
Bimorphique	0%
Polymorphique-domination	-15%
Dispatch-virtuel	-40%
Dispatch-d'interface-0	-55%

TAB. 4.5: Écart de performance moyen des états non-adaptés par rapport à l'état Bimorphique pour les distributions bimorphiques considérées (cf. Figure 4.15).

4.3.4 Résultat d'exécution d'une distribution polymorphique

Les états stables pour les distributions dont le nombre de types impliqués est au moins de 3 sont les états polymorphiques à savoir : l'état polymorphique-domination, qui est un état optimisé ; les états avec dispatch à savoir soit dispatch-virtuel, soit le dispatch-d'interface. Ces états sont tous finaux (cf. Figure 4.4).

4.3.4.1 Dispatch virtuel VS. dispatch d'interface

Quelque soit les distributions exécutées, l'état dispatch-d'interface souffre d'un écart de performance quasi-constant par rapport à l'état dispatch-virtuel. Cet écart est en moyenne de -20% sur les distributions considérées et provient de la différence de nature entre `invokevirtual` et `invokeinterface` (cf. Section 4.2.2.6). La mesure considère le cas ou l'index de l'interface dans la *itable* est 0 (dispatch-d'interface-0).
En effet, la performance de l'état dispatch-d'interface dépend des indexes de l'interface dans les *itable* des types composant la distribution. La Figure 4.16 montre la performance en fonction de cet index dans le cas d'une distribution monomorphique. La performance est maximale pour l'index 0, car plus l'index est grand, plus la durée d'exécution est longue du fait de la recherche itérative de l'interface dans la *itable*. Les écarts de performance par rapport à l'index 0 sont résumés Table 4.6. Comme la recherche de l'interface correspond à l'exécution d'une boucle dont le nombre d'itérations est égal à l'index, la performance quand l'index croit tend théoriquement vers 0 avec une pente convexe (la durée d'exécution tend linéairement vers $+\infty$). En pratique il est très rare qu'une classe implémente plus de 5 interfaces, même si les spécifications limitent le nombre à 65535 [60]. L'index dans la *itable* de la classe implémentant l'interface correspond à la position de l'interface qui suit le mot clef `implements` dans la définition de cette classe.

Index dans la *itable*	**Écart de performance (en %) par rapport à l'index 0**
0	0%
1	-9%
2	-21%
3	-31%
4	-37%

TAB. 4.6: Impact de l'index de l'interface dans la *itable* sur la performance : écart par rapport à l'index 0

4.3.4.2 Impact de la grandeur du polymorphisme

La Figure 4.17 montre les performances des états polymorphiques en fonction de la grandeur du polymorphisme. Les distributions considérées sont les distributions aléatoires équirépartie. Par exemple pour une grandeur du polymorphisme de 4 la distribution consi-

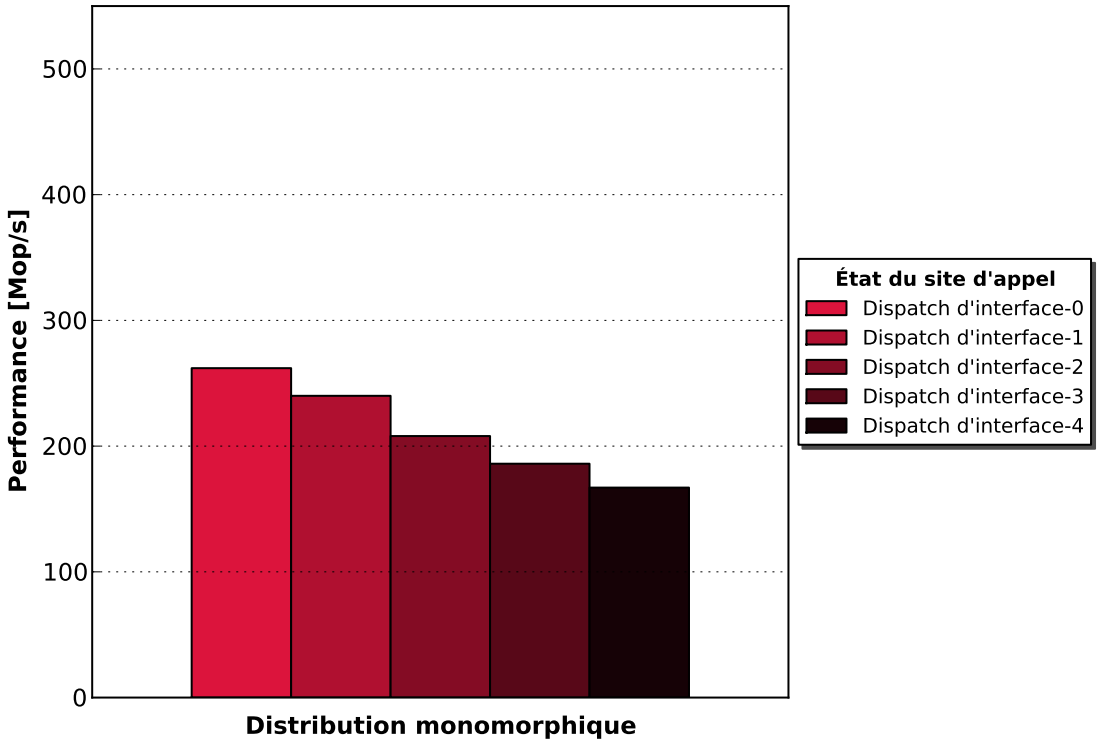


FIG. 4.16: Performance du dispatch d'interface en fonction de l'index de l'interface dans la itable. L'état dispatch-d'interface-X indique que l'index est X. L'index correspond à la position de l'interface derrière le mot clef `implements` dans la définition de la classe (la numérotation commençant à 0)

dérée est *Al. 25-25-25-25* (cf. Section 4.3.1).

Pour les états de dispatch (dispatch-virtuel et dispatch-d'interface-0), comme le code généré ne contient pas de branchements conditionnels, la complexité du code exécuté est la même quelque soit la distribution. Les écarts de performance observés sont dues à la prédiction de branchement. La diminution des performances pour les états polymorphiques lorsque la grandeur du polymorphisme augmente est résumée Table 4.7. Ces écarts sont mesurés par rapport à la performance pour une distribution monomorphique (grandeur du polymorphisme égale à 1). Sur l'architecture considérée, la diminution est significative à partir de 2 (supérieur à 50%) et se stabilise au delà autour de 75% pour des distributions aléatoires équiréparties.

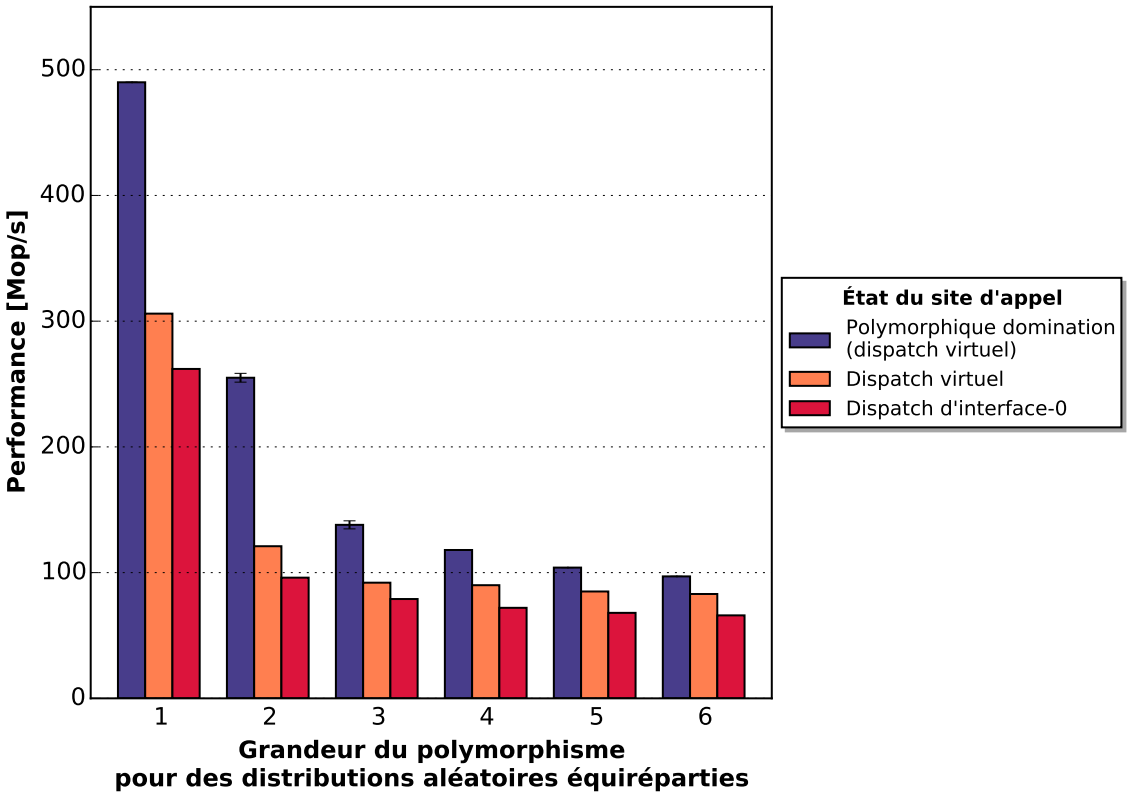


FIG. 4.17: Impact de la grandeur du polymorphisme sur les performances des états stables pour des distributions aléatoires équiréparties

Grandeur du polymorphisme	Polymorphique-domination	Dispatch-virtuel	Dispatch-d'interface
1 (monomorphique)	0%	0%	0%
2 (bimorphique)	-48%	-60%	-63%
3	-72%	-70%	-70%
4	-76%	-71%	-73%
5	-79%	-72%	-74%
6	-80%	-73%	-75%

TAB. 4.7: Impact de la grandeur du polymorphisme pour des distributions aléatoires équiréparties : écart de performance par rapport à une distribution monomorphique

4.3.4.3 Tri d'une distribution

La Figure 4.18 montre les performances des distributions 4-Polymorphiques périodiques dont la taille des paquets varie. Pour chaque état la complexité du code exécuté est la même quelque soit la distribution. Les variations de performance observées proviennent donc de la prédiction de branchement. Pour une taille de paquet de 1 la prédiction de branchement permet de maintenir un bon niveau de performance. Les performances plongent néanmoins pour les tailles de paquet supérieur puis remontent progressivement quand la taille des paquets augmente. La taille des distributions considérée étant de 2048, une taille de paquet de 512 correspond à un tableau ordonné par type. Cette distribution garantie une bonne prédiction de branchement et permet d'améliorer les performances par rapport à une distribution aléatoires. La Table 4.8 montre le facteur d'accélération obtenu pour des distributions équiréparties ordonnées par rapport à des distributions équiréparties aléatoires. Cette observation montre qu'un tri par type d'une distribution polymorphique avant l'exécution intensive par un site d'appel est une piste d'optimisation possible.

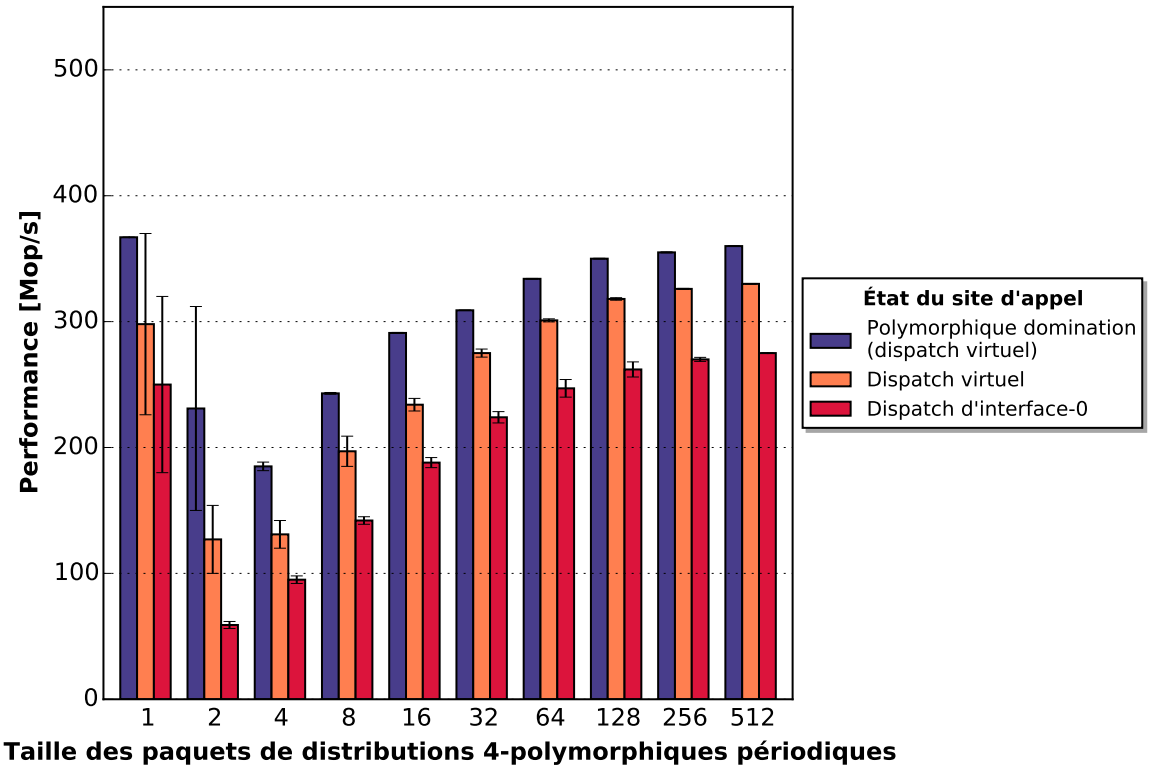


FIG. 4.18: Impact de la taille des paquets (cf. Section 4.3.1.1) sur les performances des états polymorphiques pour des distributions 4-polymorphiques périodiques

Grandeur du polymorphisme	**Polymorphique-domination**	***Dispatch virtuel***	***Dispatch d'interface***
2	$\times 2,80$	$\times 2,80$	$\times 2,80$
4	$\times 1,05$	$\times 2,70$	$\times 3,80$
8	$\times 3,80$	$\times 4,90$	$\times 4,60$

TAB. 4.8: Facteurs d'accélération des états polymorphiques observés après le tri de distributions aléatoires équiréparties

4.3.4.4 État polymorphique-domination

L'état polymorphique-domination (cf. Section 4.2.2.5) est un état polymorphique, optimisé pour améliorer la performance de l'exécution de distributions possédant un type dominant. Pour cela, le type du receveur est comparé au type profilé dominant. Si le test est vrai le code exécute la méthode correspondant au type dominant, si il est faux, il exécute la branche de dispatch qui correspond à un état polymorphique avec dispatch (état dispatch-virtuel pour `invokevirtuel` et dispatch-d'interface pour `invokeinterface`). Il est généré lorsque la distribution profilée est polymorphique avec un type présent au delà du seuil fixé par le paramètre `-XX:TypeProfileMajorReceiverPercent` de HotSpot qui vaut par défaut 90%. Néanmoins, cet état permet d'obtenir de meilleures performances que les états de dispatch également pour des proportions inférieures. Par ailleurs, l'état étant final, il peut également être moins performant si la proportion lors de l'exécution du type profilé dominant est trop faible. La Table 4.9 montre le facteur d'accélération de l'état polymorphique-domination (avec dispatch virtuel) par rapport à l'état dispatch-virtuel pour différentes proportions du type profilé dominant et différentes grandeurs du polymorphisme. Les résultats montrent que le seuil de 90% peut être abaissé sans risque comme même une proportion de 10% entraîne un gain par rapport à l'état dispatch-virtuel.

Grandeur du polymorphisme	100%	90%	50%	10%	0%
1	$\times 1,45$	N/A	N/A	N/A	$\times 0,74$
2	N/A	$\times 1,72$	$\times 1,52$	$\times 1,27$	$\times 0,80$
3	N/A	$\times 1,62$	$\times 1,62$	$\times 1,12$	$\times 0,95$
4	N/A	$\times 1,60$	$\times 1,50$	$\times 1,11$	$\times 0,97$

TAB. 4.9: Facteur d'accélération de l'état polymorphique-domination par rapport à l'état dispatch-virtuel pour différentes grandeurs du polymorphisme en fonction de la proportion du type profilé dominant. Les distributions considérées sont aléatoirement réparties, les autres types composant la distribution sont présents en même proportion. Par défaut, l'état polymorphique-domination est émit seulement si la proportion du type dominant est au moins de 90%

4.3.5 Extrapolation des résultats

Comme montré Section 4.1, les résultats de performance quantifient le coût d'un site d'appel en rendant négligeables les opérations adjacentes. Cette section donne différents éléments à considérer pour l'extrapolation des résultats à des cas d'exécution applicatifs. Soit t_{cs} la durée d'exécution moyenne d'un site d'appel cs, cette dernière peut s'écrire :

$$t_{cs} = PB + disp + \sum_{i=0}^{n-1}(f_i \times In_i \times T_i) \tag{4.1}$$

où n est la grandeur du polymorphisme de la distribution ; f_i la fréquence d'appel de la cible i ($\sum f_i = 1$), les n méthodes cibles étant numérotées de 0 à $n-1$; T_i le coût de la méthode i ; In_i le facteur d'accélération lorsque la méthode i est inlinée (vaut 1 si la méthode n'est pas inlinée) ; PB le surcoût occasionné par les défauts de prédiction de branchement au niveau du dispatch (PB vaut 0 lorsque la prédiction est parfaite) ; $disp$ le coût du site d'appel allant de l'entrée dans le dispatch à l'appel de la cible (ce dernier dépend également des fréquences f_i). Les paramètres PB et $disp$ dépendent de la distribution comme montré Section 4.1.2.

Coût des méthodes ciblées et inlining Dans le micro-benchmark considéré les principales variables quantifiées sont PB et $disp$. En effet, les T_i sont égaux et minimaux ($T_i = T_{min}$) et l'inlining est désactivé ($In_i = 1$), on a ainsi $t_{cs} = PB + disp + T_{min}$.
Dans les cas d'exécution réels, contrairement aux hypothèses faites, le coût moyen d'exécution des cibles ($\sum(f_i \times In_i \times T_i)$) peu rendre négligeable les coût de prédiction de branchement (PB) et/ou de dispatch ($disp$). Les résultats sont donc à relativiser à ce coût moyen. Par ailleurs le potentiel d'optimisation à l'inlining (In_i) doit être considéré pour l'optimisation du site d'appel. Ce dernier est possible uniquement en cas d'appel direct vers la cible. Dans le cas d'une cible peu fréquemment appelée mais avec un potentiel d'optimisation important il peut être plus bénéfique de générer un site d'appel permettant son inlining (ce qui peut nécessiter du dispatch explicite). L'effet de l'inlining pour les différents états générés par C2 est exposé Section 4.2.1.2. En supposant que PB est constant pour une distribution donnée et que $disp$ est négligeable, l'équation à minimiser est $t_{cs} = \sum_{i=0}^{n-1}(f_i \times In_i \times T_i)$ en jouant sur la nature du site d'appel pour permettre ou non l'inlining.

Le micro-benchmark est par ailleurs exécuté en isolation. Les conséquences sont que le site d'appel a l'exclusivité à la fois sur la prédiction de branchement et sur les données de profiling, ce qui s'avère être une hypothèse optimiste.

Prédiction de branchement Le caractère global de la prédiction de branchement (cf. Section 2.1.1.1 Chap. 2) rend la prédiction au niveau d'un site d'appel (i.e. le paramètre

PB) dépendante des branchements qui lui précèdent. Par exemple, dans le cas d'appels polymorphiques imbriqués, la prédiction de branchement peut être altérée au niveau d'un site d'appel, même si le pattern de branchement pris isolément au niveau du site d'appel est prédictible. Le micro-benchmark se place dans le cas où le site d'appel est isolé.

Pollution du profiling Dans des cas d'exécution réels une méthode contenant un site d'appel polymorphique peut être appelée dans différents contextes et/ou différents threads. Dans le cas du micro-benchmark seul un contexte est considéré et donc la distribution profilée correspond à la distribution issue de l'exécution de ce contexte. Dans le cas de plusieurs contextes, le profiling provenant d'un contexte peut perturber celui venant d'un autre contexte on parle alors de pollution du profiling [132]. Par exemple si la distribution est monomorphique dans un contexte et polymorphique dans l'autre, la distribution globale est polymorphique et le contexte monomorphique ne peut être optimisé (plus précisément lorsque la méthode contenant le site d'appel est inlinée). Le micro-benchmark se place dans un cas idéal où il n'y a pas de pollution du profiling.

4.4 Alternatives pour l'optimisation des performances

Comme vu Section 4.1.2 et observé dans les tests de performance, deux paramètres distincts affectent les performances d'un site d'appel `invokevirtual` ou `invokeinterface`, pour chacun d'eux l'impact dépendant de la distribution exécutée comme montré Figure 4.3 :

1. L'état du code généré par le JIT ;
2. La prédiction de branchement.

Les alternatives concernant l'état du code sont traités Section 4.4.1 et celles concernant la prédiction de branchement Section 4.4.2.

4.4.1 Concernant l'état du code généré

Défaut de profiling Concernant l'état du code généré par le JIT, les analyses précédentes permettent de distinguer deux cas de figure : avec ou sans *défaut de profiling*. Il y a *défaut de profiling* lorsque le code est spécialisé pour une distribution qui n'est pas représentative de la distribution globale considérée sur toute la durée d'exécution l'application. Elle survient lorsqu'il y a un changement de nature de distribution entre l'exécution pré-compilation effectuant le profiling et l'exécution post-compilation qui constitue la majeure partie du temps d'exécution. Par exemple, dans un cas extrême, la distribution lors du profiling peut être polymorphique et passer dans un régime monomorphique post-compilation jusqu'à la fin de l'exécution. Le potentiel d'amélioration lorsqu'il y a défaut de profiling peut être estimé en comparant les écarts de performances entre états adaptés et autres états stables (cf. Tables 4.4 et 4.5). L'alternative pour contourner le défaut de profiling est

d'utiliser un dispatch explicite mais suppose une connaissance au niveau applicatif de la distribution impliquée.

Limitations de profiling En dehors des défauts de profiling, l'autre aspect à considérer est la limitation du JIT concernant les distributions bénéficiant d'un état adapté. Les 3 distributions bénéficiant d'un état adapté sont les distributions monomorphiques, bimorphiques et polymorphiques avec un type dominant (cf. Figure 4.4). La majorité des distributions polymorphiques n'est donc pas optimisée par le JIT. Pour ces distributions, là encore, un dispatch explicite peut permettre d'améliorer les performances ce qui suppose une connaissance de la distribution au niveau applicatif. Par ailleurs, concernant les distributions polymorphiques, les appels virtuels `invokevirtual` sont à privilégier par rapport aux appels d'interface `invokeinterface` dont les performances dépendent de surcroit des indexes de l'interface dans les *itables* des types constituant la distribution (cf. Table 4.6). Enfin, un autre facteur pouvant améliorer les performances est d'étendre la génération de l'état polymorphique-domination à un nombre plus grand de distributions. Par défaut cet état correspond à des distributions polymorphiques dont un type est présent à plus de 90%, or cet état obtient de meilleurs performances que les états de dispatch également pour des seuils inférieurs. En baissant le paramètre `-XX:TypeProfileMajorReceiverPercent` de la JVM qui fixe ce seuil, on peut ainsi améliorer les performances pour des distributions polymorphiques avec un type présent en proportion importante.

4.4.2 Concernant la prédiction de branchement

L'autre dimension concerne la prédiction de branchement. Celle-ci affecte toutes les distributions non-monomorphiques et constitue un goulot d'étranglement majeur. Les patterns reconnues par l'unité de prédiction de branchement sont machine-dépendant et son impact précis relativement à une distribution requiert une analyse plus poussée avec des compteurs CPU. Concernant le pattern étudié, le tri de distribution polymorphique par type, constitue un facteur d'optimisation des performances significatif, en améliorant la prédiction de branchement (cf. Table 4.8). De manière générale ce type de solution n'est envisageable que lorsque le site d'appel exécute une distribution depuis une structure de donnée pouvant être triée, ce qui ne couvre pas tout les patterns d'utilisation intensive du polymorphisme. Le coût du tri de la distribution doit également est pris en compte et doit être couvert par l'exécution du site d'appel.

4.4.3 Dispatch explicite

Un dispatch explicite peut constituer une alternative ou un complément à l'utilisation du polymorphisme afin d'améliorer les performances pour certaines distributions polymorphiques. Il peut notamment permettre de combiner les différents états générés par le JIT afin de bénéficier de l'inlining.

La stratégie utilisée par le JIT est de placer les branchements conditionnels directs par ordre de probabilité d'exécution, ces derniers pouvant être inlinés, et de placer en dernier le dispatch via la table des méthodes (utilisant des branchements indirectes), ce qui permet également de réduire le coût du dispatch (cf. Eq. 4.1 Section 4.3.5) .
Ces optimisations peuvent être appliquées à des distributions polymorphiques hiérarchisées, non considérées par le JIT, en utilisant un dispatch explicite.

Application du dispatch explicite pour une distribution La Figure 4.4.3 montre une implémentation de dispatch explicite optimisée pour une distribution polymorphique avec le type `Type0` dominant suivi du type `Type1`. La Figure 4.20 montre les performances obtenues pour différentes distributions respectant cette hiérarchie. En moyenne pour les distributions considérées, elle permet d'obtenir un facteur d'accélération de 270% par rapport à l'état dispatch-virtuel, qui est celui généré par défaut, et de 156% par rapport à l'état polymorphique-domination. Pour des distributions 3-polymorphiques et 4-polymorphiques, cette implémentation permet de contourner l'utilisation de la *vtable* et permet l'inlining de toutes les méthodes. En effet le troisième branchement générera l'état monomorphique pour une distribution 3-polymorphique, et bimorphique pour une distribution 4-polymorphique. L'implémentation utilise l'opérateur `instanceof` pour tester le type du receveur. La redondance fonctionnelle des opérations `checkcast` (générée par la conversion de type du receveur) et de `instanceof` est optimisée sans problème par C2.

```
public void dispatch_method(AbstractType rcv) {
    if (rcv instanceof Type0) {
        Type0 rcv0 = (Type0) rcv;
        rcv0.method();
    } else {
        if (rcv instanceof Type1) {
            Type1 rcv1 = (Type1) rcv;
            rcv1.method();
        } else
            rcv.method();
    }
}
```

FIG. 4.19: Méthode `dispatch_method` effectuant un dispatch explicite vers la méthode `method` (de signature `void(void)`), optimisée pour une distribution polymorphique hiérarchique avec `Type0` dominant suivie de `Type1`, le reste de la distribution étant indéterminé

Paramétrisation de la distribution La distribution peut être connue mais variable par instance d'application. Dans ce cas, la méthode effectuant le dispatch peut être une méthode virtuelle pour permettre la paramétrisation de la distribution. L'implémentation dépend alors du type de distribution qui implémente sa version la méthode effectuant le dispatch. La Figure 4.4.3 reprend le pattern étudié (cf. Figure 4.1) en intégrant le dispatch explicite qui dépend du type de distribution `distType`. En considèrant que la distribution

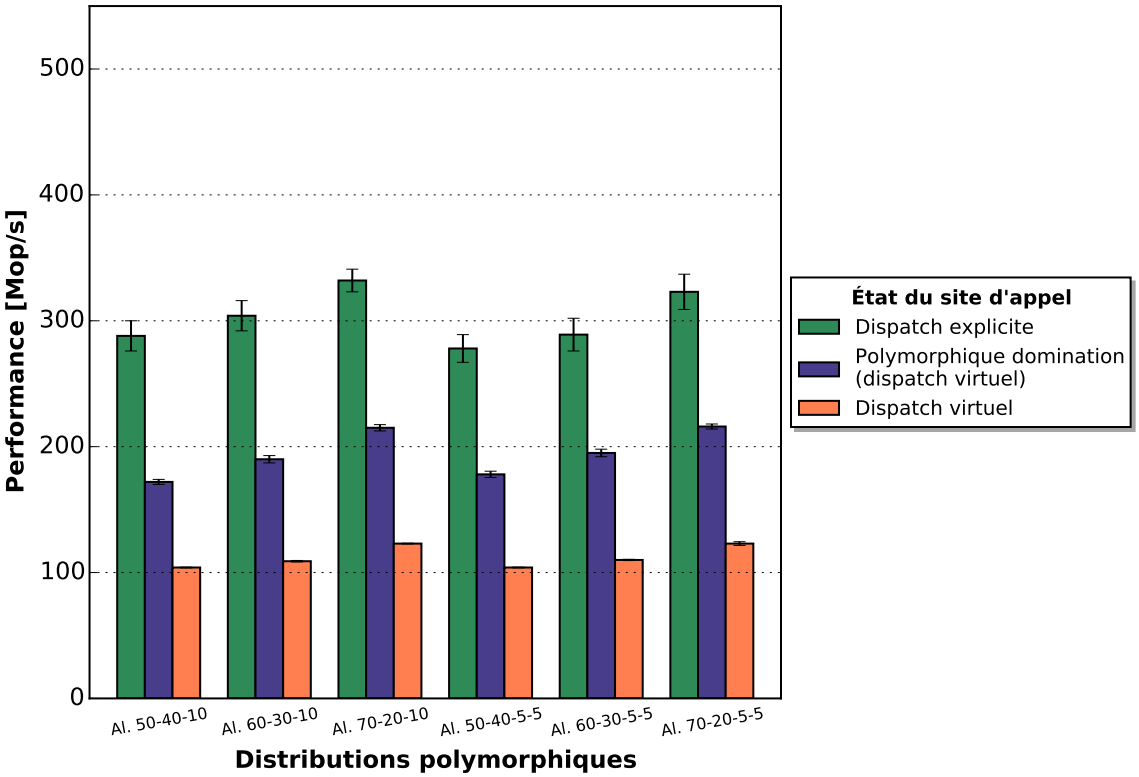


FIG. 4.20: Performances pour différentes distributions polymorphiques hiérarchiques. L'état dispatch-virtuel est l'état adapté généré par défaut.

est unique par instance d'exécution, la méthode de dispatch sera inlinée par le JIT.

```
static void compute(AbstractType[] data, distType dist){
    int n = data.length;
    for (int i = 0; i < n; ++i)
        dist.dispatch_method(data[i);
}
```

FIG. 4.21: Méthode `compute` implémentant le pattern *appel polymorphique dans une boucle* avec dispatch explicite dépendant du type de distribution `distType` qui définit la méthode `dispatch_method`. Une version de cette méthode pour une distribution polymorphique hiérarchique est donnée Figure 4.4.3

4.5 Conclusion

Cette étude propose une analyse des performances du polymorphisme. Dans un premier temps, une métrique de performance du polymorphisme est définie. Cette dernière repose sur la conception d'un micro-benchmark générique i.e. indépendant de tout traitement applicatif spécifique. Cette généricité a pour objectif de favoriser l'extrapolation des

résultats à des patterns similaires au sein de l'application et d'obtenir une quantification des performances.
Dans un second temps, l'étude consiste en l'analyse des différents états pouvant être générés par le JIT relativement à la distribution des types au niveau du site d'appel. Cette étape à pour objectifs, d'une part, de permettre aux développeurs de connaître les optimisations effectuées par le JIT et, d'autre part, de savoir si ces dernières son appliquées aux cas d'utilisation applicatifs.
Les concepts d'états *adaptés* et d'états *stables* ont été définis afin de couvrir tous les cas possibles pouvant se présenter lors de l'exécution de l'application. Cette étude a permis de quantifier les baisses potentielles de performance et de mettre en avant des opportunités d'optimisation en considérant les deux paramètres fondamentaux que sont l'état du site d'appel et la prédiction de branchement.
L'état du site d'appel peut être amélioré dans les cas où il y a un *défaut de profiling* ou dans les cas non pris en charge par le JIT (cf. Section 4.4.1). Dans ces cas, une connaissance applicative de la distribution mise en jeu, peut, avec du dispatch explicite emmener à améliorer les performances. Pour certaine distribution, le réglage des paramètres du JIT peut également permettre d'améliorer les performances.
La prédiction de branchement repose principalement sur les capacités matérielles de l'unité de prédiction de branchement, néanmoins, au niveau code source Java, le tri de la distribution par type permet d'améliorer substantiellement les performances en réduisant la variance sur les branches d'exécution. Un potentiel d'optimisation existe également en prenant en compte les informations sur l'unité de prédiction de branchements. Ces dernières doivent néanmoins être effectuées côté runtime considérant la forte dépendance à l'architecture et la forte compétence métier qu'elles requièrent.
L'approche utilisée dans cette étude, qui vise à quantifier les performances d'un bytecode donné (en l'occurrence `invokevirtual` et `invokeinterface`) relativement aux paramètres impactant l'état du code, peut être généralisée à d'autres bytecodes. En particulier ceux bénéficiant d'information dynamique pour lesquels l'état du code généré possède une grande variabilité. Cette connaissance globale améliorait nettement l'analyse des performances du code Java et favoriserait également l'identification du code critique.
Notons pour finir que les optimisations identifiées sont relatives à HotSpot. Cette démarche s'inscrit donc dans une approche par programmation JIT-Friendly pour l'amélioration des performances (cf. Section 7.1.1 Chap. 7).

Chapitre 5

Compromis qualité du code natif/coût d'intégration via JNI

Ce chapitre considère l'utilisation de JNI (Java Native Interface) afin d'améliorer les performances d'application Java faisant intensivement appel à des routines calculatoires. JNI désigne à la fois une spécification [101] et une API C du JDK permettant à du code Java d'appeler des fonctions natives depuis des librairies dynamiques et de manipuler des objets Java depuis ce code natif.
Son efficacité pour l'amélioration des performances repose sur le compromis coût d'intégration/qualité du code natif. Ce compromis signifie que l'on accepte, en utilisant JNI, de payer un coût constant élevé à chaque appel de méthode native (par rapport à celui de méthodes Java non natives) si le code natif appelé est plus performant que celui généré par le JIT et permet d'obtenir une accélération.
Cette condition dépend de la quantité de calcul exécutée à chaque appel qui doit être suffisante pour couvrir le coût d'intégration via JNI.
On se propose dans ce chapitre de comparer les performances d'implémentations Java et JNI de différentes routines élémentaires en faisant varier la quantité de calcul afin de déterminer quelle est la meilleure implémentation sur un intervalle donné.

La Section 5.1 souligne les motivations et le problème traité. La Section 5.2 propose un background sur JNI en décrivant l'origine de son coût d'intégration élevé. Dans la Section 5.3 on définit le terme de profil de performance ainsi que son usage. Dans la Section 5.4 on distinguera la nature des routines considérées pour l'analyse par profil de performance, ainsi que les optimisations permises par l'utilisation de code natif. Enfin la Section 5.5 est consacrée à la description des micro-routines traitées et à l'analyse des profils de performance.

5.1 Problématique et motivations

Lorsqu'il implémente une méthode, le développeur Java dispose de deux choix :

1. L'implémentation Java-pure (i.e. bytecode) constituant la quasi-totalité du code ;
2. L'usage de JNI qui reste marginal comme il peut impacter la portabilité de l'application et ajoute un effort de maintenance.

L'utilisation de JNI est habituellement orientée à des fins fonctionnelles (par exemple accéder à des fonctionnalités système ou relever des compteurs CPU [51, 72]), néanmoins l'idée d'utiliser JNI afin d'améliorer les performances est apparue dans les premières versions de Java [57].
JNI est ici évalué pour appeler des routines calculatoires de faible granularité, critiques en terme de performance. Ces routines représentant une portion de code négligeable dans l'application, l'usage de solutions non conventionnelles et plus fastidieuses à maintenir est donc envisageable. L'utilisation de JNI repose sur deux motivations :

1. Le manque d'expressivité et le haut niveau d'abstraction du langage Java ne permet pas de faire de l'optimisation bas niveau sur l'architecture considérée ;
2. Le niveau d'optimisation du JIT peut être moindre que celui d'un compilateur statique comme GCC ou que de l'optimisation manuelle bas-niveau.

Néanmoins, en supposant que le code natif soit de meilleure qualité, deux autres aspects mitigeant son efficacité doivent également être considérés :

1. La spécialisation du code Java permise par la compilation dynamique (profiling et inlining) offre un potentiel d'optimisation dont ne disposent pas les fonctions natives ;
2. Le coût d'intégration via JNI ralentit les performances par rapport aux méthodes Java qui, de surcroît, peuvent être inlinées.

Pour les routines calculatoires considérées, travaillant sur des structures de données en mémoire, le potentiel d'optimisation provenant de la spécialisation du code par propagation de constante est faible. Ainsi le second aspect, à savoir le coût d'intégration via JNI, est le facteur limitant à prendre en considération. Ce coût d'intégration se manifeste par un coût constant relativement élevé, payé à chaque appel de méthode native.

Considérant le coût d'intégration via JNI, on propose de déterminer à partir de quelle quantité de calcul effectuée par la routine l'utilisation de JNI est profitable. Pour cela on considère le *profil de performance* des différentes implémentations de la routine. Le profil de performance est la courbe représentant les performances en fonction de la quantité de calcul. L'objectif est ainsi de déterminer la meilleure implémentation sur un intervalle de quantité de calcul donné et, si il y en a, les points de croisement entre implémentations Java et implémentations JNI.

5.2 Coût d'intégration via JNI

La Section 5.2.1 propose un background sur JNI et les différentes composantes de son coût d'intégration. La Section 5.2.2 décrit les différents usages de JNI pour des méthodes natives manipulant des tableaux primitifs influant également sur les performances. Enfin la Section 5.2.3 propose de quantifier le coût d'intégration intrinsèque via JNI en considérant différentes signatures de méthodes vides.

5.2.1 Composantes du coût d'intégration

JNI désigne à la fois une spécification [101] et une API C du JDK permettant à du code Java d'appeler des fonctions natives depuis des librairies dynamiques et de manipuler des objets Java depuis ce code natif via des fonctions de rappel (fonctions de rappel JNI). La Figure 5.1 montre la chaîne d'appel lorsqu'une méthode Java-pure appelle une méthode Java native. Le contenu de la méthode native, qui joue le rôle d'interface entre le code Java

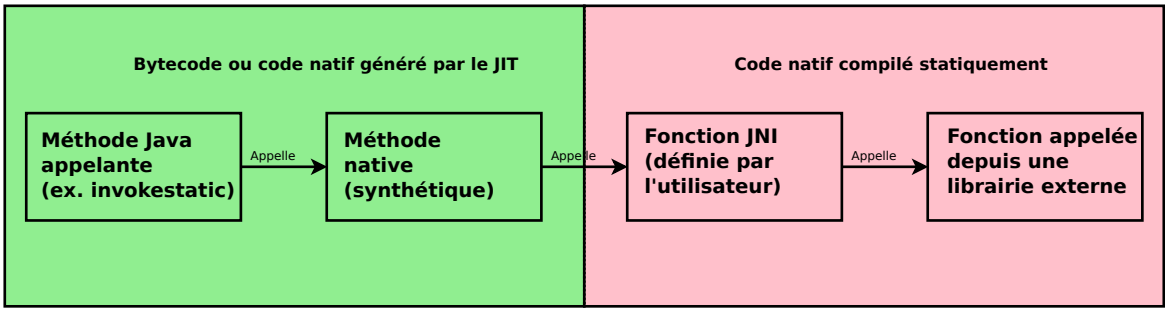


FIG. 5.1: Chaîne d'appel JNI depuis une méthode Java vers une fonction native. La méthode native est synthétique (i.e. son contenu est généré par le runtime), elle sert d'interface entre le monde Java et le monde natif. La fonction JNI utilise les fonctions de rappel JNI afin d'accéder aux données dans la heap puis appel la fonction native depuis une librairie dynamique. Notons que le code natif ciblé peut être défini directement dans la fonction JNI s'il n'est pas définit dans une fonction d'une librairie externe

et le code natif, est généré par le runtime. L'utilisateur définit uniquement sa signature relativement à la signature de la fonction native ciblée. La fonction JNI utilise les fonctions de rappel JNI afin que la fonction ciblée puisse manipuler les objets Java. Le coût d'intégration via JNI a ainsi deux composantes :

1. Le coût de la méthode native (cf. Section 5.2.1.1) ;
2. Le coût des fonctions de rappel JNI (contenu dans la fonction JNI) permettant au code natif de manipuler les structures de données Java (cf. Section 5.2.1.2).

5.2.1.1 Appel de méthode native

La première composante est le coût d'appel vers la fonction native JNI définie dans la librairie dynamique. Comme montré Figure 5.1, la méthode native Java sert d'interface entre la méthode appelante Java et la fonction native ciblée au sein d'une librairie dynamique. La méthode native peut être compilée ou interprétée comme toute autre méthode Java mais elle n'est pas inlinée. Elle effectue principalement les opérations suivantes :

- Les références passées en argument sont encapsulées dans des handles[1] ;
- Les premiers paramètres de la fonction JNI (i.e. `JNIEnv*` et `jclass` dans le cas de méthodes statiques) sont requêtés ;
- Les arguments de la fonction native sont copiés selon les conventions d'appel du code natif[2] ;
- Le moniteur du receveur est verrouillé dans le cas d'une méthode synchronisée ;
- Le thread courant bascule de l'état Java à l'état natif ;
- La fonction cible JNI est appelée ;
- L'exécution passe par un safepoint ;
- Le thread bascule de l'état natif à l'état Java ;
- Le moniteur du receveur est déverrouillé si la méthode est synchronisée ;
- Les références produites sont extraites des handles selon les conventions d'appel Java ;
- Les handles sont libérés.

Le code assembleur généré par le JIT pour ces différentes étapes peut être visualisé via l'option `-XX:+PrintNativeNMethods` lorsque la méthode native est compilée.
Dans [95], Sunderam et Kurzinyec ont étudié les performances de JNI pour différentes signatures d'appel et différentes implémentations de la JVM. Les résultats montrent des coûts d'invocation avec un facteur de pénalité allant jusqu'à 16 dans les pires cas. Les expérimentations de Murray et al. dans [112] exposent des résultats similaires.

5.2.1.2 Fonctions de rappel JNI

La seconde composante du coût d'intégration provient des fonctions de rappel JNI. Du fait du design machine-indépendant de JNI, les fonctions de rappel JNI sont accédées via un pointeur d'interface (`JNIEnv`), propre au thread courant, et passé systématiquement comme premier paramètre des fonctions natives. La structure `JNIEnv` contient un tableau de pointeurs, chacun pointant vers une fonction de rappel JNI. L'invocation d'une fonction de rappel procède ainsi à 2 niveaux d'indirection : le premier pour accéder au pointeur de fonction et le second pour appeler la fonction. Le coût dépend ensuite de la fonction de rappel utilisée.

[1] Les conventions JNI requièrent que les *oops* soient contenus dans des *handles*. Les handles sont des pointeurs mémoire connus du runtime contenant les *oops* utilisés dans JNI. Ils servent à repérer les racines du GC et à maintenir la validité des oops utilisés dans le code natif à travers le GC. La nécessité des handles provient du fait que, contrairement au code binaire généré par le JIT ou au bytecode, l'emplacement des oops dans le code natif ne peut ni être connu, ni être modifié.

[2] Sur architecture Linux x86-64 les conventions d'appels sont définies dans l'ABI System V [105]

5.2.2 Utilisation des tableaux primitifs

Les routines calculatoires considérées (cf. Section 5.5.1) sont implémentées comme des méthodes statiques manipulant des tableaux primitifs. JNI offre 3 manières de manipuler ces structures de données depuis du code natif :

1. Avec *transfert des données* ;
2. Avec *section critique* ;
3. Avec *mémoire native*.

5.2.2.1 Transfert des données

La première manière de manipuler des tableaux primitifs depuis du code natif est d'utiliser les fonctions de rappel JNI `<Get/Realease><`*`type`*`>ArrayContents` [101]. Ces fonctions déclenchent respectivement des copies des structures Java de la heap vers la mémoire native (ou mémoire hors heap) et inversement de la mémoire native vers la heap. Cette solution s'avère cependant bien trop coûteuse dans le cas de micro-routines pouvant avoir une fréquence d'appel élevée. L'usage de ces fonctions de rappel ajoute un coût asymptotique (et non un coût constant) à la routine comme sa valeur croît linéairement avec la taille des tableaux primitifs copiés.

5.2.2.2 Sections critiques

La seconde manière de manipuler des tableaux primitifs depuis du code natif est d'utiliser les fonctions `<Get/Release>ArrayCritical` [101]. Ces fonctions permettent de délimiter (d'entrer et de quitter) des *sections critiques*. Dans le contexte de JNI, Une section critique est une section de code natif dans laquelle les structures de données Java peuvent être manipulées sans copie préalable c.à.d directement depuis la heap. L'exécution d'une section critique est cependant restrictive. Les appels à JNI ou les appels systèmes pouvant bloquer le thread courant en attendant un autre thread Java ne peuvent pas être effectués. L'exécution d'une section critique est notamment bloquante pour le GC, ce qui peut entraîner des effets d'inter-blocage indésirables. Ces restrictions ne rentrent cependant pas dans le cadre de notre usage pour des micro-routines calculatoires.

5.2.2.3 Mémoire native

Enfin la troisième manière de manipuler des tableaux primitifs depuis du code natif est d'utiliser directement de la mémoire native. La classe `Unsafe` du JDK permet de manipuler la mémoire native depuis du code Java, il est ainsi possible d'utiliser du code JNI utilisant directement des structures en mémoires natives préalablement allouées depuis du code Java. L'intérêt de cette méthode est qu'elle s'affranchit du coût des fonctions de rappel JNI et est donc plus performante. Son utilisation au sein des applications s'avère néanmoins délicate comme elle requiert la manipulation de mémoire native depuis le code Java. Par ailleurs, et

en fonction des contraintes applicatives, l'utilisation de la mémoire native peut nécessiter des transferts de données Java vers la mémoire native ce qui peut impacter négativement les performances.

5.2.3 Mesure du coût d'intégration à vide

La Table 5.1 montre les performances comparées des appels de méthodes Java natives par rapport aux appels de méthodes Java-pure ainsi que le facteur de décélération observé. Les techniques d'appel JNI considérées sont avec *section critique* et avec *mémoire native* (cf. Section 5.2.2). L'étude ne considère pas les appels avec *copie*, trop coûteux par rapport aux deux autres approches.
Les valeurs mesurées représentent le débit d'appel en millier d'appels/seconde pour des méthodes à vide. Différentes signatures des paramètres d'entrée sont considérées. Le débit dépend du nombre de paramètres d'entrée pour toutes les implémentations. La variation est néanmoins moins sensible pour les implémentations *Java* et *JNI avec mémoire native*. On peut observer la différence de coût d'intégration entre *JNI avec section critique* et *JNI avec mémoire native*, d'environ 89%, provenant des fonctions de rappels pour entrer et sortir d'une section critique lorsque la routine manipule des tableaux primitifs. L'impact des fonctions de rappel augmente avec le nombre de tableaux primitifs passés en argument. Le facteur de décélération dû uniquement au coût d'appel à JNI sans les fonctions de rappel est observé en comparant *Java* et *JNI avec mémoire native* et est d'environ 75%. L'impact du coût d'intégration est, dans des cas réels, à relativiser par rapport au contenu de la méthode appelée. Néanmoins, lorsque la quantité de calcul devient nulle les performances se rapprochent théoriquement du coût d'intégration à vide.

	Java non inlinée	**JNI section critique**	**JNI mémoire native**
`()`	600	147 (-75%)	147 (-75%)
`(double[])`	595	25 (-96%)	147 (-76%)
`(double[], double[])`	580	14 (-97,6%)	143 (-76%)
`(double[], double[], double[])`	520	9 (-98,5%)	131 (-75%)

TAB. 5.1: Débit d'appel de méthodes statiques vides (en millier d'appels/seconde) pour différentes signatures des paramètres d'entrée. La valeur entre parenthèse correspond au facteur de décélération par rapport à Java

5.3 Profil de performance

Le profil de performance et son usage sont décrits Section 5.3.1. Un modèle théorique simple est ensuite proposé Section 5.3.2 afin d'observer l'impact du coût d'intégration via JNI sur les profils de performances.

5.3.1 Définition et usage

La quantité de calcul effectuée par une routine peut être variable ou constante par appel. Si elle est variable, alors elle dépend de paramètres entrants qualifiés de *facteurs d'échelle*. Les facteurs d'échelle interviennent également dans l'écriture de la complexité algorithmique. Plus précisément, l'évolution asymptotique de la quantité de calcul en fonction des facteurs d'échelle est donnée par la complexité algorithmique. Les routines considérées sont toutes de complexité linéaire.

On appel *profil de performance* d'une routine, la fonction représentant ses performances en fonction de la quantité de calcul effectuée par appel de cette routine. La variation de la quantité de calcul est obtenue en faisant varier la valeur des facteurs d'échelle qui se réduisent pour les routines considérées à la taille des données d'entrée et de sortie.

L'utilisation des profils de performance est proposée afin de déterminer, parmi différentes implémentations d'une routine, quelle est la meilleure sur l'intervalle des quantités de calcul considéré dans l'application.
Le profil de performance permet notamment d'identifier les points croisement entre implémentations JNI et implémentations Java pouvant provenir du coût d'intégration via JNI (cf. Section 5.3.2). Par ailleurs certaines optimisations comme la vectorisation sont également sensibles aux facteurs d'échelle ; leur augmentation provoquant des effets de cache et atténuant les performances. Dans ce cas, le profil de performance permet d'observer cette influence et ainsi de caractériser les routines.
Les profils de performance sont calculés pour une implémentation donnée en utilisant des micro-benchmarks selon la méthodologie décrite Chapitre 3.3.
Les profils de performance considérés sont unidimensionnels ce qui permet de simplifier leur mesure et leur représentation. En théorie, la dimension du profil de performance est égale au nombre de facteurs d'échelle. Au delà d'une dimension les points de mesure augmentent de manière exponentielle ce qui peut nécessiter l'utilisation de techniques d'interpolation plus poussées pour couvrir les performances sur tous les domaines d'intérêt.

5.3.2 Impact théorique du coût d'intégration via JNI

Les notions de coût constant, coût constant propre et coût asymptotique sont définis Section 5.3.2.1 puis, Section 5.3.2.2, un modèle descriptif simple basé sur une hypothèse de linéarité est proposé pour décrire l'impact du coût d'intégration via JNI.

5.3.2.1 Coût constant et coût asymptotique

Le coût d'intégration via JNI se manifeste par un coût constant élevé (cf. Section 5.2). Le coût constant désigne le temps d'exécution des opérations constantes, systématiquement exécutées à chaque appel de routine et dont l'occurrence ne dépend pas de la valeur

des facteurs d'échelle. Il s'oppose au coût asymptotique composé des opérations dont le nombre d'exécution par appel dépend des facteurs d'échelle. Par exemple les instructions contenues dans une boucle dont la taille dépend des paramètres d'entrée participent au coût asymptotique, alors que le coût d'initialisation de la boucle participent au coût constant. On distingue le coût constant propre à la routine qui ne considère pas le coût d'intégration via JNI et le coût constant qui considère également le coût d'intégration via JNI.

5.3.2.2 Modèle descriptif

Un modèle simple pour décrire l'impact du coût constant sur les performances est donné par la formule

$$P = P_{asympt} \times \frac{q}{C+q}$$

où P est la performance (en nombre d'opérations pas seconde), q est la quantité de calcul variable (en nombre d'opérations) et C est le nombre d'opérations constantes (de même dimension que q). L'équation est obtenue en faisant l'hypothèse que le temps d'exécution est une fonction linéaire du nombre d'opérations, la valeur en zéro étant le coût constant. Afin de visualiser l'impact de C sur les performances, on utilise la performance normalisée donnée par :

$$\frac{P_{asympt}}{P} = \frac{q}{C+q}$$

La performance normalisée fonction de la quantité de calcul q est représentée Figure 5.2 pour différentes valeurs de C. Comme on peut l'observer, plus le coût constant est élevé, plus l'impact sur les performances pour de faibles quantités de calcul est important.

Pour les implémentations Java, les routines calculatoires sont implémentées sous forme de méthodes statiques qui peuvent être inlinées. Le coût constant des implémentations Java étant bien inférieur aux implémentations JNI, pour des faibles quantités de calcul, les implémentations Java peuvent s'avérer plus performantes malgré des optimisations asymptotiques moins bonnes que les implémentations JNI. On peut alors observer un point de croisement comme montré Figure 5.3. L'implémentation la plus performante dépend donc de la quantité de calcul par appel. Un des objectifs proposé est ainsi de quantifier ce point de croisement pour des routines calculatoires applicatives. L'hypothèse de linéarité effectuée pour obtenir la formule reste une approximation afin d'observer de manière isoler l'impact du coût constant. En pratique les performances peuvent dépendre de la quantité de calcul de manière non linéaire. On observera notamment sur des routines avec une intensité calculatoire constante une baisse de performance lorsque la quantité de calcul augmente du fait des effets de cache.

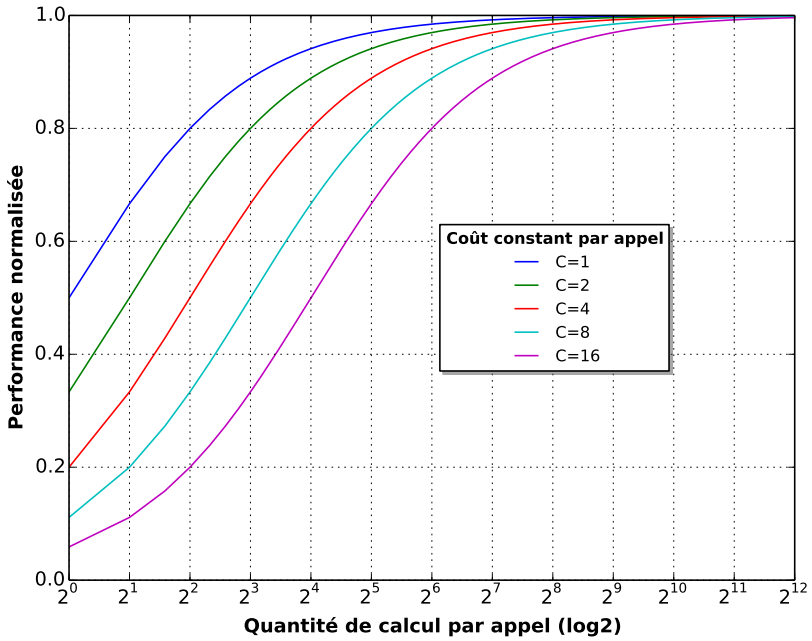


FIG. 5.2: Performance normalisée en fonction de la quantité de calcul. On observe l'impact sur les performances pour différentes valeurs du coût constant par appel C : plus il est élevé, plus les performances sont faibles lorsque les quantités de calcul par appel diminuent

5.4 Types de routines considérés et optimisations natives

Dans cette section, la distinction entre deux catégories de routines de calcul pouvant faire l'objet d'optimisations natives est faite :

1. Le premier type (cf. Section 5.4.1) contient les routines travaillant sur des structures de données complexes (par opposition aux tableaux de type primitif) et dont le coût d'appel est négligeable par rapport à son coût opérationnel, indépendamment de la valeur des facteurs d'échelle. Il s'agit généralement de routines de complexité non linéaire avec un coût constant propre important rendant le coût d'appel via JNI moins critique ;
2. Le second type (cf. Section 5.4.2) contient les micro-routines de complexité linéaire, manipulant des tableaux primitifs et pour lesquelles le coût d'appel peut avoir une importance lorsque la quantité de calcul est faible.

5.4.1 Routines utilisant des structures de données complexes

On entend par structures de données complexes, toute structure de données autre que les tableaux primitifs. La particularité des routines utilisant ces structures de données, est que leur intégration via JNI nécessite le transfert des données d'entrée depuis la heap vers la mémoire native et, si il y en a, le rapatriement des données de sortie depuis la mémoire native vers la heap. Cette copie est nécessaire car ces dernières ne peuvent pas être lues

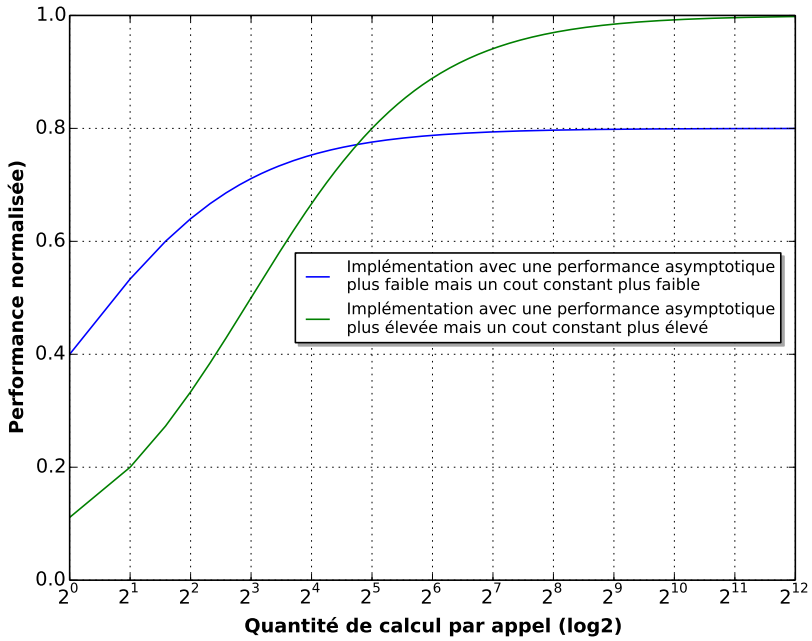


FIG. 5.3: Performance normalisée en fonction de la quantité de calcul pour deux implémentations différentes. On observe un point de croisement, la meilleur implémentation dépend de la quantité de calcul

ou écrites en-place (i.e. directement dans la heap) par du code natif. Par ailleurs, les définitions natives et Java de ces structures peuvent différer. Pour ces routines, l'utilisation directe de mémoire native est une piste permettant d'éviter les transferts de données. Cette solution bouleverse néanmoins le paradigme de programmation et peut nécessiter la modification de tout le flow de traitement. Elle s'avère donc coûteuse et risquée d'un point de vue développement.

Le coût du transfert des données s'ajoute au coût asymptotique et non au coût constant comme ce dernier dépend de la taille de la structure de données transférée qui est un facteur d'échelle. Par ailleurs, le coût constant n'est pas seulement composé du coût d'intégration via JNI mais également du coût constant propre à la routine, composé d'opérations algorithmiques, qui peut éventuellement masquer le coût d'intégration via JNI. Ainsi, pour ce type de routine, la principale contrainte à l'utilisation de JNI est le coût de transfert des données.

Le transfert des données peut se faire de deux manières. Depuis du code natif ou depuis du code Java. L'implémentation Java utilise la classe `Unsafe` pour écrire ou lire la mémoire native. Elle requiert la connaissance de l'agencement mémoire de la structure de donnée native. La seconde utilise les fonctions de rappel JNI depuis du code natif. L'implémentation Java s'avère plus performante ce qui s'explique par le coût élevé des fonctions de rappel JNI, en particulier les fonctions `GetObjectArrayElement` et `DeleteLocalRef` [101], permettant de créer et de libérer une référence locale vers un objet depuis un tableau, pour lire ou écrire des champs d'objet.

L'optimisation de telles routines applicatives peut se faire par portage du code Java en code C++ (cf. Section 5.4.1.1) en déléguant l'effort d'optimisation sur le compilateur natif, comme g++, et sur la définition des structures natives. Le gain de performance mesuré provient alors d'une meilleure optimisation statique du code par g++ par rapport à l'optimisation dynamique faite par C2, d'une meilleure expressivité du C++ par rapport au Java et/ou d'accès mémoire plus rapides dus à l'utilisation de structures native améliorant la localité des données [42]. En dehors du portage, JNI permet également l'appel à des librairies natives optimisées (cf. Section 5.4.1.2). Un des avantages majeurs du code natif est l'existence de librairies micro-optimisées pour une architecture donnée, telle que FFTW, à la différence des librairies Java dont les optimisations se limitent à la couche algorithmique.

5.4.1.1 Portage en `C++` d'un algorithme géométrique

Le portage de code Java a été effectué sur la routine *Manhattan healing*[3] qui implémente un algorithme d'union de polygone rectilinéaires par la méthode *sweepline* (ligne de balayage) [104]. Un exemple d'union de polygone rectilinéaire est montré Figure 5.4 où les polygones A, B, C et D sont unis pour former le polygone rectilinéaire E. La routine utilise

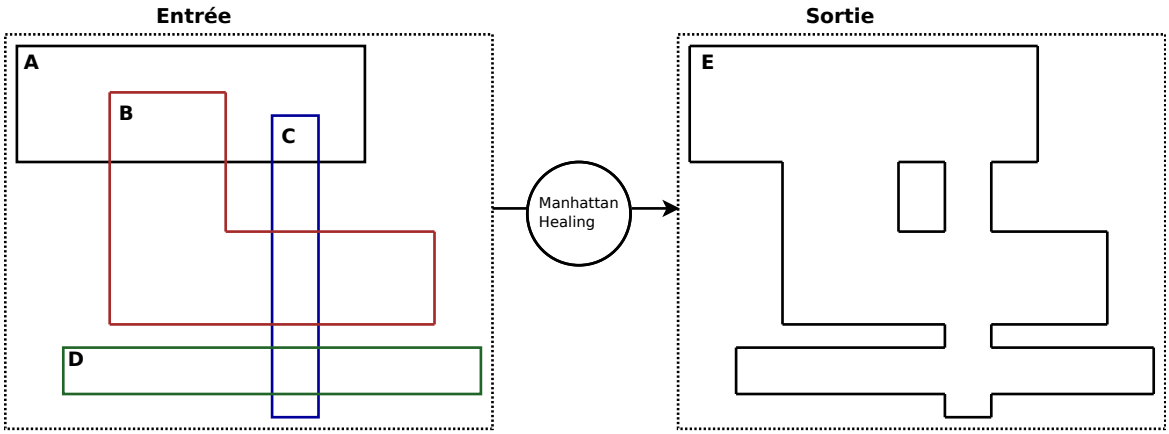


FIG. 5.4: Exemple d'union de polygone rectilinéaire, effectuée par la routine *Manhattan healing*. Les polygones d'entrée A, B, C et D sont unis pour former le polygone de sortie rectilinéaire E

une structure déjà triée en entrée implémentée en Java sous forme un tableau d'objet. L'objet nommé `Event` contient trois champs primitifs, deux entiers 64-bits correspondant aux coordonnées du point et un entier 32-bits caractérisant le type d'intersection. L'équivalent C++ utilisé pour cette structure est un `std::vector` de structure `Event`. La structure de sortie Java est une `ArrayList` de polygones ou un polygone est stocké sous la forme d'un

[3]Le terme *Manhattan* est utilisé en géométrie pour désigner des formes rectilinéaires et fait référence aux rues de Manhattan qui décrivent un réseau rectilinéaire.

tableau primitif d'entier 64-bits correspondant à ses coordonnées. L'équivalent C++ de la structure de sortie est une `std::list` contenant les pointeurs vers les tableaux d'entiers formant les polygones. Le transfert des données entre structure Java et structure native est implémenté en Java en utilisant la classe `Unsafe` pour lire et écrire la mémoire native. Cette implémentation nécessite de connaitre la définition mémoire de la structure native.
Le design d'entrée considéré est une grille carrée formée par des carreaux se touchant. Pour ce type d'entrée, le design de sortie calculé par la routine est un carreau formé par les bords de la grille (cf. Figure 5.5). Pour le design d'entrée considéré, la complexité de l'algorithme est en $O(Nlog(N))$ où N est le nombre d'arêtes du design d'entrée. Les performances de la routine sont calculées par la formule $Nlog(N)/dt$ (où dt est la durée d'exécution) exprimées en op/s (opérations par seconde).
La Figure 5.6 montre les performances obtenues par micro-benchmark pour les implémentations Java et JNI. L'implémentation JNI comporte deux variantes : avec et sans transfert de mémoire. Dans le premier cas le temps de transfert des données est mesuré, dans l'autre les données sont déjà en mémoire native. La version JNI sans transfert permet une amélioration des performances de 233% en moyenne sur les taille de grille considérées alors que pour la version JNI avec transfert, on observe une baisse de -20% par rapport à la version Java. Pour cette routine, le transfert constitue une part importante du temps global de calcul. Ainsi l'utilisation de JNI n'est bénéfique que si l'intégration permet d'utiliser la version sans transfert.

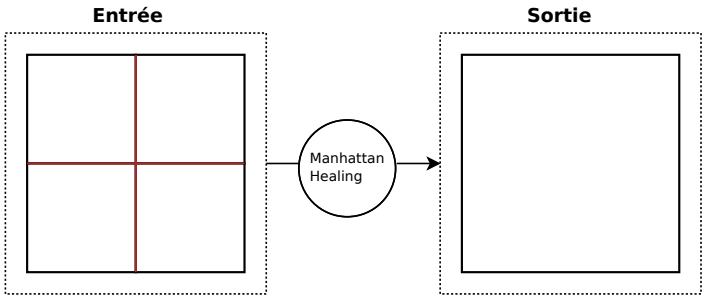


FIG. 5.5: Exemple de design d'entrée considéré pour la routine *Manhattan Healing* (grille de taille 2x2) avec la sortie correspondante. Quelques soit la taille de la grille la sortie est un carreau unique. Les arêtes en rouge représentent les intersections des carreaux. Ce design d'entrée est un cas particulier d'intersections comme ces dernières ne sont pas des points mais des segments. La complexité de l'algorithme pour ce design est en $O(Nlog(N))$ où N est le nombre d'arêtes

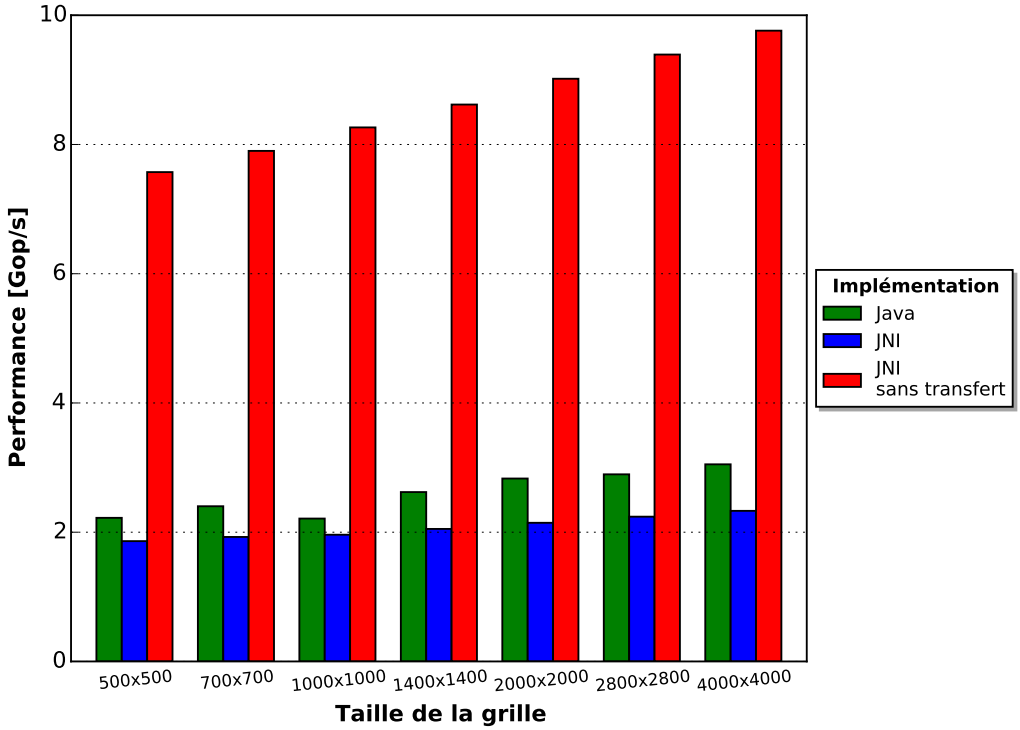


FIG. 5.6: Performance des implémentations de la routine *Manhattan healing* pour différentes tailles de grille

5.4.1.2 Utilisation de la librairie `FFTW`

En dehors du portage de code Java, l'autre moyen de mettre à profit JNI est l'utilisation de librairies natives. JNI permet de tirer partie de librairies natives micro-optimisées pour une architecture donnée, à la différence des librairies Java dont les optimisations sont essentiellement de nature algorithmiques (en dehors des librairies Java qui sont gérées par le runtime).
La librairie FFTW a été testée pour la routine de calcul de transformée de Fourier discrète (TFD) réelle en 2 dimensions calculant en double-précision et en-place (i.e. la matrice d'entrée et la matrice de sortie sont le même conteneur mémoire).
La version JNI utilise FFTW en mode `FFTW_ESTIMATE`. FFTW exécute un plan qui est le couple données/algorithme. Ce mode permet de sélectionner rapidement avec une heuristique simple l'algorithme de TFD adapté pour les données d'entrée, sans tester différents algorithmes. L'algorithme résultant est néanmoins potentiellement sous-optimal. Un plan peut être utilisé pour des données différentes si ces dernières répondent à certaines contraintes (alignement, taille et agencement mémoire) ce qui permet de rentabiliser la création d'un plan, en particulier pour les autres modes nécessitant plus de temps de création du plan. Les implémentations JNI comportent là encore les deux variantes avec et sans transfert des données. Le temps de transfert contient le temps de copie des données vers la mémoire native, le temps de création du plan FFTW et pour finir le temps de copie

du résultat vers la heap. La version sans transfert ne mesure que le temps d'exécution de la routine.
La version Java est une version séquentielle issue de la librairie open-source *JTransforms* utilisée dans différents projets scientifiques Java [181].
La Figure 5.7 montre les résultats de performance des implémentations Java et JNI. La version JNI sans transfert permet un gain de performance en moyenne de 60% par rapport à Java sur les tailles de matrice considérées tandis que la version avec transfert permet un gain de 27%. Là encore le transfert des donnée limite de manière significative les performances. Le gain de performance de 27% peut être considéré comme trop faible pour justifier l'utilisation de méthode native au sein de l'application.

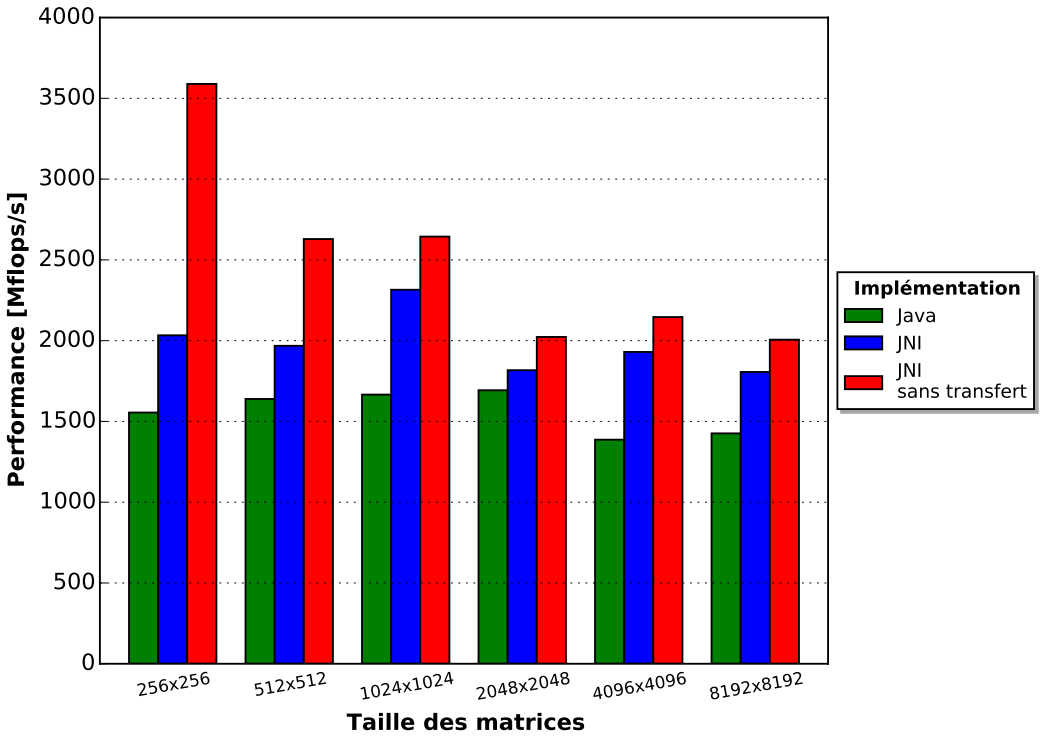


FIG. 5.7: Performance de la TFD en 2 dimensions pour différentes tailles de matrice

5.4.2 Micro-routines utilisant des tableaux primitifs

Le type de routine considéré pour l'analyse des profils de performances sont les micro-routines caractérisées par : l'utilisation de tableaux primitifs comme structure de données d'entrée et de sortie, un coût constant propre quasi-nul et une complexité algorithmique linéaire. Pour ce type de routine, l'impact du coût constant dépend de la quantité de calcul effectuée par appel et requiert une analyse approfondie via le profil de performance pour déterminer la meilleure implémentation (cf. Section 5.5).
Les optimisations appliquées aux routines considérées sont soit de nature asymptotiques, à savoir qu'elles concernent du code dont la criticité dépend des facteurs d'échelle (typique-

ment les opérations de corps de boucle) ; ces dernières sont décrites Section 5.4.2.1 ; soit de nature constante à savoir qu'elles permettent de diminuer le coût constant de la routine ; ces dernières sont décrites Section 5.4.2.2.
L'utilisation des tableaux primitifs pour ces routines permet de s'affranchir des transferts de données pour les méthodes JNI via l'utilisation des sections critiques (cf. Section 5.2.2), et permet ainsi de bénéficier des optimisations asymptotiques de manière optimum.

5.4.2.1 Optimisations asymptotiques

Les optimisations asymptotiques considérées sont de deux types. Le premier type regroupe la vectorisation et le parallélisme d'instruction qui permettent d'augmenter le débit calculatoire par l'augmentation du parallélisme niveau instruction appelé ILP (*Instruction-Level Parallelism*). Le second type est l'augmentation du débit mémoire par l'alignement des données.

La vectorisation considérée cible l'utilisation des instructions AVX permettant d'exécuter 1 opération flottantes en double-précision sur 4 opérandes simultanément. L'extension AVX est décrite plus précisément Section 2.1.1.2 Chap. 2. La qualité de la vectorisation est en grande partie liée à l'empaquetage des données sous forme de vecteur. Le coût d'empaquetage ne doit pas couvrir le bénéfice de la vectorisation. Dans la majeure partie des cas considérés, les données sont systématiquement empaquetées rendant la vectorisation toujours bénéfique.

Le parallélisme d'instruction permet d'exploiter l'exécution out-of-order décrite plus précisément Section 2.1.1.2 Chap. 2. La mise en œuvre de ce parallélisme s'effectue en divisant les chaînes de dépendances séquentielles en plusieurs sous-chaînes de dépendances indépendantes pouvant être exécutées de manière concurrente. Il s'agit donc de paralléliser la chaîne de dépendance. Cette parallélisation s'accompagne d'un épilogue pour fusionner les résultats de chaque sous-chaîne. Comme cette optimisation ne repose pas sur l'utilisation d'instructions spécifiques (comme pour la vectorisation), elle peut être mise en œuvre sans contrainte en Java.
La Table 5.2 montre différentes implémentations de la routine *Somme horizontale*, retournant la somme des éléments d'un tableau de flottants double-précision. Les implémentations vont de l'implémentation scalaire basique à l'implémentation combinant vectorisation et parallélisme d'instruction et maximisant l'ILP. La vectorisation utilise les intrinsics AVX de GCC. La routine *Somme horizontale* est une *réduction* (cf. Section 5.5.1.2) à savoir qu'elle produit un scalaire à partir d'un vecteur en suivant une chaîne de dépendance. Le parallélisme d'instruction est obtenu en divisant la chaîne de dépendances initiale en 4 sous chaînes de dépendances indépendantes numérotées de 0 à 3, la chaîne i sommant les éléments dont l'index modulo 4 vaut i.
Le nombre optimal de sous-chaînes permet de saturer le pipeline et d'obtenir un débit

d'exécution optimal. Il peut être estimé en divisant la latence de la chaîne de dépendances par le débit réciproque de l'unité la plus limitante.

	Sans vectorisation	**Avec vectorisation**
Sans parallélisme d'instructions	`for(i=0;i<n;++i){ sum+=a[i]; }` Degré d'ILP : 1	`for(i=0;i<n;i+=4){ p=mm256_loadu_pd(&(a[i])); sum=mm256_add_pd(sum,p); }` Degré d'ILP : 4
Avec parallélisme d'instructions	`for(i=0;i<n;i+=4){ sum0+=a[i]; sum1+=a[i+1]; sum2+=a[i+2]; sum3+=a[i+3]; }` Degré d'ILP : 4	`for(i=0;i<n;i+=16){ p0=mm256_loadu_pd(&(a[i])); p1=mm256_loadu_pd(&(a[i+4])); p2=mm256_loadu_pd(&(a[i+8])); p3=mm256_loadu_pd(&(a[i+12])); sum0=mm256_add_pd(sum0,p0); sum1=mm256_add_pd(sum1,p1); sum2=mm256_add_pd(sum2,p2); sum3=mm256_add_pd(sum3,p3); }` Degré d'ILP : 16

TAB. 5.2: Implémentations C du corps de boucle de la routine *somme horizontale* en double-précision. Les implémentations utilisent la vectorisation et/ou le parallélisme d'instruction. Le degré d'ILP correspond au nombre d'opérations flottantes du corps de boucle exécutables en parallèle. La combinaison de la vectorization et du parallélisme d'instruction permet d'obtenir un degré d'ILP optimal de 16. L'épilogue de la boucle terminant les itérations et cumulant les sommes indépendantes ne figure pas dans le code

Vectorisation en Java Dans la version de HotSpot du JDK8 l'algorithme d'auto-vectorisation utilisé par C2 [98] ne permet pas de vectoriser les réductions comme la routine *Somme horizontale*. Ainsi le code généré par C2 ne peut mixer parallélisme d'instruction et vectorisation et bénéficier d'un degré d'ILP optimal. Par ailleurs la vectorisation repose sur le déroulage préalable de boucle qui peut être contraint par différents paramètres, en particulier la taille du corps de boucle déroulé, pouvant ainsi constituer une contrainte à la vectorisation. D'autres idiomes comme par exemple l'*induction* (cf. Figure 5.8) ne sont pas vectorisés.

La définition machine indépendante de Java ne permet pas l'utilisation efficace d'intrinsics représentant des instructions vectorielles qui permettraient la vectorisation manuelle de n'importe quelle routine vectorisable. Cette utilisation nécessiterait une extension du langage pour les types et les instructions vectorielles (cf. Section 6.1.1.2 Chap. 6).

Une autre limitation de la vectorisation dans HotSpot concerne l'*aliasing* de pointeurs. L'aliasing de pointeurs survient lorsque deux tableaux désignés par deux pointeurs se che-

vauchent créant ainsi des dépendances de données interdisant la vectorisation. La Table 5.3 montre deux versions de la routine *Addition de tableaux*, avec et sans déphasage. Pour que le JIT puisse garantir qu'il n'y ait pas de dépendances de données et vectoriser cette routine, la distance (en valeur absolue) entre les pointeurs doit être strictement supérieure à la taille du vecteur, soit 4 en double précision. Dans le cas de la routine avec déphasage cette distance vaut (`a`+`aOffset`)-(`b`+`bOffset`). Ainsi les conditions pour qu'il y est une dépendance de donnée sont :

- D'une part que les tableaux soient identiques (`a`=`b`) ;
- D'autre part que le déphasage |`aOffset - bOffset`|=1,2 ou 3.

En pratique l'usage de la routine garantit que (`a`≠`b`) et exclue la première condition. Il n'y a cependant pas de moyen de communiquer cette information au JIT et la seconde condition n'est pas exclue, excepté dans les cas où `a` et `b` sont des constantes de compilation. De même la seconde condition ne peut être exclue que si les offsets sont des constantes de compilation. Dans tous les autres cas la routine ne peut pas être vectorisée.
Cette considération restreint considérablement la gamme des routines vectorisées par le JIT.
Dans la version sans déphasage, la distance est nulle si `a`=`b`, sinon elle est au moins supérieure à la taille des vecteurs. La seconde condition est donc exclue et la version sans déphasage peut être vectorisée sans contrainte.
Le compilateur C2 de HotSpot ne fait pas de retour d'information sur la vectorisation, qui

Version sans déphasage	`static void add(double[] a,`
	`  double[] b, int offset, int n){`
	`for(int i=0;i<n;++i)`
	`  a[offset+i] += b[offset+i];`
	`}`

Version avec déphasage	`static void add(double[] a, int aOffset,`
	`  double[] b, int bOffset, int n){`
	`for(int i=0;i<n;++i)`
	`  a[aOffset+i] += b[bOffset+i];`
	`}`

TAB. 5.3: Versions Java avec et sans déphasage de la routine *addition de tableaux*. La version avec déphasage n'est pas vectorisée par C2 pour cause d'*aliasing* potentiel entre les tableaux `a` et `b`. Même si en pratique l'utilisation de la routine avec déphasage garantit que `a` ≠ `b` et donc l'absence d'aliasing, l'information ne peut être transmise au JIT et la routine n'est pas vectorisée

est donc invisible aux développeurs. Le code assembleur doit être examiné pour déterminer s'il a bien été vectorisé. Ce retour d'information s'avère précieux pour les développeurs car il permet de localiser les failles de performances sans efforts d'analyse trop poussé. La Table 5.4 résume les limitations de la vectorisation en Java, en comparant GCC à C2

```
for(i=0;i<n;++i)
    a[i]=i;
```

FIG. 5.8: Idiome d'*induction*. Non vectorisé par C2 dans Java 8

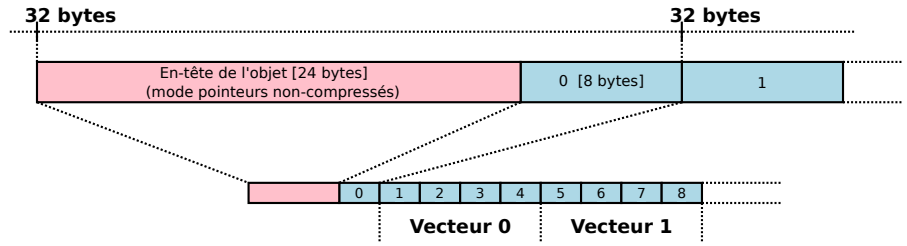
sur les différents aspects soulignés précédemment limitant ou favorisant la vectorisation du code.

	HotSpot C2	**GCC**
Aliasing	Non résolus	Résolus explicitement avec le mot clef `restrict` ou bien avec des vérifications au runtime
Auto-vectorisation	Peu d'idiomes supportés	Nombreux idiomes supportés [1]
Vectorisation manuelle	Non supportée	Intrinsics AVX ou inline d'assembleur
Feedback sur l'auto-vectorisation	Pas de retour d'information	Retour d'information (causes de non-vectorisation, seuils de profitabilité...)

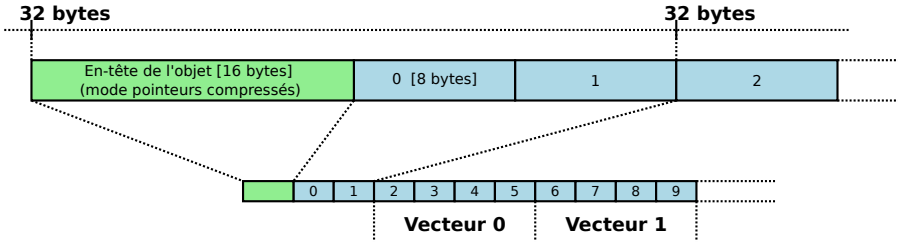
TAB. 5.4: Limitations de la vectorization en Java, comparaison du compilateur dynamique C2 avec GCC

Alignement des données L'objectif de l'alignement est d'assurer des accès alignés dans le code qui permettent de profiter de toute la bande passante mémoire. Typiquement, les transferts mémoire entre L1d et les registres vectoriels se font par mot alignés de 16 bytes (cf. Section 2.1.1.3 Chap. 2). Ainsi, le transfert d'un mot de 32 bytes entre un registre vectoriel `YMM` de 32 bytes et le cache L1 nécessite 2 transferts de 16 bytes dans le cas ou le mot est aligné sur 16 bytes et 3 dans le cas contraire (le mot de 32 bytes est alors à cheval sur 3 mots de 16 bytes). Ainsi l'alignement des vecteurs est nécessaire pour bénéficier de la vectorisation et l'alignement des vecteurs sur 16 bytes permettent généralement de garantir des accès quasi-optimaux.
Deux types d'instructions existent pour les transferts mémoires. Les instructions non-alignées qui ne requiert pas l'alignement des mots transférés et les instructions alignées pour lesquelles les mots transférés doivent être au moins alignés sur la taille des registres vectoriels. Par exemple les mots de 32 bytes formés par 4 doubles doivent être alignés sur 32 bytes pour être transféré depuis ou vers un registre `YMM` (dans le cas contraire le matériel lève une erreur de segmentation). Néanmoins, la différence de performance entre instructions explicitement alignées et instructions non alignées est négligeable lorsque les vecteurs sont au moins alignés sur 16 bytes et que les accès se font majoritairement dans L1d.
En mémoire native l'alignement des données peut être explicitement demandé à l'allocation par exemple avec la fonction `posix_memalign` de la librairie standard C. En Java l'alignement des tableaux primitifs ne peut pas être explicitement demandé à l'allocation.

Le JIT peut néanmoins générer des accès mémoire alignés en utilisant la technique du *peeling* de boucle, consistant à démarrer la boucle vectorisée à partir du premier décalage aligné. Cependant cette technique ne fonctionne pas si la boucle parcourt différents tableaux déphasés (i.e. avec des alignements différents), dans ce cas il n'y a pas de décalage commun permettant l'alignement de tous les accès. Ainsi l'alignement des accès en Java n'est pas garanti ce qui peut occasionner des pertes de performance sur les routines vectorisées. C'est par exemple le cas pour la routine *Addition de tableaux* (cf. Figure 5.3) si l'un des vecteurs est aligné sur 8 bytes et le second sur 16 bytes ou plus (ce cas est abordé plus précisément Section 5.5.2.3).
Pour garantir l'alignement des vecteurs dans les tests, on demande aux objets d'être alignés sur 32-byte en réglant l'option `-XX:ObjectAlignmentInBytes=32` (valant 8 par défaut). Comme montré Figure 5.9, en mode pointeurs compressés (option `-XX:+UseCompressedOops`), qui est le mode par défaut, l'élément d'index 2 est le premier aligné sur 32 bytes ; en mode pointeurs non-compressés (option `-XX:-UseCompressedOops`) il s'agit de l'élément d'index 1. Les routines JNI peuvent ainsi utiliser des données alignées en passant explicitement les indexes de base des tableaux en arguments. Pour les routines Java, le passage des indexes de base en arguments empêche la vectorisation du fait de l'aliasing potentiel, les indexes de bases doivent ainsi être des constantes de compilation pour bénéficier de l'alignement des données.



(a) Mode pointeurs compressés. Le premier élément aligné sur 32 bytes est celui d'index 2



(b) Mode pointeurs non-compressés. Le premier élément aligné sur 32 bytes est celui d'index 1

FIG. 5.9: Structure mémoire sur 64-bits d'un tableau primitif Java de doubles dans les modes pointeurs compressés 5.9(a) et pointeurs non-compressés 5.9(b). Lorsque les objets sont explicitement alignés sur 32 bytes via l'option `-XX:ObjectAlignmentInbytes=32`, le premier vecteur aligné démarre à l'index 2 en mode compressé et 1 en mode non-compressé

5.4.2.2 Optimisations constantes

Les optimisations constantes considérées sont l'inlining s'appliquant aux méthodes Java et l'utilisation de mémoire native s'appliquant aux méthodes natives.

Inlining des méthodes Java-pures L'amélioration due à l'inlining provient de deux facteurs distincts. D'une part l'élimination du coût d'appel et d'autre part la spécialisation du code au site d'appel. La spécialisation du code est un aspect pouvant rendre difficile la comparaison entre méthodes natives et méthodes Java comme les performances vont dépendre du site d'appel et de son potentiel de spécialisation à l'inlining. Par exemple, un site d'appel où les paramètres d'entrée sont constants peut avoir un fort potentiel d'optimisation dû à la propagation de constante post-inlining. L'inlining possède également un fort potentiel d'optimisation concernant l'élimination de calcul redondants dans le contexte d'appel et la méthode inlinée.
Pour les micro-routines considérées, le potentiel de spécialisation est nul et la complexité de la routine est invariante par inlining. Ainsi les méthodes Java bénéficie de l'inlining comme une optimisation constante diminuant le coût constant par appel. Le bénéfice de l'inlining dépend dans ce cas du site d'appel considéré qui détermine le coût d'appel. Pour les tests effectués, la méthode est inlinée dans la méthode contenant les timers et mesurant la durée d'exécution (cf. Section 4.1). Le gain de performance provenant de l'inlining n'impacte que les très faibles quantités de calcul et manière peu significative. Ainsi l'impact de l'inlining comme optimisation constante est jugé négligeable et n'est pas évalué dans les tests.

Utilisation de la mémoire native pour les méthodes JNI Comme vu Section 5.4.1, la mémoire native permet d'améliorer grandement les performances en évitant le transfert des données Java en mémoire native. Dans le cas des micro-routines utilisant des types primitifs, le transfert de données Java n'est pas nécessaire, et l'utilisation de sections critiques est mise en œuvre (cf. Section 5.2.2). L'utilisation de mémoire native permet néanmoins de diminuer le coût constant par appel en évitant l'exécution des fonctions de rappel JNI pour manipuler des tableaux primitifs depuis la heap (cf. Section 5.2.3). La comparaison des performances entre une routine utilisant de la mémoire native et une routine de la mémoire heap permet de mesurer le coût des fonctions de rappel JNI.

5.5 Profils de performance des micro-routines

Les micro-routines considérées sont décrites et caractérisée Section 5.5.1. Les profils de performance des différentes routines sont ensuite analysés Section 5.5.2.

5.5.1 Description des micro-routines

Le choix des micro-routines s'est porté sur des routines élémentaires pouvant être catégorisées en 3 types : les routines vectorielles (cf. Section 5.5.1.1), les réductions (cf. Section 5.5.1.2) et les routines mixtes (cf. Section 5.5.1.3).
Les routines considérées sont de complexité linéaire par rapport à la taille des données, avec un coût constant propre négligeable. Ces dernière montrent un comportement plus réguliers du profil de performance ainsi qu'une sensibilité aux faibles quantité de calcul. Les routines calculent en double-précision ce qui, en contrepartie d'une meilleure précision, d'une part augmente la taille des données en mémoire rendant plus précoce les effets de cache lorsque la quantité de calcul augmente et d'autre par réduit le bénéfice de la vectorisation (cf. Section 5.4.2.1).
Pour chaque type de micro-routines on considère deux routines qui se distinguent par leur nature soit *memory-bound* ou bien *CPU-bound* déterminée par leur intensité arithmétique (cf. Section 2.1.2.3 Chap. 2). *Memory-bound* signifie que les opérations limitant majoritairement les performances de la routine sont les accès mémoire tandis que *CPU-bound* signifie qu'il s'agit des opérations arithmétiques.
L'intensité arithmétique évolue en fonction de l'implémentation, les valeurs considérée sont celles de l'implémentation scalaire sans déroulage de boucle. L'intensité arithmétique théorique pour l'architecture considérée est de 1/6 flops/byte. Ainsi une routine ayant une intensité arithmétique au dessus (resp. au dessous) est *CPU-bound* (resp. *memory-bound*) pour cette architecture. La Table 5.5 résume les caractéristiques des différentes micro-routine.

Micro-routines	Intensité arithmétique [Flop/Byte]	Caractéristiques
Addition de tableaux Additionne le second tableau au premier passé en paramètre	1/24	Routine vectorielle *memory-bound*. Vectorisée par le JIT
Exponentielle Calcul la fonction exponentielle sur chaque élément du tableau d'entrée et écrit le résultat dans le tableau de sortie	1/2	Routine vectorielle *CPU-bound*. Non-vectorisée par le JIT
Somme horizontale Retourne la somme des éléments du tableau d'entrée	1/8	Réduction *memory-bound*. Non-vectorisée par le JIT
Aire de polygone Retourne l'aire du polygone représenté par les coordonnées des points le constituant	1/4	Réduction *CPU-bound*. Non-vectorisée par le JIT
Horner par-coefficient Calcul l'image par un polynôme quelconque (définie par son tableau de coefficient) de chaque élément du tableau d'entrée et écrit le résultat dans le tableau de sortie. Le parcours dans la boucle externe se fait par coefficient	$\frac{1}{12+4(\frac{1}{N}+\frac{1}{D})}$ Avec N le nombre d'éléments et D le degré du polynôme	Routine mixte *memory-bound*. Vectorisée par le JIT
Horner par-élément Idem *horner par-coefficient* excepté que le parcours dans la boucle externe se fait par éléments	$\frac{1}{4+\frac{8}{D}}$	Routine mixte *CPU-bound* (pour $D > 4$). Non-vectorisée par le JIT

TAB. 5.5: Descriptif des micro-routines considérées. Sous fond bleu figurent les routines vectorielles, sous fond vert les réductions et sous fond rouge les routines mixtes. L'intensité arithmétique à l'équilibre pour l'architecture considérée est de 1/6 Flop/Byte (cf. Section 2.1.2.3 Chap. 2). L'intensité arithmétique des différentes routines est calculée pour la version scalaire sans déroulage de boucle

5.5.1.1 Routines vectorielles

Les routines vectorielles sont les plus adaptées pour la vectorisation car elles expriment un parallélisme intrinsèque. Ce sont les routines du type `out[i]=fonction(in[i])` ; à savoir qu'elles calculent pour chaque valeur d'entrée une valeur de sortie, le calcul étant identique pour chaque élément et indépendant des autres. Pour ce type de routine, lorsque les données sont déjà empaquetées en mémoire sous forme de vecteurs, la vectorisation est toujours profitable.
Contrairement aux réductions (cf. Section 5.5.1.2), le parallélisme d'instruction est déjà exprimé dans les routines vectorielles du fait de l'indépendance des itérations[4]. Il peut y avoir, si la boucle n'est pas déroulée, des anti-dépendances mais celles ci sont résolues au niveau matériel par renommage de registre (cf. Section 2.1.1.2 Chap. 2).

[4]Cette indépendance suppose le non-aliasing entre les tableaux `in` et `out`.

On considère deux routines vectorielles. La première étant l'*Addition de tableaux* effectuant la somme de deux tableaux en-place (`inout[i]=inout[i]+in[i]`) et ayant une intensité arithmétique de 1/24 (*memory-bound*). La seconde étant la routine *exponentielle* calculant l'exponentielle du tableau d'entrée dans le tableau de sortie (`out[i]=exp(in[i])`) et ayant une intensité arithmétique de 1/2 (*CPU-bound*).

Routine *exponentielle* L'implémentation Java utilise la fonction `exp` de la classe `FastMath` de la librairie *Apache Commons Math* [167]. Celle-ci est en moyenne 3 fois plus rapide que la version utilisant la classe `StrictMath` du JDK. La version native utilise une implémentation assembleur vectorisée et micro-optimisée pour Sandy Bridge générée par `PeachPy` [38], librairie python pour la génération de routines assembleurs[5]. Les deux implémentations respectent le standard IEEE 754 [18] pour la fonction exponentielle ce qui garantit que le gain de performance d'une implémentation ne se fait pas au détriment d'une perte de précision numérique.
L'implémentation Java de la routine *exponentielle* n'est pas vectorisée par le JIT. C'est, de manière similaire, le cas pour d'autres fonctions usuelles fournies par la classe `Math` du JDK (`sin`, `cos`, `sqrt`, `log`...) lorsqu'elles sont appliquées de manière itérative sur des données empaquetées en mémoire. La vectorisation de la routine nécessite deux optimisations préalables :

- L'inlining de la méthode (en l'occurrence `exp`) ;
- Le déroulage de boucle d'un facteur au moins égal à la taille des vecteurs (en l'occurrence 4).

Concernant l'inlining, celui de `exp` doit être forcé car son empreinte mémoire (niveau bytecode) dépasse la limite fixée par les heuristiques d'inlining de HotSpot, soit 325 bytes par défaut[6].
Concernant le déroulage de boucle, une première contrainte concerne le nombre de nœuds constituant le corps de boucle dans la représentation intermédiaire de C2 ; il ne doit pas après déroulage dépasser un certain seuil fixé par le paramètre `-XX:LoopUnrollLimit` qui vaut 60 par défaut. Pour permettre le déroulage de boucle de la routine, ce seuil a donc été augmenté.
Malgré les ajustements pour garantir l'inlining et le déroulage de boucle, la routine *exponentielle* n'est pas vectorisée ce qui peut : soit provenir d'une autre contrainte sur le déroulage de boucle ; soit provenir d'une contrainte de vectorisation liée au corps de boucle. Après vérification il s'avère que la boucle est bien déroulée, il s'agirait donc plus vraisemblablement d'une contrainte sur le corps de boucle.

[5] La librairie Yeppp ! [56] fournit une API Java basée sur JNI permettant de bénéficier de ces implémentations micro-optimisées

[6] Dans le cas où un site d'appel chaud est rencontré, la taille limite est fixée par le paramètre `-XX:FreqInlineSize` valant 325 bytes par défaut. Dans le cas d'un site d'appel froid il s'agit du paramètre `-XX:MaxInlineSize` valant par défaut 35 bytes

5.5.1.2 Réductions

Les *réductions* sont les routines réduisant un tableau d'entrée à une valeur scalaire en sortie au travers d'une chaîne de dépendance. Les réductions consistant en une chaîne de dépendance, elles peuvent par conséquent être plus difficiles voire impossibles à paralléliser et donc à vectoriser.
Les optimisations sur les réductions consistent en la division de la chaîne de dépendances en plusieurs sous-chaînes traitant chacune une partition du tableau d'entrée et fusionnant les résultats à l'arrivée. Cette division permet d'exprimer à la fois le parallélisme vectoriel et le parallélisme d'instruction (cf. Section 5.4.2.1). Contrairement au routines vectorielles (cf. Section 5.5.1.1) où le parallélisme d'instruction peut être exprimé par déroulage de boucle, ce dernier doit être sémantiquement exprimé au niveau du code généré dans le cas des réductions.
On considère deux réductions. La première est la *somme horizontale* retournant la somme des éléments d'un tableau (cf. Section 5.4.2.1), qui a une intensité arithmétique de 1/8 (i.e. *memory-bound*). La seconde est la routine *aire de polygone* retournant l'aire d'un polygone, qui a une intensité arithmétique de 1/4 (i.e. *CPU-bound*).

Routine *aire de polygone* Concernant la routine *aire de polygone*, les contraintes applicatives nécessitent des polygones encodés par un tableau de double sous la forme `[x0, y0, x1, y1...]` ou le couple `(xi, yi)` représente les coordonnées du i-ème point du polygone. Cette représentation des données est qualifiée de *tableau de structures* [42] où une structure est dans le cas présent un couple de coordonnées i.e. un point. Elle n'est généralement pas la mieux adaptée pour la vectorisation car elle peut nécessiter des opérations additionnelles pour empaqueter les données dans les registres vectoriels. À l'inverse, un polygone encodé par un tableau de double sous la forme d'une *structure de tableaux* [42], à savoir `[x0, x1... y0, y1...]`, contient des données déjà empaquetées pouvant être directement chargées dans les registres vectoriels.
La Figure 5.10 montre le graphe de *dataflow* du corps de boucle de deux implémentations vectorisées du calcul d'aire de polygone, l'une utilisant une structure de tableaux et l'autre un tableau de structures. À chaque itération les éléments e_i associés à des segments successifs du polygone sont calculés puis ajoutés aux accumulateurs acc_i. En fin de routine les valeurs de chaque accumulateur sont sommées pour former la valeur de retour.
La version utilisant une structure de tableaux permet de calculer 4 éléments e_i (`[e0 e1 e2 e3]`) par itération avec une intensité arithmétique de 1/8 Flop/Byte, ce qui la rend donc *memory-bound*. Notons que l'implémentation proposée n'est pas optimale comme la plupart des coordonnées sont chargées deux fois (16 valeurs sont chargées à chaque itération sur 10 utiles) et que les accès ne sont pas tous alignés. La version avec tableau de structures permet de calculer seulement 2 éléments par itération mais avec une meilleure efficacité mémoire comme il n'y a pas d'information chargée redondante. Son intensité arithmétique de 3/8 Flop/Byte est également plus élevée. Les défauts de cette implémentation, à savoir

le faible parallélisme vectoriel est le surcoût des opérations d'empaquetage (`VPERM2F128` et `VPERMILBD`), la rendent néanmoins moins performante que l'autre version.

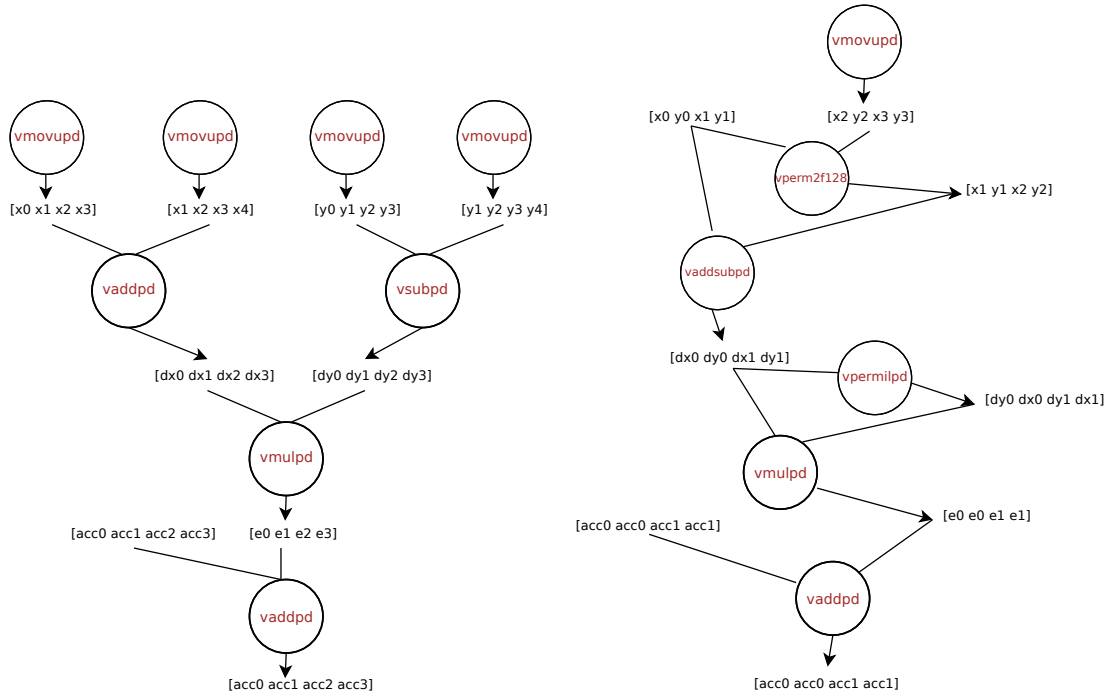


FIG. 5.10: Impact de la structure de donnée sur la vectorisation. Les figures montrent les graphes de dataflow en assembleur du corps de boucle de deux implémentations vectorisées du calcul d'aire de polygone. L'implémentation 5.10(a) utilise en entrée une structure de tableaux de la forme `[x0, x1... y0, y1...]` facilitant la vectorisation. L'implémentation 5.10(b) utilise en entrée un tableau de structures de la forme `[x0, y0, x1, y1...]` rendant la vectorisation plus fastidieuse. C'est le cas pour la routine *aire de polygone* étudiée

5.5.1.3 Routines mixtes

Les routines mixtes sont les routines vectorielles effectuant également des réductions. C'est le cas de la routine *Horner* considérée, qui retourne l'image par un polynôme variable de chaque élément d'un tableau d'entrée, l'image étant calculée par la méthode d'Horner (cf. Table 5.6). Elle possède deux facteurs d'échelle qui sont le nombre d'élément d'entrée et le degré du polynôme (correspondant à l'index de son coefficient de plus haut degré non nul).

Pour deux raisons principales, les tests ne considèrent que le nombre d'élément comme facteur d'échelle variable, le degré du polynôme étant fixé à 64 :

- La première raison est qu'en terme d'utilisation, le passage à l'échelle se fait plus classiquement sur le nombre de données d'entrée que sur le degré du polynôme ;
- La seconde raison est que les profils de performances à 2 dimensions sont plus fasti-

dieux à mesurer.

La routine *Horner* est qualifiée de routine mixte car c'est une routine vectorielle du point de vue du tableau des éléments d'entrée (du type `out[i]=fonction(in[i])`) et c'est par ailleurs une réduction du point de vue du tableau des coefficients (si l'on fixe une entrée, on obtient la fonction qui pour n'importe quel polynôme calcule l'image de l'entrée fixée, cette fonction étant une réduction).

Les deux facteurs d'échelle permettent de définir deux versions de la routine, *par-élément* et *par-coefficient*, dont les algorithmes sont exposés Figure 5.6.

La première version *Horner par-coefficient* parcourt le tableau de coefficient dans la boucle externe. Ainsi chaque étape de la réduction est calculée pour tous les éléments d'entrée. Une implémentation par bloc du parcours par coefficient, parcourant des blocs d'éléments dont la taille garantit qu'ils logent dans le cache L1d, est également exposée (cf. Section 5.5.2.3) afin d'améliorer la localité mémoire et les performances.

La seconde version *Horner par-élément* parcourt le tableau des éléments de la boucle externe et effectue la réduction pour chaque élément. Contrairement à la *somme horizontale*, la réduction par la méthode d'Horner n'est pas vectorisable. Le bénéfice principal de cette version provient de l'augmentation de l'intensité arithmétique (cf. Table 5.5) la rendant *CPU-bound* contrairement à la version précédente *memory-bound*.

Parcours par éléments	**Parcours par coefficients**

for $i \leftarrow 0, n-1$ **do**
  $y_i \leftarrow c_d$
  for $j \leftarrow d-1, 0$ **do**
  $y_i \leftarrow y_i * x_i + c_j$
  end for
end for
	for $i \leftarrow 0, n-1$ **do**
	  $y_i \leftarrow c_d$
	end for
	for $j \leftarrow d-1, 0$ **do**
	  for $i \leftarrow 0, n-1$ **do**
	  $y_i \leftarrow y_i * x_i + c_j$
	  end for
	end for

TAB. 5.6: Algorithme d'Horner pour le calcul polynomial avec parcours par éléments (i.e. parcourant les éléments d'entrée en premier) et parcours par coefficients (i.e. parcourant les coefficients polynomiaux en premier). Les éléments d'entrée sont notés $(x_i)_{i=0,\dots,n-1}$; les éléments de sortie $(y_i)_{i=0,\dots,n-1}$ et les coefficients du polynôme de degré d $(c_i)_{i=0,\dots,d}$

5.5.2 Analyse des profils de performance

Pour chaque micro-routine décrite Section 5.5 correspond un graphe contenant les profils de performance des différentes implémentations.

Fenêtre et résolution du graphe de mesures Le profil de performance d'une implémentation représente ses performances en fonction de la quantité de calcul effectuée par appel (cf. Section 5.3). L'augmentation de la quantité de calcul se fait par l'augmentation de facteur d'échelle à savoir la taille des données d'entrée. L'axe des abscisses correspond

ainsi à la quantité de calcul exprimée en Flops. Pour chaque routine, les graphes contiennent également les seuils de quantité de calcul pour lesquels la taille mémoire des données d'entrée et de sortie atteint celle des différents niveaux de caches (cf. Figure 2.3 Section 2.1.1.3 Chap. 2) afin d'observer l'impact du ralentissement des accès mémoire sur les performances de la routine. Les performances sont mesurées selon la méthodologie détaillée Chap. 3 (la méthode contenant les timers n'utilisant pas de boucle d'itération).
L'intervalle des quantités de calcul couvre les tailles des données allant de la taille minimale 1 à la taille excédant la taille du cache L3 (soit 6MB). Les points de mesure correspondent aux puissances de 2 et sont tracés à l'échelle logarithmique. Les valeurs aux points intermédiaires peuvent être mesurées en augmentant la résolution du profil de performance sur un sous intervalle ou bien interpolées linéairement de manière approximative (des oscillations de faibles amplitudes pouvant exister entre deux points de mesure). Le comportement au delà peut être extrapolé à la baisse en considérant l'augmentation croissante des défauts de cache, y compris du cache TLB survenant à partir d'un certain seuil.

Nommage des implémentations Une implémentation est définie par son type : *Java* ou *JNI* ; par ses optimisations asymptotiques : vectorisation et parallélisme d'instruction (cf. Section 5.4.2.1) et par ses optimisations constantes : *mémoire native* pour les implémentations *JNI*. Pour les implémentations *Java*, l'effet de l'inlining étant négligeable, cette dernière est désactivée (cf. Section 5.4.2.2).
Le label "AVX" désigne une implémentation vectorisée et le label "OOO" désigne une implémentation avec déroulage de boucle et plusieurs chaînes de dépendances pour exprimer le parallélisme d'instruction. Les implémentations JNI avec mémoire native sont précédées du label "Native". Par exemple, l'implémentation "Native JNI+AVX+OOO" désigne une implémentation *JNI* vectorisée avec plusieurs chaînes de dépendances utilisant de la mémoire native et l'implémentation "Java+AVX", une implémentation *Java* vectorisée.

La Section 5.5.2.1 propose une description générale de l'allure des profils de performance en fonction du type de routine. La Section 5.5.2.2 montre comment sélectionner, à partir des profils de performance, la meilleur implémentation. Enfin la Section 5.5.2.3 fait une analyse des différentes optimisations prises isolément afin de comprendre l'allure de chacun des profils.

5.5.2.1 Allures générales des profils de performance

Les profils de performance correspondant aux différentes routines sont liés Table 5.7. On distingue deux allures distinctes correspondant soit aux routines *memory-bound*, soit aux routines *CPU-bound* (cf. Table 5.5.1).
Les routines *memory-bound* sont caractérisées par une sensibilité plus accrue des performances à l'augmentation de la quantité de calcul, linéairement reliée à la taille des données. On observe ainsi pour ces routines une chute des performances dès que la taille des données

Addition de tableaux	Figure 5.11
Exponentielle	Figure 5.12
Somme horizontale	Figure 5.13
Aire de polygone	Figure 5.14
Horner par-coefficient	Figure 5.15
Horner par-élément	Figure 5.16

TAB. 5.7: Figures correspondant aux profils de performance des micro-routines

d'entrée et de sortie excède la taille du cache de données L1d (soit 32KB). La baisse de performance étant d'autant plus importante que l'intensité arithmétique est faible. C'est le cas des routines *Addition de tableaux* avec une baisse d'environ -50% (cf. Figure 5.11), *Somme horizontale* avec une baisse d'environ -45% (cf. Figure 5.13) et *Horner par-coefficient* avec une baisse d'environ -47% (cf. Figure 5.15). Ces chutes de performance impactent les implémentations avec un fort ILP rendant les opérations mémoire d'avantage critiques. Pour les routines *CPU-bound*, cette baisse de performance est soit faible, c'est le cas de la routine *Aire de polygone* (cf. Figure 5.14), soit inexistante, c'est le cas des routines *Exponentielle* (cf. Figure 5.12) et *Horner par-élément* (cf. Figure 5.16). Les routines avec une forte intensité arithmétique comme *Exponentielle* sont insensibles à l'augmentation de la quantité de calcul.

Pour toutes les routines on observe une phase de décroissance des performances lorsque la quantité de calcul diminue du fait de l'augmentation de la part du coût constant (cf. Section 5.3.2). La pente de décroissance étant d'avantage importante que le coût constant relatif est élevé. Ainsi, et en particulier pour les routines *CPU-bound*, on observe un pic de performance pour la quantité de calcul située juste avant le seuil L1d ; à cette valeur la quantité de calcul est suffisante pour couvrir le coût constant et les accès mémoire se font tous dans le cache L1d. La Table 5.8 résume les pics de performance de chaque routine avec les quantités de calcul correspondantes et le pourcentage de performance crête CPU atteint. On note que les performances de la routine *Horner par-élément* avec vectorisation et parallélisme d'instruction atteignent la crête CPU. Pour la routine *Horner par-coefficients* on observe un déphasage du pic de performance pour des implémentations vectorisées. Un pic plus précoce caractérise des défauts de cache également plus précoces ce qui suppose une pression plus élevée sur le cache. Ce phénomène peut provenir de l'alignement des données. On observe également pour cette routine des oscillations pour des quantités de calcul excédant le cache L3 dues à des effets de cache.

Routine	Pic [Gflops/s]	% crête CPU (29,6 Gflops/s)	Quantités de calcul correspondantes [Flops]
Addition de tableaux	5,0	17%	1K
Exponentielle	16,3	55%	de 32K à 8M
Somme horizontale	9,8	33%	2K
Aire de polygone	8,5	29%	de 4K à 64K
Horner par-coefficient	11,3	38%	256K
Horner par-élément	27,5	93%	de 128K à 32M

TAB. 5.8: Pics de performance des différentes routines avec les quantités de calcul correspondantes. Les implémentations prises en compte sont les implémentations Java ou JNI sans usage de mémoire native. Les routines *CPU-bound* sont caractérisées par un pic de performance persistant sur tout un intervalle

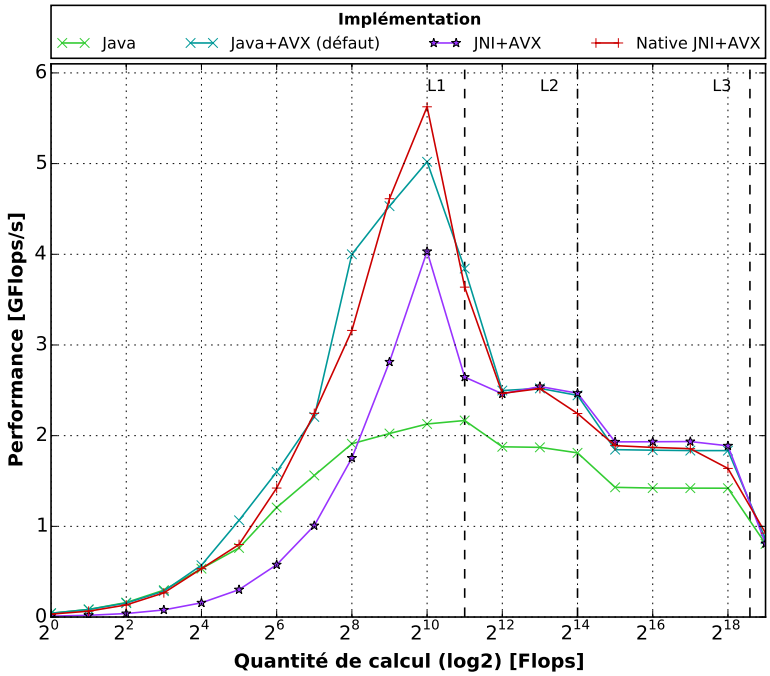


FIG. 5.11: Profils de performance de la routine *Addition de tableaux*. La version Java non vectorisée est obtenue en désactivant la vectorisation via l'option `-XX:UseSuperWord=false`

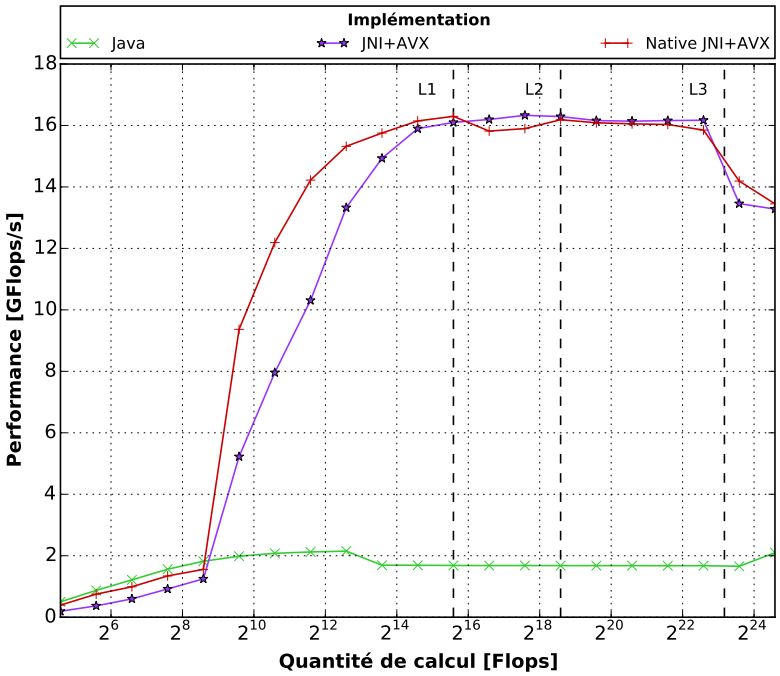


FIG. 5.12: Profils de performance de la routine *Exponentielle*

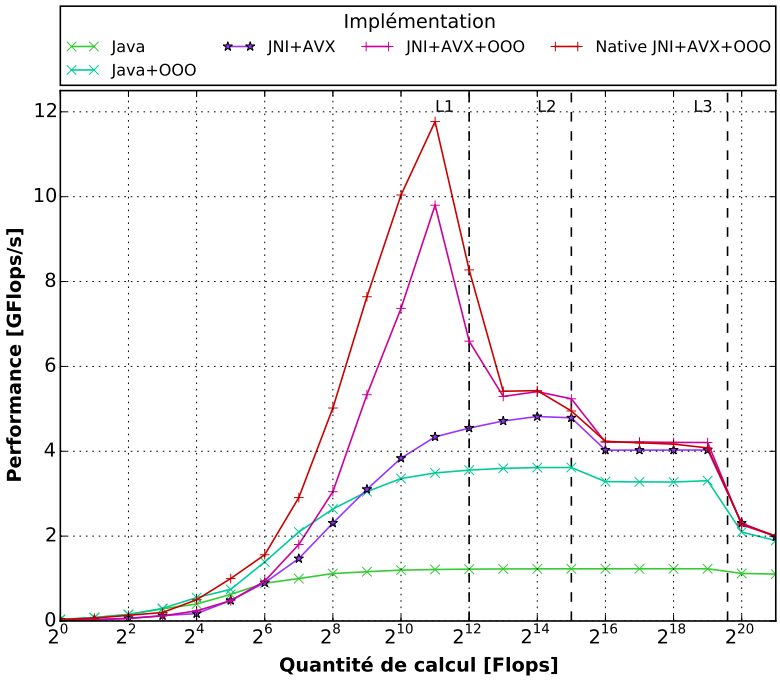


FIG. 5.13: Profils de performance de la routine *Somme horizontale*

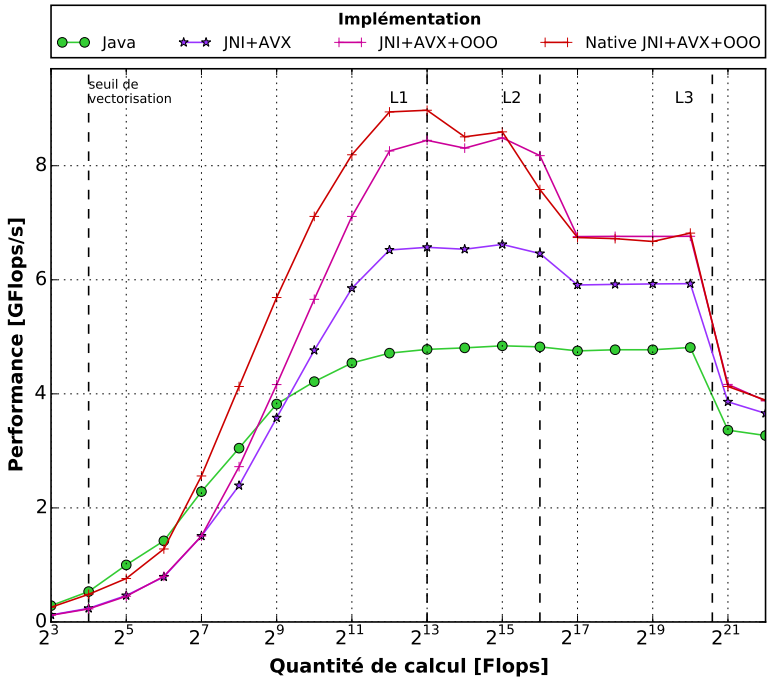


FIG. 5.14: Profils de performance de la routine *Aire de polygone*

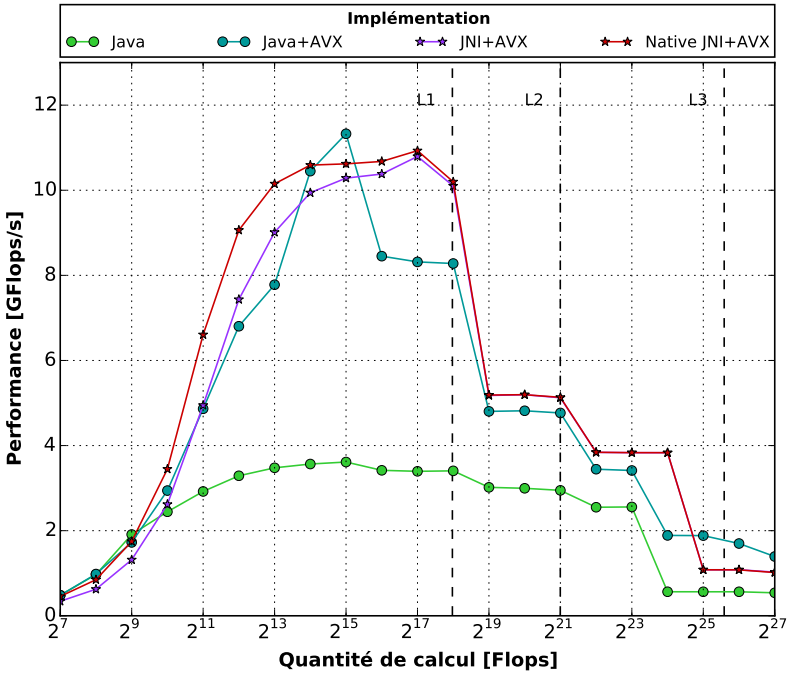


FIG. 5.15: Profils de performance de la routine *Horner par-coefficients*. La version Java non vectorisée est obtenue en désactivant la vectorisation via l'option `-XX:UseSuperWord=false`

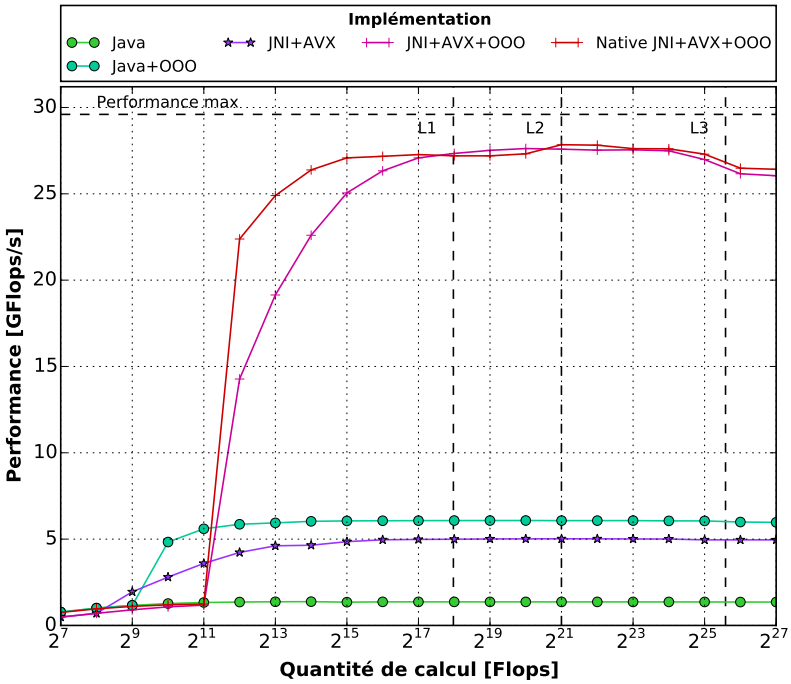


FIG. 5.16: Profils de performance de la routine *Horner par-éléments*

5.5.2.2 Choix de la meilleure implémentation

En supposant que l'intervalle des quantités de calcul couvert par l'application est connu avant l'exécution de la routine, le profil de performance permet de sélectionner la routine la plus performante sur cet intervalle. Il peut également servir à calibrer la taille des paquets dans le dataflow applicatif.
L'utilisation intensive des micro-routines au sein de l'application peut avoir deux tendances opposées : des appels haute-fréquence de micro-routine exécutant de faibles quantités de calcul ou bien des appels basse-fréquence exécutant des quantités de calcul plus élevées. Cette tendance d'utilisation permet de déterminer quelle implémentation choisir en se basant sur le profil de performance.
Les implémentations éligibles pour cette sélection sont les implémentations Java, et JNI sans usage de mémoire native. L'utilisation de la mémoire native modifiant profondément le paradigme de programmation, les implémentations en faisant usage sont écartées. Son impact est analysé plus loin Section 5.5.2.3.
La Figure 5.17 montre pour chaque routine les accélérations obtenues par la meilleure implémentation JNI contre la meilleure implémentation Java (les valeurs négatives représentent en valeur absolue l'accélération des implémentations Java par rapport aux implémentations JNI).
Comme attendu (cf. Section 5.3.2.2), au dessous d'un certain seuil de quantité de calcul les implémentations Java l'emportent systématiquement sur les implémentations JNI du fait du coût d'intégration des fonctions natives.
Les routines *CPU-bound* (*Exponentielle*, *Aire de polygone* et *Horner par-élément*) possèdent un seul point de croisement au delà duquel l'implémentation JNI l'emporte systématiquement. Cette domination s'explique pas l'absence d'optimisations asymptotiques effectuées par le JIT pour les routines vectorielles non vectorisées comme *Exponentielle* ainsi que la non-vectorisation des réductions (cf. Section 5.4.2.1). On constate par ailleurs que les points de croisement pour ces routines sont relativement bas, l'intervalle sur lequel l'implémentation Java l'emporte étant minime par rapport à la largeur du domaine considéré.
La routine *Somme horizontale* possède également un point de croisement situé plus en amont, sa nature *memory-bound* limite néanmoins le bénéfice des optimisations asymptotiques à un intervalle restreint. La routine *Addition de tableaux* est à l'avantage de Java comme elle bénéficie de la vectorisation au même titre que l'implémentation JNI. Enfin pour la routine *Horner par-coefficient*, également vectorisée par le JIT, il y a plusieurs points de croisement du fait d'un déphasage du pic de performance entre les deux implémentations (cf. Section 5.5.2.1) cependant les performances globales entre les deux implémentations restent équilibrées comme elles bénéficient toutes deux de la vectorisation. La Table 5.9 résume les points de croisement avec pour chaque intervalle l'accélération moyenne obtenue.

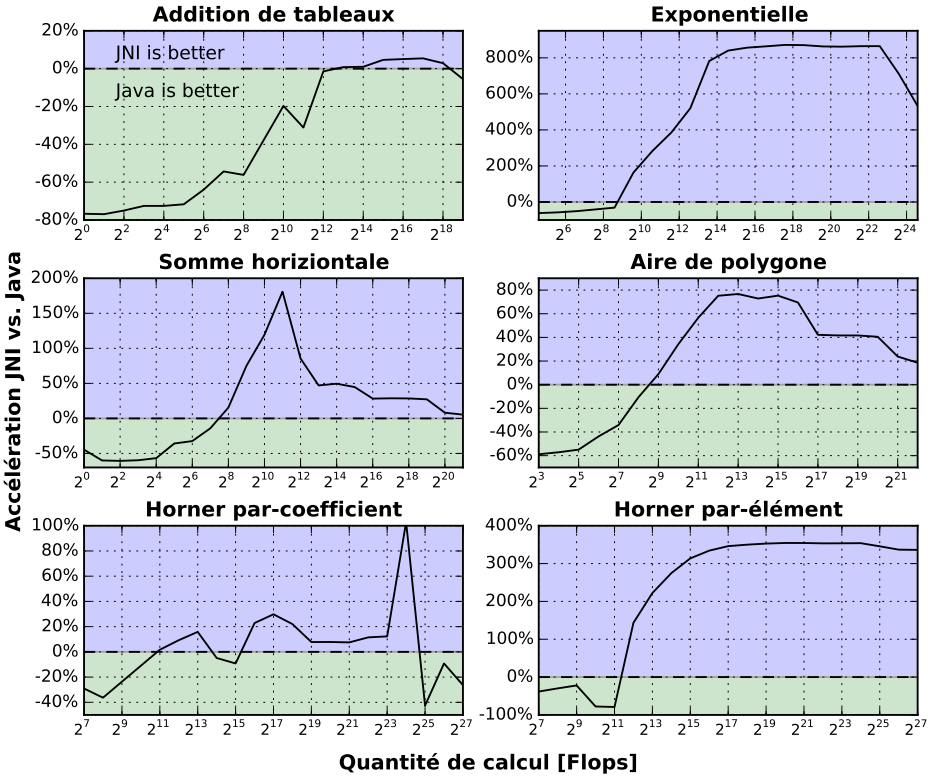


FIG. 5.17: Accélération (en %) de la meilleure implémentation JNI contre la meilleure implémentation Java. Lorsque la courbe est sous fond vert (resp. bleu) l'implémentation Java (resp. JNI) est plus performante (les valeurs négatives représentent en valeur absolue l'accélération des implémentations Java par rapport aux implémentations JNI)

De manière plus générale, pour des micro-routines de nature similaire (vectorisables et de complexité linéaire par rapport à la taille des tableaux d'entrée), deux caractéristiques permettent de prédire l'allure du profil de performance :

- Si l'idiome vectorisable utilisé dans la routine est bien vectorisé par le JIT. S'il ne l'est pas, comme c'est le cas pour les *réductions*, une implémentation JNI permettra de décupler les performances. Le gain sera d'avantage important si la vectorisation est combinée à du parallélisme d'instruction ;
- La nature *memory-bound* ou *CPU-bound* de la routine : si la routine est *memory-bound* le pic de performance sera localisé autour du point maximisant la quantité de calcul et pour lequel les données logent dans le cache de niveau 1.

L'information des points de croisement (cf. Table 5.9) peut être utilisée pour générer la version la plus performante en fonction de l'usage applicatif. Dans le cas d'appels haute-fréquence exécutant de faibles quantités de calcul, les performances attendues se situent vers les basses valeurs du profil de performance ; elles se situent dans les hautes valeurs dans le cas d'appels plus basse-fréquence exécutant des faibles quantités. La détermination préalable de la fréquence d'exécution pour un intervalle de quantité de calcul permet de sélectionner (en utilisant par exemple du polymorphisme) la version maximisant les perfor-

Routine	Intervalle [Flops] : meilleure implémentation (% accélération)
Addition de tableaux	[1, 4K] : Java (55%) [4K, 256K] : JNI (3%) [256K, 1M] : Java (5,5%)
Exponentielle	[32, 512] : Java (49%) [512, 32M] : JNI (700%)
Somme horizontale	[1, 128] : Java (45%) [128, 2M] : JNI (53%)
Aire de polygone	[8, 256] : Java (43%) [256, 2M] : JNI (48%)
Horner par-coefficient	[128, 2K] : Java (25%) [2K, 16K] : JNI (9%) [16K, 32K] : Java (7%) [32K, 16M] : JNI (25%) [16M, 128M] : Java (26%)
Horner par-élément	[128, 2K] : Java (50%) [2K, 128M] : JNI (320%)

TAB. 5.9: Points de croisement avec pour chaque intervalle la meilleure implémentation (Java ou JNI) et l'accélération moyenne obtenue

mances. Le cas le plus simple étant lorsque l'intervalle exécuté est inclus dans un intervalle où une implémentation domine. Dans les autres cas, des calculs de moyenne sur l'intervalle en question peuvent être mis en œuvre.

5.5.2.3 Impact des différentes optimisations

Vectorisation Comme vu Section 5.4.2.1, le JIT de HotSpot, dans sa version Java 8, ne vectorise pas les idiomes de réduction. Ainsi les versions Java des routines *somme horiziontale*, *aire de polygone* et *horner par-élément* ne sont pas vectorisées. Par ailleurs les idiomes du type `out[i]=fonction(in[i])`, nécessitant préalablement inlining et déroulage de boucle, ne sont pas vectorisés par le JIT. C'est du moins le cas des fonctions usuelles (`sin`, `cos`, `exp`, `sqrt`, `log`...) fournies par la classe `Math` du JDK. Par conséquent la routine *exponentielle* considérée ici n'est également pas vectorisée.

La Figure 5.18 montre les accélérations provenant de la vectorisation pour chaque routine. Pour les routines *memory-bound* (*addition de tableaux*, *somme horizontale*, *horner par-coefficient*), le gain obtenu par vectorisation est maximal de manière ponctuelle. Ce point correspond à la quantité de calcul maximale pour laquelle les données logent dans le cache L1d. En ce point, la couverture du coût constant est optimale tout en conservant des accès mémoire dans L1d.

Pour la routine *horner par-coefficient*, on observe néanmoins un autre pic au niveau des quantités de calcul excédant le cache L3. Ce pic est dû à une sensibilité plus forte de la version scalaire au franchissement de ce seuil. Il est possible d'améliorer la localité mémoire de cette routine pour bénéficier de la vectorisation sur un intervalle plus large en utilisant une implémentation du type *cache-oblivious* [32] améliorant la localité mémoire.

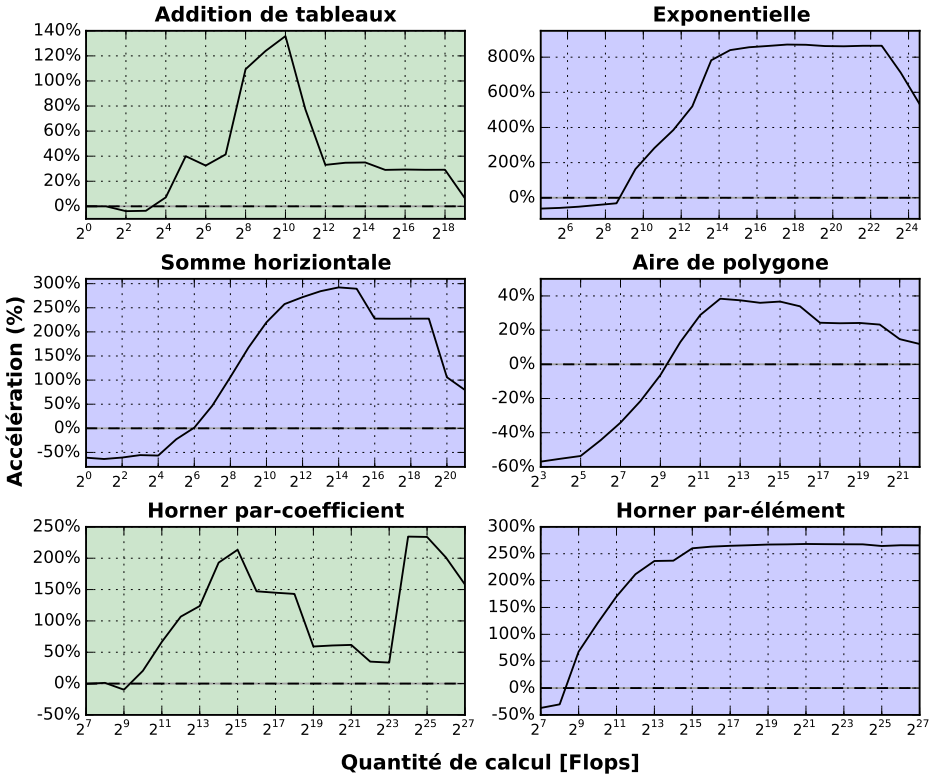


FIG. 5.18: Accélération (en %) provenant de la vectorisation. Les routines sous fond vert sont celles vectorisées par le JIT. Dans ce cas l'accélération est celle de la version *Java+AVX* contre la version *Java*. Les routines sous fond bleu sont celles non vectorisées par le JIT. L'accélération est dans ce cas celle de la version *JNI+AVX* contre la version *Java*

Le principe est qu'au lieu d'itérer sur tous les indices i, à savoir de 0 à $n-1$, pour chaque coefficient c_j (cf. Table 5.6, *parcours par coefficient*), on parcourt, pour chaque coefficient c_j, des blocs de taille réduite logeant dans le cache L1d. Le taux de défauts de cache est ainsi largement diminué comme la chaîne de dépendances est calculée sur un bloc logeant dans L1d. Cette implémentation nécessite un troisième niveau d'itération itérant sur les blocs ce qui ajoute un coût constant supplémentaire. La Figure 5.19 montre le profil de performance de l'implémentation Java vectorisée par bloc de 2KB, comparativement à la version sans bloc. On constate que ses performances sont persistantes au delà du cache L1d contrairement à la version classique. Néanmoins le pic de performance est plus important dans la version classique car le coût des boucles dans la version par bloc est plus important.

Concernant la routine *addition de tableaux*, pour laquelle le JIT génère un code vectorisé d'aussi bonne qualité que du code natif, le seuil garantissant un gain de vectorisation significatif est plus petit dans la version Java qui possède un coût constant bien inférieur (cf. Figure 5.11, *Java+AVX* VS. *JNI+AVX*).
Pour les routines *CPU-bound* (*exponentielle*, *aire de polygone*, *horner par-élément*) le gain de vectorisation est persistant sur tout une plage de quantités de calcul. Ce gain est,

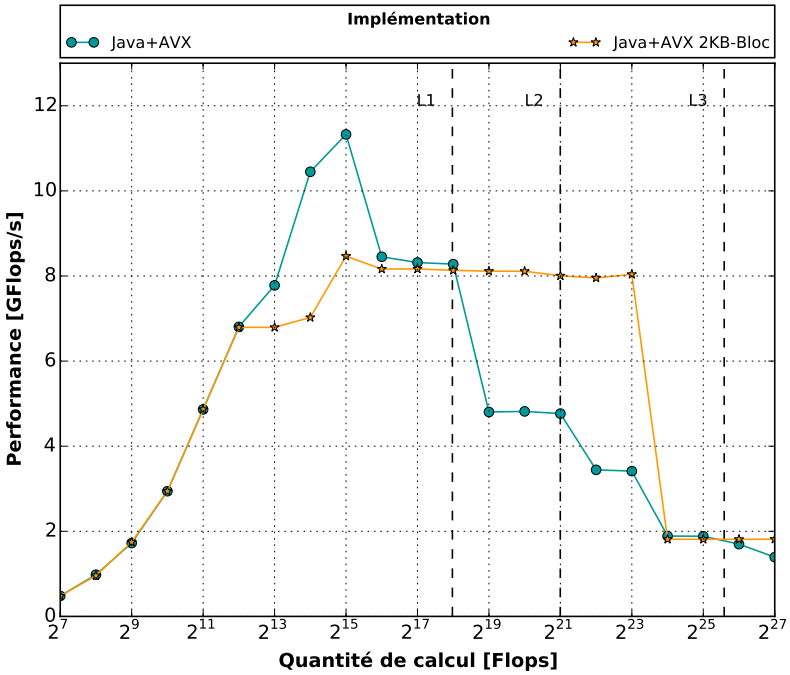


FIG. 5.19: Comparaison des profils de performance de l'implémentation classique et de l'implémentation par bloc de la routine *horner par-coefficient*

potentiellement, d'autant plus fort que la routine est *CPU-bound*. Ainsi pour la routine *exponentielle* qui a la plus forte intensité arithmétique (0.5 Flop/Bytes) on observe le gain le plus élevé (520% en moyenne). Concernant la routine *aire de polygone* le gain est relativement faible ($< 40\%$). Ceci s'explique par le fait que, contrairement aux autres routines *CPU-bound*, les données ne sont pas empaquetées en mémoire (cf. Section 5.5.1.2), ce qui nécessite des opérations d'empaquetage additionnelles qui limitent le gain de la vectorisation (cf. Figure 5.10(b)). Par ailleurs son intensité arithmétique de 0,375 Flop/Byte est plus faible que pour les autres routines CPU-bound (0.5 pour exponentielle, et environ 0.8 pour *horner par-élément* dans sa version vectorisée) ce qui lui confère un comportement proche des routines *memory-bound* avec une sensibilité des performances commençant à la frontière L2.

Parallélisme d'instruction Le parallélisme d'instruction est mis en œuvre pour les micro-routines exposant un idiome de réduction (cf. Section 5.5.1) à savoir les routines *somme horizontale*, *aire de polygones* et *Horner-par-élément*. Les routines *vectorielles* (cf. Section 5.5.1.1) n'ont par définition pas de dépendances réelles entre les itérations qui peuvent donc être exécutées dans le désordre (cf. Section 2.1.1.2 Chap. 2).

Pour la routine *somme horizontale* (cf. Figure 5.13), le parallélisme d'instruction seul permet d'obtenir une accélération moyenne de 110% dans le cas des implémentations Java (*Java+OOO* vs. *Java*) et de 20% dans le cas des implémentations JNI (*JNI+AVX+OOO*

vs. *JNI+AVX*) sur tout l'intervalle des quantités de calcul. L'accélération est moins significative pour les implémentations JNI dont les performances s'écrasent pour les faibles quantités de calcul à cause du coût constant mais également pour les grandes quantités de calcul du fait du caractère *memory-bound* de la routine (cf. Section 5.5.2.1).
Sur l'architecture considérée, le nombre minimal de chaîne de dépendances requis pour bénéficier d'une exécution d'instructions optimale est de 3. En effet, l'instruction la plus critique de la chaîne de dépendances est l'instruction `MOVQ` qui charge un mot de 64-bits depuis la mémoire vers un registre flottant `XMM` (routine *memory-bound*). En supposant que les données logent dans le cache L1d, cette instruction à une latence de 3 cycles pour un débit réciproque de 1 cycle. Ainsi, pour saturer le pipeline de son unité d'exécution et bénéficier de son débit réciproque, il doit y avoir au minimum $3/1 = 3$ lectures mémoire indépendantes donc 3 chaînes indépendantes. La Table 5.10 montre les performances en fonction du nombre de chaînes de dépendances pour une quantité de calcul fixée, on observe bien qu'au delà de 3 chaînes le gain de performance se stabilise.

Nombre de chaîne de dépendances	**Performance [GFlops/s]**
1	1,2
2	2,3 (+90%)
3	3,3 (+40%)
4	3,3 (+0%)
6	3,3 (+0%)

TAB. 5.10: Performance de la routine *somme horizontale* en fonction du nombre de chaînes de dépendances pour une quantité de calcul de 1 KFlops. La valeur entre parenthèse est l'accélération obtenue par rapport au nombre de chaînes précèdent. On observe une stabilisation de l'accélération au delà de 3 qui correspond au nombre minimal de chaînes de dépendances requis pour bénéficier de manière optimale de l'exécution out-of-order

Pour la routine *Aire de polygones* (cf. Figure 5.14), le parallélisme d'instruction ne permet pas d'obtenir un gain de performance pour les implémentations Java. La Table 5.11 montre le corps de boucle de deux implémentations Java pour cette routine, l'une avec 1 chaîne de dépendances et l'autre avec 2. La seconde est, en théorie, censée exploiter d'avantage l'exécution out-of-order comme le nombre d'instructions indépendantes dans la fenêtre d'exécution est plus élevé. Néanmoins, les performances pour les deux implémentations demeurent similaires. La raison étant que les ports d'exécution mis en jeu sont déjà saturés avec seulement 1 chaîne de dépendance. En effet après le chargement des variable `xn` et `yn` (cf. Table 5.11), 4 opérations indépendantes peuvent être exécutées. Celles-ci sont le calcul de `dx` et `dy` avec respectivement les instructions `ADDSD` et `SUBSD` puis le chargement de `xp` et `yp` avec l'instruction `MOVAPD`. Ces 4 opérations indépendantes utilisant le même port d'exécution suffisent à saturer le débit.
Concernant les implémentations JNI de la routine *Aire de polygones*, on observe un gain de performance de 15% en moyenne sur tout le domaine des quantités de calcul (*JNI+AVX+OOO*

vs. *JNI+AVX* Figure 5.14). Le gain moyen est limité par les faibles quantités de calcul et devient quasi-nul à partir d'un certain seuil, ainsi que pour les grandes quantités de calcul dont la taille des données excède le cache L3. Entre ces deux seuils le gain moyen est de 25%. Ce gain de performance ne provient pas de l'exécution out-of-order qui n'est pas exploitable par l'architecture comme vu précédemment mais de l'exploitation de ce parallélisme par des instructions vectorielles.

Implémentation classique avec 1 chaîne de dépendances

```
for(i=2; i<n2; i+=2){
    xn = points[i+0];
    yn = points[i+1];
    dx = xn + xp;
    dy = yn - yp;
    area += dx * dy;
    xp = xn;
    yp = yn;
}
```

Implémentation avec 2 chaînes de dépendances

```
for (i=4; i<n4; i+=4){
    xn0 = xp1;
    yn0 = yp1;
    xn1 = points[i+0];
    yn1 = points[i+1];
    dx0 = xn0 + xp0;
    dy0 = yn0 - yp0;
    dx1 = xn1 + xp1;
    dy1 = yn1 - yp1;
    area0 += dx0 * dy0;
    area1 += dx1 * dy1;
    xp0 = xn1;
    yp0 = yn1;
    xp1 = points[i+2];
    yp1 = points[i+3];
}
```

TAB. 5.11: Corps de boucle Java de la routine *aire de polygone* avec une et deux chaînes de dépendances. Les performances des deux implémentations sont similaires. L'implémentation avec deux chaînes est un exemple où l'expression du parallélisme d'instruction n'est pas exploitable par l'architecture : une chaîne de dépendances suffit à saturer les ports d'exécution. L'épilogue et l'initialisation de la boucle ne figurent pas dans le code

Enfin pour la routine *Horner par-élément* on observe un gain de performance de 310% en moyenne sur tout le domaine (cf. Figure 5.15) pour les implémentations JNI (*JNI+AVX+OOO* vs. *JNI+AVX*) et de 290% pour les implémentations Java (*Java* vs. *Java+OOO*). La routine étant fortement *CPU-bound* le gain reste significatif également pour les grandes quantités de calcul.
Néanmoins ce gain de performance ne provient pas de l'exécution out-of-order de la chaîne de dépendance. En effet, le calcul du résultat de la chaîne de dépendances $y \longleftarrow y \times x + c_j, j = d-1, ..., 0$ (cf. Table 5.6) ne peut pas s'exprimer efficacement en fonction du résultat de plusieurs sous-chaînes de dépendances indépendantes. Autrement dit, la chaîne n'est pas parallélisable efficacement. Comme montré Table 5.12, le gain observé provient du déroulage de boucle qui permet d'une part l'exécution out-of-order de plusieurs itérations successives indépendantes figurant, après déroulage, dans la même fenêtre d'exécution et d'autre part, de l'augmentation de l'intensité arithmétique provenant de la factorisation des accès mémoire. En effet, les accès aux coefficients c_j du polynôme en mémoire peuvent

être effectués une seule fois pour chaque chaîne de dépendance. La quantité d'accès mémoire effectuée par la routine est de $N(16 + \frac{8D}{F})$ Bytes où N est le nombre d'éléments, D le degré du polynôme et F le facteur de déroulage de boucle.

La Table 5.13 montre les performances de la routine *horner par-élément* pour différents

Implémentation sans déroulage de boucle

```
for(i=0; i<n; ++i){
    yi = cmax;
    xi = x[i];
    for(j=dmax; j>-1; --j){
        yi *= xi;
        yi += c[j];
    }
    y[i] = yi;
}
```

Implémentation avec déroulage de boucle de facteur 2

```
for(i=0; i<n2; i+=2){
    yi0 = cmax;
    yi1 = cmax;
    xi0 = x[i+0];
    xi1 = x[i+1];
    for(j=dmax; j>-1; --j){
        yi0 *= xi0;
        yi1 *= xi1;
        cj = c[j];
        yi0 += cj;
        yi1 += cj;
    }
    y[i+0] = yi0;
    y[i+1] = yi1;
}
```

TAB. 5.12: Corps de boucle Java de la routine *horner par-élément* avec et sans déroulage. Le déroulage de boucle permet l'exécution dans le désordre de deux itérations indépendantes successives qui figurent, après déroulage, dans la même fenêtre d'exécution. Le déroulage de boucle permet de surcroît de factoriser les accès aux coefficients polynomiaux (c_j) en mémoire pour chaque chaîne de dépendances. L'épilogue et l'initialisation de la boucle ne figurent pas dans le code

facteurs de déroulage de boucle. Pour bénéficier d'une exécution out-of-order optimale il faut au moins dérouler d'un facteur 5. En effet, l'opération la plus critique de la chaîne de dépendances étant la multiplication `MULSD` qui a une latence de 5 cycle et un débit réciproque de 1 cycle, la saturation de son débit nécessite $5/1 = 5$ chaînes de dépendances. Au delà de 5 l'accélération provient de la factorisation des accès mémoire. Le nombre de chaînes permettant d'obtenir les performances optimales est de 7. Au delà de 7, le débit de chargement du ROB est réduit du fait de la saturation de l'instruction-queue causée par l'augmentation de la taille du corps de boucle (cf. Section 2.1.1 Chap. 2) .

Alignement des données Les problèmes d'alignement touchent à la fois les implémentations Java et les implémentations JNI (hors mémoire native pour lesquelles l'alignement peut être demandé explicitement à l'allocation). Comme vu Section 5.4.2.1, l'objectif de l'alignement pour les routines considérées est de garantir que les mots de 32 bytes, transférés vers ou depuis les registres `YMM` dans la boucle vectorisée, soient alignés sur 16 bytes. L'alignement des tableaux sur 16 bytes est ainsi une condition suffisante pour garantir l'alignement des transferts.

Facteur de déroulage de boucle	Performance [GFlops/s]
1	1,4
2	1,8 (+28%)
3	3,0 (+66%)
4	3,8 (+26%)
5	4,6 (+21%)
6	5,5 (+19%)
7	6,3 (+14%)
8	6,0 (-5%)

TAB. 5.13: Performance de la routine *horner par-élément* en fonction du facteur de déroulage de boucle pour une quantité de calcul de 128 KFlops. La valeur entre parenthèse est l'accélération obtenue par rapport au facteur de déroulage précédent. Le facteur de déroulage minimal pour obtenir une exécution out-of-order optimale est de 5. Au delà de 5 l'accélération est due à l'augmentation de l'intensité arithmétique. Au delà de 7 les performances diminuent à cause de l'augmentation de la taille du code

Le problème de l'alignement se pose pour des routines faisant des transferts mémoire depuis ou vers au moins deux tableaux. Pour les routines manipulant un seul tableau, à savoir les réductions *somme horizontale* et *aire de polygone*, l'alignement peut toujours être résolu via du *peeling* de boucle (cf. Section 5.4.2.1), à savoir que l'on démarre la boucle de vectorisation au premier décalage aligné.
Pour les autres routines, faisant des transferts vers deux tableaux de double, deux cas problématiques se posent :

1. les deux tableaux sont en phase et au moins un est non-aligné ;
2. les deux tableaux sont déphasés et exactement un est non-aligné.

Les tableaux étant toujours alignés sur 8 bytes, on ne peut pas avoir deux tableaux qui soient déphasés et non alignés car cela impliquerait un alignement inférieur à 8 (à savoir 4, 2 ou 1) pour l'un des tableaux.
Le *peeling* permet de résoudre l'alignement dans le premier cas mais pas dans le second. Le second cas correspond précisément à la configuration "8-16" à savoir un des tableaux est aligné sur 8 bytes et l'autre sur 16 bytes. Dans ce cas les transferts pour l'un des tableaux seront non-alignés.
L'enjeu est ainsi de pouvoir garantir des accès alignés même en cas de configuration "8-16". Pour les routines *horner par-élément* et *exponentielle*, le problème se pose uniquement concernant les implémentations JNI, les implémentations Java étant non vectorisées. Pour les routines *horner par-coefficient* et *addition de tableaux* le problème se pose pour les deux implémentations, Java et JNI.
Concernant les implémentations JNI, l'alignement peut être résolu systématiquement en alignant "artificiellement" les tableaux primitifs. La solution retenue est de contenir le tableau de double dans un objet contenant 2 champs. Le premier champs étant le tableau de double et le second un entier correspondant à l'index du premier élément du tableau aligné

sur la valeur voulue. Cet index est ensuite communiqué aux méthodes JNI, possédant un argument prévu à cet effet, et qui sert de base lors du parcours du tableau. Pour que la solution soit utilisable en production, il faut néanmoins que l'index soit invariant par rapport au ramasse-miette. Le seul moyen de garantir cette condition est de demander l'alignement des objets sur l'alignement désiré via l'option `-XX:ObjectAlignmentInBytes=<value>`. L'index est alors constant par instance d'application et dépend uniquement de l'activation ou non du mode pointeur compressé comme montré Figure 5.9.
Pour les implémentations Java, l'ajout d'un argument pour la base des tableaux empêche l'auto-vectorisation du fait de l'aliasing potentiel (cf. Section 5.4.2.1). Par conséquent, pour les méthodes Java il n'y a pas de solution pour contourner le défaut d'alignement lors d'une configuration "8-16". La Table 5.14 montre les baisses de performance dues au non-alignement d'un des tableaux lors d'une configuration "8-16" pour la quantité de calcul correspondant au pic de performance. La solution alternative valable pour les tests mais non réalisable dans du code de production, est là encore de forcer l'alignement des objets sur l'alignement voulu. L'index du premier élément aligné est alors une constante de compilation utilisable en dur directement depuis les méthodes. Cette alternative peut servir pour des tests mais est néanmoins totalement obsolète comme elle rend la validité fonctionnelle d'une méthode dépendante des spécifications et de la configuration de l'environnement d'exécution.

Addition de tableaux	Tableau en lecture : **-36%** Tableau en lecture/écriture : **-36%**
Horner par-coefficient	Tableau en lecture : **-10%** Tableau en lecture/écriture : **-20%**

TAB. 5.14: Baisses de performance (en %) des versions Java dues au non-alignement d'un des tableaux lors d'une configuration "8-16". La baisse de performance est calculée pour la quantité de calcul correspondant au pic de performance par rapport à une exécution sans défauts d'alignement. La Section 5.4.2.2 montre la technique utilisée pour contourner les défauts d'alignement

Mémoire native Comme observé dans les différents profils de performance le coût constant devient significatif en deçà d'un certain seuil des quantités de calcul. Les performances chutant à mesure que la quantité de calcul diminue (cf. Section 5.3.2). Cette partie du profil de performance peut être qualifiée de *constant-bound* (en référence à *memory-bound* et *CPU-bound*) comme la partie constante y est le principal goulot d'étranglement. Dans la partie *call-bound*, les versions Java des routines sont toujours plus performantes (cf. Section 5.5.2.1) du fait d'un coût constant bien moins élevé. L'utilisation de mémoire native apparait ainsi comme une optimisation constante permettant, en contournant le coût d'intégration de JNI, de réduire considérablement l'écart de performance avec les implémentations Java sur la partie *constant-bound*, tout en conservant les optimisations significatives sur les grandes quantités de calcul permises par du code natif.

En admettant que son utilisation au sein des applications demeure très contraignante, comme elle bouleverse profondément le paradigme de programmation, elle permet néanmoins d'obtenir une mesure du coût d'intégration via JNI en comparant les performances des implémentations avec et sans mémoire native.
Le Figure 5.20 montre pour chaque micro-routine les accélérations provenant de l'utilisation de mémoire native. Comme escompté, l'accélération est d'autant plus grande que la quantité de calcul est petite. La routine pour laquelle le coût d'intégration via JNI est le plus important étant l'*addition de tableaux* avec une accélération d'environ 250% pour la quantité de calcul minimale. Pour les routines *horner par-élément* et *exponentielle* on observe un pic très marqué au niveau de la pente décroissante, ce pic marque le franchissement du seuil d'exécution de la boucle optimisée (i.e. vectorisée et déroulée). En effet, le coût relatif des instructions flottantes est alors divisé et le coût d'intégration devient alors plus important dans la contribution du coût global, ce qui implique une accélération plus importante lorsque le coût d'intégration est supprimé.

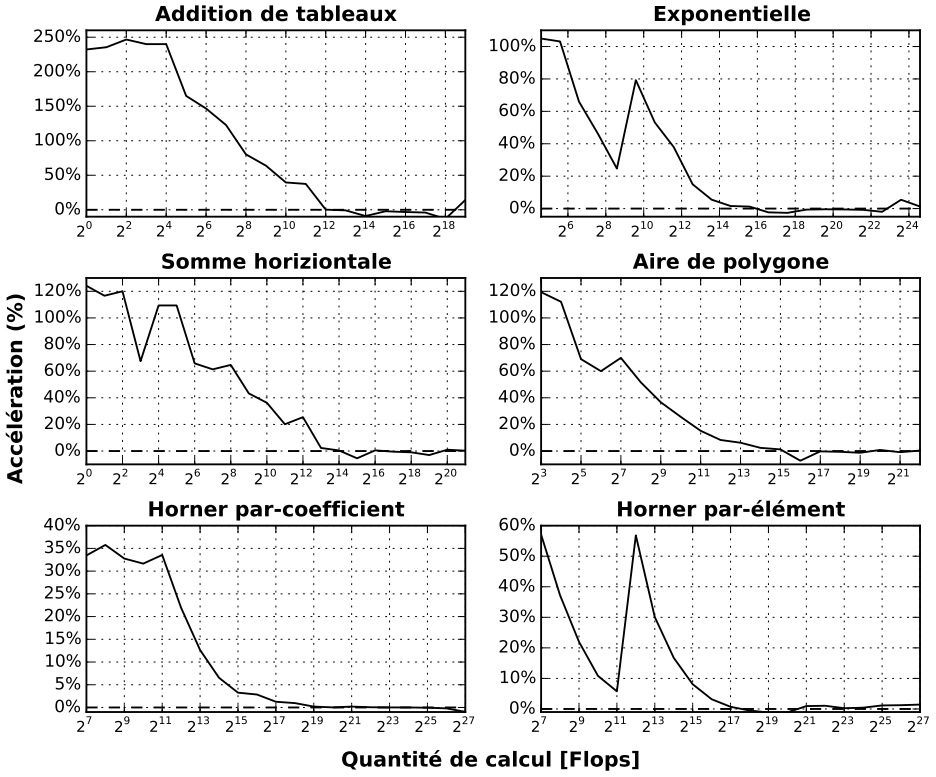


FIG. 5.20: Accélération (en %) obtenue par l'utilisation de mémoire native pour les différentes micro-routines

5.6 Conclusion

Ce chapitre a montré que JNI représentait une alternative crédible pour pallier aux défauts du langage Java et du JIT en matière de performance. Il propose une analyse des

performances de micro-routines vectorisables de complexité linéaire en identifiant les limitations du JIT et en considérant le coût d'invocation via JNI.
L'analyse est basée sur la mesure et le tracé des profils de performance des micro-routines qui permet de sélectionner la meilleure implémentation sur un intervalle de quantité de calcul donné. L'usage des profils de performances peut être envisagé de manière dynamique lorsque l'intervalle des quantités de calcul sollicité n'est pas connu statiquement. Il peut également permettre de calibrer la taille des paquets afin de maximiser les performances de la micro-routine.
Les investigations ont montrées que les limitations du JIT impactaient des idiomes de base fréquemment utilisés dans les applications scientifiques. Dans de tels cas l'utilisation de JNI permet d'obtenir des gains significatifs. Néanmoins JNI s'avère toujours limité par son coût et son usage ne peut être généralisé pour pallier à toutes les limitations du langage (cf Chap. 6)
Les allures des profils de performances ont été analysées en fonction des caractéristiques propres à chaque routine afin de fournir des éléments d'analyse applicables à d'autres routines. L'usage de JNI s'inscrit dans une approche JIT-Friendly (cf. Section 7.1.1 Chap. 7) dans laquelle l'implémentation est guidée par le comportement du JIT afin de fournir la meilleure implémentation. L'aspect JIT-Friendly a principalement consisté à analyser si les idiomes considérés étaient ou non vectorisés par le JIT.

L'étude effectuée dans ce chapitre a fait l'objet d'une publication dans un workshop international [65]. Considérant les difficultés de l'approche JIT-Friendly, qui nécessite une connaissance avancée du comportement du JIT, l'étude s'est limitée à des idiomes de base représentés dans un échantillon de micro-routines réduit. L'évolution des capacités du JIT pousse les futurs travaux à s'intéresser à des idiomes plus complexes et plus représentatifs des algorithmes utilisés dans les applications scientifiques.

Chapitre 6

Extension du compilateur dynamique pour des méthodes applicatives

Dans le chapitre 5, on a considéré des micro-routines de complexités linéaires par rapport à la taille des données possédant ainsi une quantité de calcul variable. Pour ces routines l'efficacité des méthodes natives repose sur la valeur de la quantité de calcul qui doit couvrir le coût d'intégration via JNI. Ce chapitre se focalise sur des micro-routines de faibles complexités relativement au coût d'intégration via JNI, pour lesquelles, son utilisation n'est d'emblée pas efficace.
La piste explorée pour de telles routines est leur implémentation sous forme d'intrinsics du compilateur dynamique C2 de la JVM HotSpot, la motivation étant de pouvoir bénéficier d'optimisation native sans le surcoût occasionné par JNI.

Un ***intrinsic***[1] d'un compilateur (aussi appelé *built-in* dans GCC) est une fonction reconnue par le compilateur dont le traitement à la compilation est prédéfini et spécifique et diffère donc du traitement standard appliqué aux autres fonctions. Les intrinsics permettent de générer un code plus adapté que celui que produirait le compilateur avec un traitement standard. Ils peuvent ainsi être vu comme des macro-instructions (à l'instar du bytecode) permettant de relier une fonction à du code spécifique.
Le mécanisme des intrinsics est dépendant du compilateur. Concernant le compilateur C2 de HotSpot, l'ensemble des intrinsics peut être étendu au niveau de la JVM, notamment à des méthodes applicatives, on parle ainsi d'*extension* comme il s'agit de modifications additives n'ayant pas d'effet sur la compilation des méthodes tierces.
Par abus de langage on dira qu'une méthode intégrée comme un intrinsic est une méthode ***intrinsifiée***. On parlera d'***intrinsification*** pour désigner le fait d'intrinsifier une méthode.
La Figure 6.1 montre le principe de fonctionnement des instrinsics dans la JVM HotSpot. Ce dernier est détaillé plus loin dans le chapitre.

[1] Le terme intrinsic est un anglicisme, le terme français équivalent est *intrinsèque* ou *fonction intrinsèque*

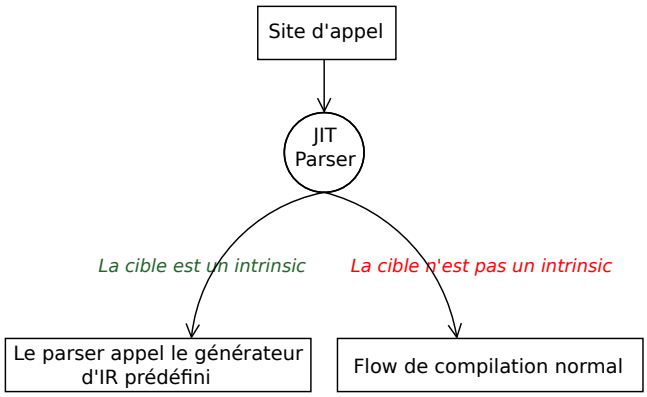


FIG. 6.1: Principe de fonctionnement des intrinsics dans la JVM HotSpot. Lorsque le parser du compilateur dynamique rencontre un site d'appel il vérifie si la cible de l'appel est un intrinsic ou non. Si la cible est un intrinsic, il appel un générateur prédéfini dans la JVM qui inline la représentation intermédiaire correspondante, sinon le flow de compilation normal est suivi

L'intrinsification de méthodes application-spécifique[2](i.e. utilisées à priori exclusivement dans l'application) est un pas vers la spécialisation du runtime à l'application ce qui, pour les concepteurs de runtime, n'est pas envisageable comme ce dernier est conçu pour exécuter efficacement n'importe quelle application. C'est pourquoi ce genre d'implémentation doit être effectué coté utilisateur du runtime (*user-end*). Dans le cas du logiciel Inscale étudié, l'application Java et le runtime sont packagés et livrés ensemble, le runtime n'est dans ce cas prévu que pour une seule application et peut donc être adapté.

La Section 6.1 décrit la méthodologie mise en œuvre pour l'ajout d'instrinsic. La Section 6.2 présente les méthodes intrinsifiées ainsi que les optimisations effectuées. Enfin la Section 6.3 présente et discute les résultats obtenus.

6.1 Méthodologie d'ajout des intrinsics

La méthodologie mise au point et utilisée pour l'ajout d'intrinsics est illustrée Figure 6.2. L'étape 1 "*Sélection de méthodes admissibles*" est détaillée Section 6.1.1, les étapes 2 "*Développement de versions optimisées*" et 3 "*Micro-benchmark des différentes versions*" Section 6.1.2 et enfin les étapes 4 "*Implémentation de l'intrinsic*" et 5 (*Test des performances de l'intrinsic*) Section 6.1.3.

[2]On distingue les méthodes "applicatives" qui désignent l'ensemble des méthodes utilisées dans l'application et les méthodes "application-spécifiques" qui est un attribut empirique pour désigner des méthodes propres à l'application par opposition aux méthodes dites génériques communes à toutes les applications

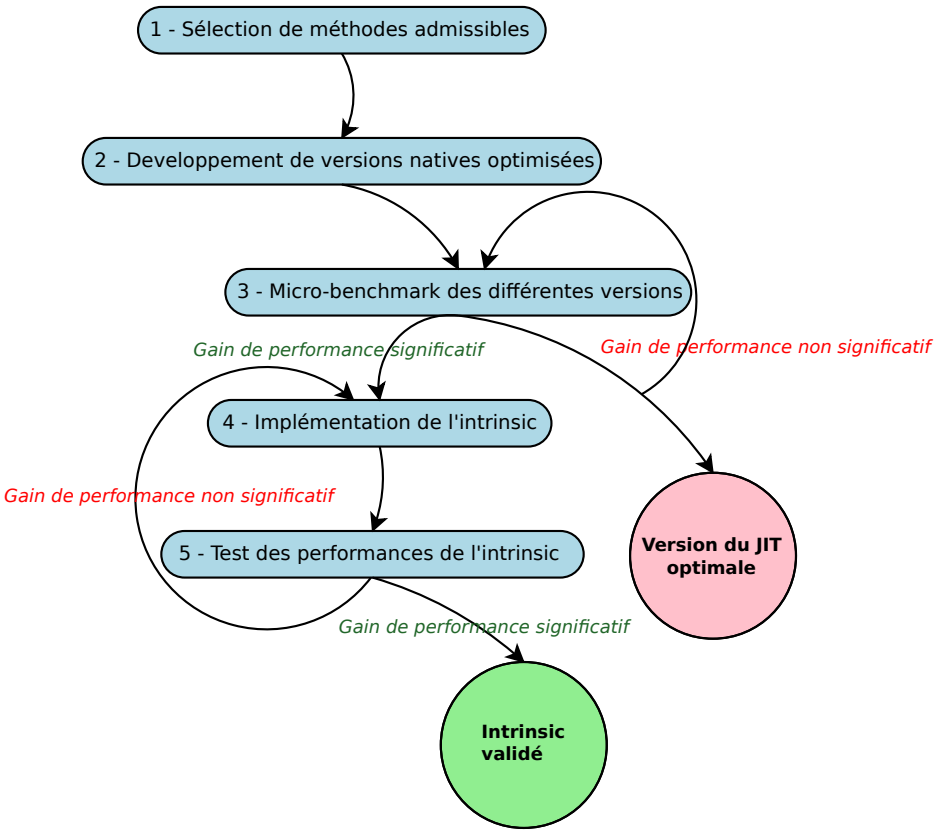


FIG. 6.2: Méthodologie d'ajout des intrinsics mise en œuvre

6.1.1 Sélection des méthodes admissibles

La problématique de sélection de méthodes admissibles pour être intrinsifiées est initialement présente pour les développeurs de la JVM HotSpot. La différence fondamentale avec un intrinsic application-spécifique étant que ce dernier doit être profitable à toutes les applications et concerne donc en premier lieu des méthodes génériques au sein des librairies standards de l'environnement Java. De nombreux intrinsics sont implémentés au sein de HotSpot, on en décompte environ 200 dans la version utilisée dans Java 8. C'est par exemple le cas de nombreuses fonctions usuelles définies dans la classe `java.lang.Math` ou encore de la majorité des méthodes de la classe `sun.misc.Unsafe`. Dans Java 9, l'annotation `@HotSpotIntrinsicCandidate` a été ajoutée pour identifier un candidat pour être intrinsifié dans HotSpot et éviter les divergences fonctionnelles qui pouvait survenir lorsque la version Java était modifiée.

6.1.1.1 Caractérisation des méthodes admissibles

Le premier pré-requis pour qu'une méthode applicative soit intrinsifiée est qu'elle soit un point-chaud applicatif en prenant en compte sa durée d'exécution propre dans la mesure de sa contribution à la durée d'exécution globale (i.e. sa durée d'exécution globale moins

la durée d'exécution des éventuelles sous-fonctions). On peut observer deux tendances d'exécution opposées pour une méthode constituant un point chaud :

1. Tendance localisée : la méthode est exécutée intensivement dans un contexte unique ;
2. Tendance diffuse : la méthode est exécutée dans une multitude de contextes.

Les points chauds respectant la seconde tendance sont généralement plus difficilement identifiables par du profiling comme leur contribution à la durée d'exécution globale est diffuse, néanmoins le gain potentiel a un impact plus large comme il ne se réduit pas à un contexte unique.
En plus d'être un point-chaud la méthode doit présenter un potentiel d'optimisation. Par exemple lorsqu'elle peut bénéficier d'instructions machine adaptées non générées par le JIT. Différentes caractéristiques pouvant rendre l'optimisation difficile voire impossible sont à prendre en considération :

- La méthode est trop large ou trop complexe ;
- La méthode a un coût incompressible ;
- Le JIT génère un code optimal ou quasi-optimal pour cette méthode.

La méthode doit idéalement être petite (du point de vue du nombre d'instruction) et ne pas appeler de sous-méthode (méthode dite *leaf* ou "feuille"). Elle doit être fonctionnellement simple (pas de d'exception levée ni, idéalement, d'accès à des champs d'instance ou statiques). Les méthodes statiques sans entrées/sorties (que ce soit des appels de fonction ou des accès en *heap*), ayant pour argument et valeur de retour des types primitifs sont ainsi des candidats idéals. La méthode doit pouvoir être optimisée via des instructions spécifiques non générées par le JIT et ne doit pas être un point-chaud du fait d'opérations ne pouvant être optimisées, par exemple dans le cas d'une méthode *memory-bound*.

6.1.1.2 Granularité de la méthode

On peut établir une corrélation empirique entre granularité et généricité. Plus l'on considère des méthodes de faible granularité qui tendent vers des opérations élémentaires, plus elles seront utilisables dans une gamme variée de contexte (tendance diffuse vue Section 6.1.1.1) et plus elles seront simples à implémenter. À contrario plus l'on considère des méthodes de forte granularité, plus leur fonctionnalité est complexe et difficile à implémenter et plus leur traitement est spécifique et localisé. Cette considération conduit à privilégier des méthodes de faible granularité.
Réduire la granularité au maximum conduit à intrinsifier des méthodes représentant des instructions machine. On se heurte alors à différentes contraintes portant sur la version Java de l'intrinsic :

- La contrainte de portabilité. Cette dernière garantit que la méthode intrinsifiée est définie sur toutes les architectures ciblées sans que son utilisation n'implique de régression majeure de performance sur l'une d'elles. Dans le cas de l'application considérée, la portabilité n'étant pas une contrainte majeure, cette dernière peut

être évacuée ;

- La contrainte de prototypage. Cette dernière garantit que la méthode peut être prototypée dans le langage Java. C'est la contrainte la plus forte car, si elle n'est pas résolue, seule une modification du langage permet de la résoudre ;
- La contrainte d'expressivité. Elle se pose uniquement dans le cas d'une méthode Java pure (i.e. non native). Cette dernière garantit que l'effet de l'opération machine peut être exprimé au niveau bytecode[3]. L'utilisation d'une méthode native permet de résoudre cette contrainte en encapsulant directement l'instruction machine.

L'intrinsification d'instructions vectorielles AVX, par exemple, se heurte à la contrainte de prototypage. Cette dernière permettrait de vectoriser n'importe quel idiome non-vectorisé par le JIT pour les architectures supportant cette extension. Si l'on considère l'instruction `VADDPD`, qui additionne deux vecteurs de 4 doubles, on se heurte en Java à la contrainte de prototypage comme le langage ne définit pas de type vectoriel (cf. Table 6.1). Il est alors possible de prototyper un intrinsic Java manipulant des tableaux primitifs au lieu des types vectoriels comme montré Table 6.1. Néanmoins la méthode permettra une vectorisation efficace uniquement si les lectures et écritures supplémentaires introduites peuvent être éliminées (i.e. factorisées au niveau du bloc de base), ce qui peut s'avérer impossible dans le cas de certains idiomes manipulant des vecteurs temporaires.

C/GCC	`__v4df __builtin_ia32_addpd256(__v4df, __v4df)`

Java/C2	`static void addpd256(double[] src, int bsrc,`
	`double[] dst, int bdst){`
	`dst[bdst+0]+=src[bsrc+0]`
	`dst[bdst+1]+=src[bsrc+1]`
	`dst[bdst+2]+=src[bsrc+2]`
	`dst[bdst+3]+=src[bsrc+3]`
	`}`

TAB. 6.1: Fonction C et méthode Java équivalentes à l'instruction vectorielle `VADDPD`. La première ligne montre le prototype de l'intrinsic GCC permettant de générer l'instruction (elle utilise le type primitif vectoriel `__v4df`). La seconde ligne montre le prototype et la définition Java d'un intrinsic du JIT permettant de générer l'instruction. Contrairement au C, Java ne définit pas de type primitif vectoriel

[3]Il est fort probable, mais non vérifié, que le caractère Turing-complet du langage Java rende la contrainte d'expressivité inutile comme il garantit que toute sémantique assembleur peut être exprimé en Java. Dans ce cas la contrainte d'expressivité concerne d'avantage les performances de la version bytecode lorsque la version intrinsifiée n'est pas utilisée, par exemple si la méthode intrinsifiée est l'une des cibles d'un appel polymorphique

6.1.2 Optimisations et micro-benchmark

Lorsqu'une méthode admissible est sélectionnée, la seconde étape consiste à fournir des implémentations natives optimisées de la méthode. Le langage qui fut utilisé pour le développement des versions natives est le C qui, d'une part, permet une programmation niveau assembleur et, d'autre part, expose de nombreux intrinsics permettant d'émettre les instructions machine voulues.
Dans la troisième étape, les performances des versions natives sont comparées à celles de la version générée par le JIT à l'aide d'un micro-benchmark. Cette étape est détaillée Section 6.1.2.2.
Il peut y avoir une boucle de plusieurs itérations entre l'implémentation des versions natives et les tests de performance afin de maximiser le gain de performance en ajustant le code. Si aucun gain n'est observé à l'issue de plusieurs itérations la version générée par le JIT peut être considérée comme optimale et la méthode n'est pas intrinsifiée.

6.1.2.1 Isofonctionnalité entre version native et Java

Un des aspects à prendre en compte lors de l'implémentation des versions natives concerne les spécifications de la méthode. En effet une méthode intrinsifiée possèdera deux implémentations dont les spécifications peuvent diverger :

- La version Java qui est soit une méthode Java pure, interprétée, puis exécutée après compilation ; soit une méthode native passant par JNI ;
- La version native optimisée qui sera générée lors de l'inlining de la méthode.

La version native destinée à être intrinsifiée dans C2 peut omettre certaines vérifications pour améliorer les performances et donc avoir un comportement différent pour des cas d'exécution exceptionnels. Ces vérifications sont spécifiées pour la majorité des bytecodes comme par exemple `aastore`, stockant une référence dans un tableau, qui peut lever les exceptions `NullPointerException` ou `ArrayStoreException` [60]. Les spécifications de la méthode intrinsifiée devront donc préciser que le comportement est indéterminé dans les cas où il y a divergence fonctionnelle (ce qui peut en limiter l'usage).
Le respect de certaines règles d'admissibilité, énoncées dans la section précédente (i.e. méthodes statiques manipulant des types primitifs), permet d'éviter ce type de divergences fonctionnelles. Un moyen d'assurer l'isofonctionnalité est de définir la version Java en tant que méthode native, ainsi le code inliné et interprété/compilé peut être totalement similaire. La méthode intrinsifiée calculant l'arrondi d'un double, décrite Section 6.2.1, montre un exemple de divergence fonctionnelle entre version Java et version native.

6.1.2.2 Micro-benchmark des différentes versions

Micro-benchmark C et retranscription du code généré par le JIT La troisième étape consiste en l'implémentation d'un micro-benchmark. Celui-ci peut être implémenté de deux manières :

- En C : dans lequel sont développées les versions natives optimisées ;
- En Java : dans lequel sont développées les versions Java.

L'écriture du micro-benchmark en Java nécessite l'utilisation de JNI qui introduit un surcoût trop important du fait de la faible granularité des méthodes testées et qui biaise les résultats de performance. Ainsi, l'utilisation du C est privilégiée ce qui nécessite néanmoins la retranscription du code généré par le JIT dans le code C afin de comparer les performances.
Le micro-benchmark C possède le même design que la version Java décrite Chapitre 3. Le dispatche vers les différentes versions se fait via le rappel des fonctions retournant la durée minimale d'exécution parmi une multitude itérations. Les différentes versions testées peuvent être arbitrairement inlinées ou non à la compilation. Néanmoins et contrairement au micro-benchmark Java, la version C permet de s'affranchir des considérations concernant l'inconsistance des mesures provenant des optimisations dynamiques ou de l'état du code au moment des mesures.
Afin de retranscrire le code généré par C2 dans le micro-benchmark C, ce dernier est, dans un premier temps, affiché sur la sortie standard grâce à l'option `-XX:CompileCommand=print,<method>` (cf. Section 3.4 Chap. 3). Il est ensuite retranscrit dans le micro-benchmark grâce à la directive `asm` de GCC permettant d'inliner de l'assembleur dans le code C.
Lors de la retranscription du code deux points sont à considérer, la dépendance du code aux données de profiling et la présence d'opérations additionnelles non fonctionnelles.

Dépendance du code aux données de profiling Le premier point à considérer est la dépendance du code généré aux données de profiling. Cette dépendance survient pour du code compilé contenant des branchements conditionnels ou bien des bytecodes profilés (cf. Table 3.3 Chap. 3). Le code généré par C2 est retranscrit, peut alors être spécifique aux données utilisées en entrée lors de l'exécution. C'est par exemple le cas de la méthode `floorD2I` décrite Section 6.2.1 contenant des branchements conditionnels pour laquelle les différentes versions générées par C2 sont montrées Table 6.5. Dans l'autre sens, le code retranscrit peut être générique et ne pas prendre en compte des spécialisations effectuées dans les cas d'utilisation réels.
Dans de tels cas, les dépendances aux données d'entrées doivent être déterminées et le gain de performance obtenu grâce aux versions natives doit être mesuré en prenant en compte les gains relatifs à chaque version et si possible en prenant en compte les versions les plus susceptibles d'être générées compte tenu du profil d'exécution applicatif.

Opérations additionnelles dans le code généré par C2 Le second point à considérer est la présence d'opérations additionnelles qui, en admettant que la méthode sera inlinée, doivent être supprimées lors de la retranscription dans le micro-benchmark. Il s'agit des opérations de gestion de la pile en entrée et sortie de méthode ainsi qu'un point de

synchronisation pour le ramasse miette ou *safepoint* (cf. le contenu d'une méthode vide Section 4.1.1.1). Conserver ces opérations conduit à mesurer des performances plus faibles que dans un cas d'exécution réel où elles seraient inlinées.

6.1.3 Implémentation des intrinsics

Une fois qu'une version native optimisée obtient des gains de performance significatifs par rapport à la version générée par C2, l'étape 4 consiste en l'implémentation de l'intrinsic afin de générer le code optimisé.
Un micro-benchmark Java permet, dans une cinquième étape, de valider l'implémentation de l'intrinsic. L'objectif pour valider étant de retrouver un gain de performance similaire à celui mesuré via le micro-benchmark C dans l'étape 3 (cf. Section 6.1.2.2). L'ajout d'une option booléenne `-XX:+UseCustomIntrinsic` à la JVM permet d'activer ou désactiver l'intrinsification afin de comparer les performances.
Plusieurs itérations peuvent avoir lieu entre l'implémentation de l'intrinsic et les tests de performance.
Cette section détaille succinctement la procédure d'implémentation des intrinsics qui comprend deux étapes. La première est la définition d'un intrinsic dans C2 qui est détaillée Section 6.1.3.1. La seconde est l'implémentation de l'intrinsic. On distingue deux manières d'implémenter un intrinsic :

1. Générer un appel vers une sous-routine native qui encapsule le code optimisé, Section 6.1.3.2 ;
2. Générer le code binaire à partir d'un nœud spécifique dans la représentation intermédiaire (IR), Section 6.1.3.3.

6.1.3.1 Mécanisme et définition d'un nouvel intrinsic

Le code d'un intrinsic est contenu dans un générateur au sein de la JVM. Lorsque qu'une méthode appelant l'intrinsic est compilée, ce générateur est appelé afin d'inliner le code optimisé (comme montré Figure 6.1). Notons que l'intrinsic peut également avoir une version compilée qui est exécutée lorsque la méthode appelante est interprétée[4]. Lorsque C2 compile une méthode la première phase est le parsing du bytecode qui traduit le bytecode vers l'IR (cf. Section 2.3.2.4). Lorsqu'il rencontre un appel vers une méthode il peut alors effectuer deux traitements :

- Générer un nœud d'appel ou inliner si la méthode ciblée n'est pas d'un intrinsic. L'inlining est alors géré par le parser qui intègre l'IR de la méthode appelée ;
- Inliner la méthode si cette dernière est un intrinsic. L'inlining est dans ce cas délégué à la fonction `LibraryCallKit::inline_<intrinsic>`, spécifique à l'intrinsic, qui rend ensuite la main au parser.

[4]Par exemple dans le cas où l'intrinsic est une des cibles d'un appel polymorphique passant par une table des méthodes, la cible sera la version compilée et non la version inlinée. La version compilée est également celle sollicitée lorsque la méthode appelante est interprétée

La Figure 6.3 montre la pile d'appel jusqu'à la fonction `inline_<intrinsic>`. La fonction `Parse::do_call` est en charge d'identifier si la méthode appelée est un intrinsic afin, si c'est le cas, d'appeler la fonction `inline_<intrinsic>` correspondante. Le premier point consiste ainsi en la définition de l'intrinsic pour que la méthode soit reconnue comme telle. L'implémentation de l'intrinsic correspond ensuite à l'implémentation de la fonction `inline_<intrinsic>` en charge de générer l'IR correspondant au code optimisé.
Tous les intrinsics sont définies dans le fichier source *vmSymbols.hpp*. Les intrinsics sont définies comme des éléments d'une énumération C (`enum`) qui comporte différents champs d'informations. Pour faciliter l'ajout des intrinsics des macros préprocesseur ont été définis dans HotSpot. Notamment la macro `do_intrinsic` permettant de définir l'intrinsic et nécessitant les informations suivantes :

- Le nom, la classe et la signature de la méthode intrinsifiée ;
- Un flag indiquant si la méthode est native et/ou synchronisée et/ou statique. Les différents flags possibles sont montrés Table 6.2 ;
- Un symbole identifiant l'intrinsic.

```
#0 LibraryCallKit::inline_<intrinsic>()
#1 LibraryIntrinsic::generate(JVMState*, Parse*)
#2 Parse::do_call()
#3 Parse::do_one_bytecode()
#4 Parse::do_one_block()
#5 Parse::do_all_blocks()
#6 Parse::Parse(JVMState*, ciMethod*, float, Parse*)
#7 ParseGenerator::generate(JVMState*, Parse*)
#8 Compile::Compile(ciEnv*, C2Compiler*, ciMethod*, int, bool, bool, bool)
#9 C2Compiler::compile_method(ciEnv*, ciMethod*, int)
```

FIG. 6.3: Pile d'appel du déclenchement de la compilation de la méthode appelant l'intrinsic (i.e. la fonction `C2Compiler::compile_method`) au générateur d'intrinsic (i.e. `LibraryCallKit::inline_<intrinsic>`) qui inline sa représentation intermédiaire

	statique	**native**	**synchronisée**
`F_R`	*N*	*?*	*N*
`F_S`	*O*	*?*	*N*
`F_Y`	*N*	*?*	*O*
`F_RN`	*N*	*O*	*N*
`F_SN`	*O*	*O*	*N*
`F_RNY`	*N*	*O*	*O*

TAB. 6.2: Les différents flags utilisés dans HotSpot pour caractériser les méthodes intrinsifiées ("*O*" signifie oui, "*N*", non et "*?*", oui ou non)

Une condition nécessaire pour que la méthode intrinsifée soit inlinée est que sa classe soit chargée par le chargeur de classe primordial (*bootstrap class-loader*). Pour ce faire, il faut que le package contenant la classe figure dans les répertoires spécifiques du JRE comme le

répertoire *.../jre/lib/ext/*.
La fonction `inline_<intrinsic>` est appelée dans la fonction `try_to_inline` qui est le point d'entrée commun à tous les intrinsics et qui effectue le dispatch. L'ajout d'un intrinsic ajoute donc une entrée à cette fonction, le coût de dispatch reste néanmoins négligeable compte tenu du nombre d'intrinsic. L'impact de l'ajout d'un intrinsic sur la durée de compilation est directement lié à la durée d'exécution de la fonction `inline_<intrinsic>` mais permet généralement d'améliorer même la vitesse de compilation.

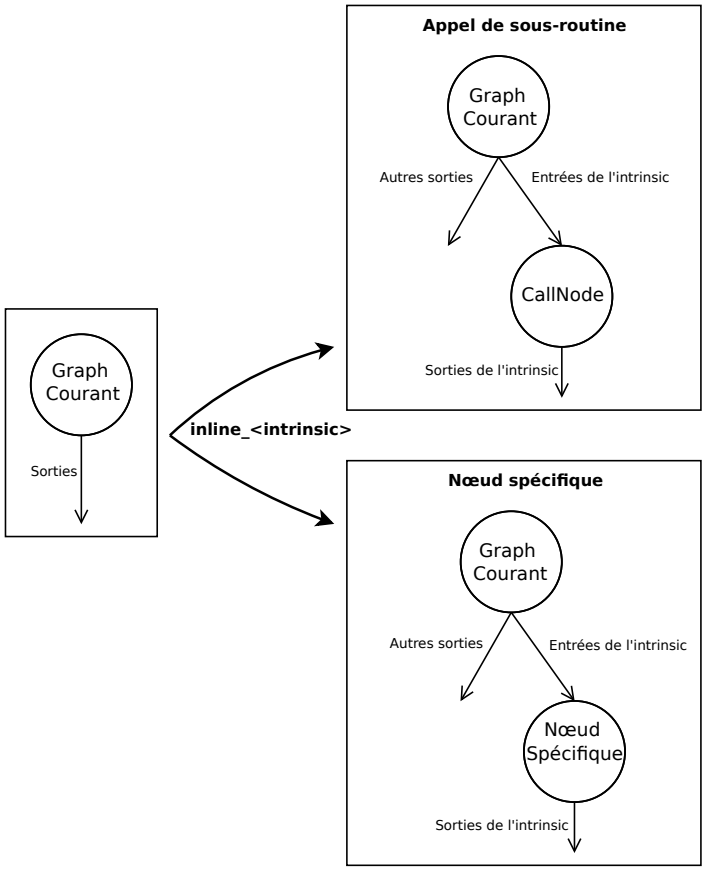


FIG. 6.4: Les deux techniques utilisées pour l'implémentation des intrinsics. Dans la première (avec appel de sous-routine) le nœud généré par `inline_<intrinsic>` est un nœud d'appel vers une routine native compilée statiquement. Dans la seconde, le nœud généré est un nœud spécifique à l'intrinsic, le code binaire associé est alors défini dans le back-end de C2 et généré à la sélection d'instructions

6.1.3.2 Implémentation avec appel de sous-routine

L'implémentation avec appel de sous-routine consiste à encapusler le code natif optimisé dans une fonction C/C++ classique de sémantique équivalente à la méthode Java. Cette approche est similaire à celle passant par JNI, elle permet néanmoins de s'affranchir de son coût, l'intégration étant gérée à la compilation des nœuds d'appel définis dans l'IR.

Différents intrinsics de HotSpot utilisent ce type d'implémentation comme la méthode `System.nanoTime` (décrite Section 3.3.3.2) ou encore la méthode `CRC32.update` pour le calcul de redondances cycliques.
Le rôle de `inline_<intrinsic>` (cf. Section 6.1.3.1) est d'inliner l'IR de la méthode intrinsifiée dans celle de la méthode appelante en cours de compilation. Dans le cas d'un appel de sous-routine, le nœud généré est donc un nœud d'appel vers la sous-routine comme montré Figure 6.4.
La hiérarchie des différents nœuds d'appel définis dans l'IR est montrée Figure 6.6. Les nœuds utilisés pour l'appel de routines natives (i.e. respectant les conventions d'appel C/C++) sont les nœuds héritant de `CallRuntimeNode` à savoir `CallLeafNode` et `CallLeafNoFPNode`. La différence entre ces deux nœuds est expliquée plus loin. Les appels à des méthodes natives via JNI utilisent le nœud `CallStaticJavaNode`[5] qui appel avec les conventions Java une fonction (ou wrapper) qui gère l'appel à la fonction native (cf. Section 5.2.1 Chap. 5). Contrairement aux appels avec les conventions JNI (cf. Section 5.2.1.1), les appels directs ne sont pas des safepoints (cf. Section 2.3.1), ils utilisent des pointeurs directs vers les objets dans la heap et sont par conséquent bloquants pour le GC. Il leur est ainsi interdit d'appeler du code Java susceptible de déclencher un GC, c'est pourquoi ces appels sont qualifiés de *leaf* (feuille). La Table 6.3 compare les débits d'appel mesurés pour des méthodes vides lorsqu'elles sont appelées avec les conventions JNI avec section critique pour l'acquisition des pointeurs d'objets (cf. Section 5.2.2.2), et lorsqu'elles sont appelées en émettant un nœud `CallLeafNode` via le mécanisme des intrinsics. Le coût d'entrée et de sortie des sections critiques augmentent avec le nombre de tableaux primitifs passés en argument comme vu Section 5.2.3.

Différence entre les nœuds `CallLeafNoFPNode` et `CallLeafNode` sur Intel64 Le nœud `CallLeafNoFPNode` peut être utilisé lorsque la sous-routine appelée ne modifie pas l'état d'exécution des instructions flottantes ce qui est assuré si la sous-routine n'utilise pas d'instructions SSE. Sur Intel64 ces différents états d'exécution sont :

- État A : la moitié supérieure des registres YMM est considérée nulle ;
- État B : la moitié supérieure des registres YMM n'est pas considérée nulle excepté pour les instructions AVX-128 ;
- État C : la moitié supérieure des registres YMM n'est pas considérée nulle par les instructions SSE. Ainsi les moitiés supérieures de tous les registres YMM sont sauvegardées puis restaurées automatiquement par le processeur.

Les transitions sont montrées Figure 6.5. Les différents types d'instructions flottantes influant sur l'état d'exécution sont les instructions SSE (ou legacy-SSE i.e. des instructions SSE sans préfixe VEX[6]) opérant sur les registres 128-bits XMM (moitiés inférieures des registres YMM), les instructions AVX-128 (instructions SSE avec préfixe VEX) opérant

[5] Ce dernier est également utilisé pour les appels statiques et les appels virtuels optimisés
[6] Le préfixe VEX pour Vector EXtension est un préfixe d'encodage des instructions x86-64 permettant de définir de nouvelles instructions vectorielles notamment pour l'extension AVX

sur les registres XMM et les instructions AVX-256 opérant sur les registres 256-bits YMM. La différence entre SSE et AVX-128 est que les instructions SSE doivent conserver inchangée la partie supérieure des registres YMM alors que les instructions AVX-128 la charge à 0. Par ailleurs les instructions `VZEROUPPER` (chargeant les moitiés supérieure à zéro) et `VZEROALL` (chargeant les registres à zéro) permettent de revenir à l'état A.
Les transitions problématiques sont les transitions B->C ou C->B/A qui entraînent respectivement une sauvegarde puis restauration des moitiés supérieures des registres YMM. Ces opérations ont un coût élevé d'environ 70 cycles d'horloge par transition sur Sandy Bridge [47]. Comme le JIT ne génère pas d'instructions SSE mais AVX-128, les transitions problématiques sont B->C. Elles surviennent lorsque la méthode appelante utilise des instructions AVX-256 et que la sous-routine appelée utilise des instructions SSE. Dans ce cas il est important d'utiliser le nœud `CallLeafNode` pour l'appel de la sous routine qui génère l'instruction `VZEROUPPER` afin d'éviter le coût élevé de la transition B->C. Si la sous-routine appelée n'utilise pas d'instruction SSE, comme c'est le cas pour les instrinsics implémentés, l'utilisation du nœud `CallLeafNoFPNode` est à privilégier comme il permet de s'affranchir de l'instruction `VZEROUPPER` même si elle ne coûte qu'un cycle d'horloge sur Intel64.

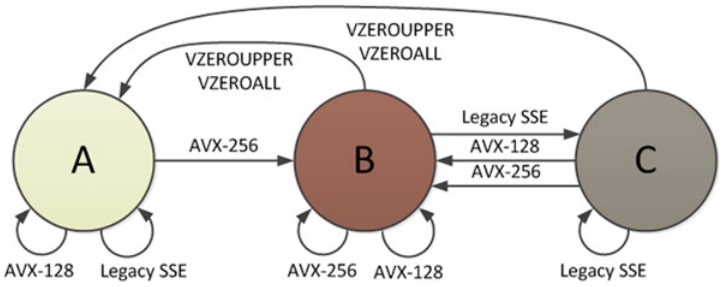


FIG. 6.5: États d'exécution des instructions flottantes sur Intel64. Le code généré par le JIT n'utilisant pas d'instructions SSE mais AVX-128, les transitions problématiques lors de l'appel de sous-routine peuvent être les transitions B->C (cas où la méthode appelante utilise des instructions AVX-256 et la sous-routine appelée des instructions SSE). Pour éviter cette transition le nœud d'appel `CallLeafNode` génère une instruction `VZEROUPPER` pour passer dans l'état A avant appel. *Crédit image* : [89]

Avantage et inconvénient de cette implémentation L'avantage majeur de cette technique d'implémentation n'est autre que sa simplicité. La sous-routine peut être implémentée en C/C++, assembleur ou encore provenir d'une librairie statique ou dynamique. Les nœuds d'appel invoquant les routines natives sont déjà définis dans l'IR. Par ailleurs cette méthode permet de réduire la durée de compilation dans la mesure où un seul nœud dans la représentation intermédiaire intervient.
Le principal inconvénient est que la routine ne peut pas bénéficier d'optimisations de contexte provenant de l'allocation de registre, de la propagation de constante ou encore

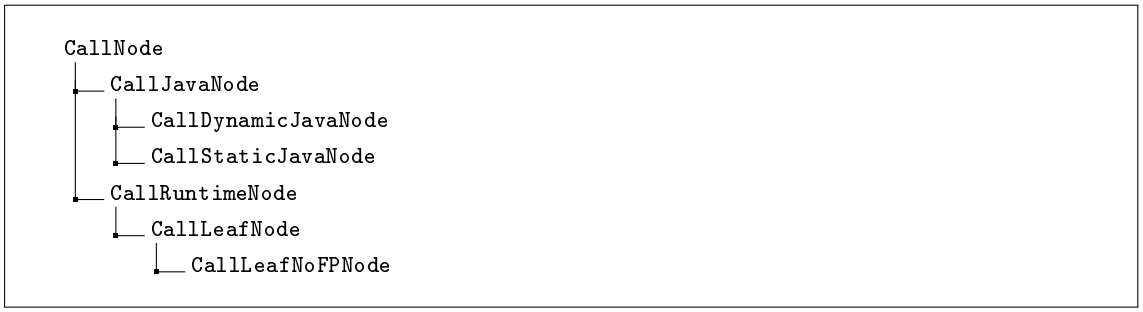


FIG. 6.6: Hiérarchie des nœuds d'appel définis dans l'IR de C2

	JNI section critique	**Intrinsic CallLeafNode**
`()`	147	890
`(double[])`	25	820
`(double[], double[])`	14	785
`(double[], double[], double[])`	9	775

TAB. 6.3: Débit d'appel pour des méthodes statiques vides en millier d'appels par seconde, pour différentes signatures des paramètres d'entrée. La fonction native cible vide est soit appelée via JNI, soit via un intrinsic qui émet un nœud d'appel `CallLeafNode`

de l'élimination d'expression redondante. Le problème est, à ce niveau là, identique à JNI. Cette méthode est donc à privilégier pour des méthodes avec un potentiel d'optimisation de contexte ainsi qu'un coût d'appel négligeable.

6.1.3.3 Implémentation avec modification du back-end

L'implémentation avec modification du back-end consiste à définir un nouveau nœud dans l'IR associé à la méthode intrinsifiée. Ce nœud est par la suite associé à un nœud machine défini dans le back-end de C2 pour l'architecture x86-64 (fichier x86_64.ad) qui sera émis lors de la sélection d'instruction. Comme montré Figure 6.4, le nœud spécifique est généré lors de l'appel à `inline_<intrinsic>`. Par exemple, le nœud associé à la méthode `floorD2I` (décrite Section 6.2.1) est nommé `FloorD2I`.
Il est également possible d'exprimer l'intrinsic avec des nœuds existants dans l'IR ou d'ajouter des instructions dans le back-end afin de générer le code optimisé [179]. Mais cette façon de faire requiert un effort d'implémentation plus poussée aussi bien au niveau de l'IR que du back-end.
La définition d'un nouveau nœud dans l'IR s'accompagne de la définition d'optimisations locales. Ces dernières sont au nombre de 3 :

- *Identity* : cette optimisation supprime le nœud lorsque ce dernier est équivalent à la fonction identité, à savoir lorsqu'il retourne une valeur retournée par un de ses antécédents. La Figure 6.7 montre l'effet de l'optimisation pour le nœud `FloorD2I`.

Un autre exemple concerne le nœud `ADDI` effectuant une addition entière qui peut être supprimé lorsque l'une de ses entrées est nulle ;

- *Global Value Numbering* (GVN) : cette optimisation recherche si un noeud équivalent existe dans le graphe. Un nœud équivalent est un noeud ayant les même entrées et la même fonction. Si tel est le cas uniquement l'un d'entre eux est conservé et le second est supprimé par élimination de code mort ;
- *Constant folding* : cette optimisation transforme le nœud en une constante lorsque les entrées sont constantes et que la fonction peut être évaluée statiquement.

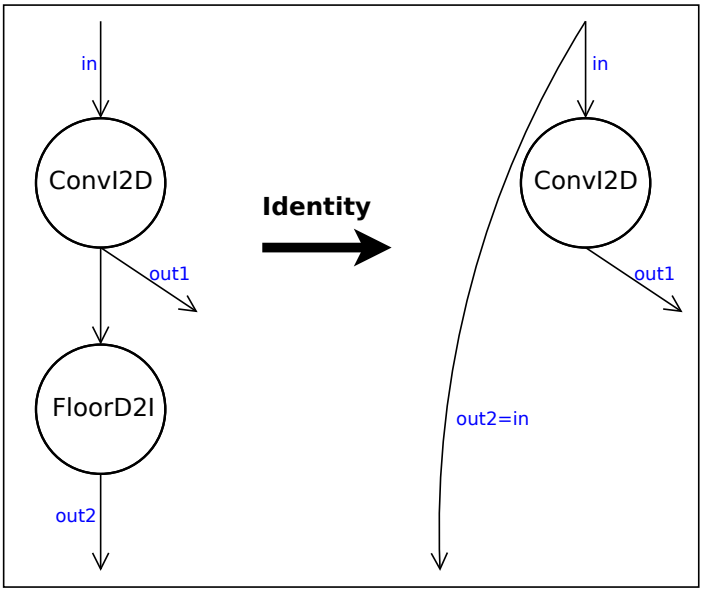


FIG. 6.7: Optimisation *identity* pour le nœud `FloorD2I` qui renvoie le plus grand entier 32-bits inférieur au double en entrée. Lorsque l'entrée de `FloorD2I` est un nœud `ConvI2D` qui convertit l'entier 32-bits entrant `in` en double `out1`, la sortie `out2` de `FloorD2I` est égale à `in` et le nœud peut être supprimé

La Figure 6.8 montre la définition du nœud machine `floorD2I_reg_reg` dans le back-end de C2 pour l'architecture x86-64. Cette dernière contient différentes informations parmi lesquelles :

- Le code assembleur à émettre (directive *ins_encode*). Une instruction assembleur est en fait un micro-générateur écrivant le binaire associé dans le code-cache ;
- Les effets sur les registres (directive *effect*). Par exemple les effets KILL, USE et DEF signifient respectivement que le registre est modifié, lu ou qu'il contient la valeur produite. L'effet TEMP désigne les registres intermédiaires utilisés dans l'instruction ;
- Le sous-arbre de l'IR correspondant à l'instruction (directive *match*). Cette information est utilisée lors de la sélection d'instruction. L'algorithme reconnait des arbres dans l'IR correspondant à des instructions du back-end. Le nœud machine `floorD2I_reg_reg` est par exemple émis lorsque le nœud `FloorD2I` est rencontré ;
- Le coût du nœud machine (directives *ins_pipe* et *ins_cost*). Cette information

contient les unités d'exécution sollicitées par les instructions machines composant le nœud, le nombre d'instructions qui le compose et sa latence approximative. Elle est utilisée lors de la sélection d'instruction pour maximiser l'efficacité du pipeline. Le modèle du pipeline utilisé dans le back-end de C2 est celui des micro-architectures Intel P2/P3.

D'autres informations peuvent être ajoutées comme des prédicats pour l'émission du nœud. Par exemple, dans du nœud machine `floorD2I_reg_reg`, le prédicat concerne la présence de l'extension SSE 4.1 pour l'instruction `ROUNDSD`. Le back-end permet également de définir des optimisations peephole (directive *peepmatch*), ces dernière pouvant être activées ou désactivées avec l'option `-XX:-OptoPeephole`.

```
instruct floorD2I_reg_reg(rRegI dst, regD src, regD tmp, rFlagsReg cr)
{
predicate(useSSE41Intrinsics);
match(Set dst (FloorD2I src));
effect(DEF dst, USE src, KILL cr, TEMP tmp);
ins_encode{
Label done;
__ roundsd($tmp$XMMRegister, $src$$XMMRegister, 1);
__ cvttsd2sil($dst$$Register, $tmp$$XMMRegister);
__ bind(done);
}
ins_pipe(pipe_slow);
}
```

FIG. 6.8: Définition du nœud machine `floorD2I_reg_reg` associé à `FloorD2I` dans le back-end de C2 pour l'architecture x86-64

Avantage et inconvénient de cette implémentation Cette méthode d'implémentation permet de produire un code de meilleur qualité comme d'une part un nœud dans l'IR est l'objet d'optimisations locales (propagation de constante, élimination de code redondant ou mort) et que d'autre part le code généré bénéficie de l'allocation de registre. Son principal inconvénient est qu'elle est plus fastidieuse à implémenter comme elle nécessite la définition d'un nouveau nœud dans l'IR ainsi que le nœud-machine associé dans le back-end.

6.2 Expérimentations

L'intégration d'intrinsic a été expérimentée pour différentes méthodes dont deux sont présentées dans les sections suivantes. Pour chacune d'elles l'implémentation Java ainsi que l'implémentation native optimisée sont décrites. La Section 6.2.1 présente la méthode `floorD2I`[7] calculant la partie entière inférieure d'un double. La Section 6.2.2 présente la

[7] "floor" désigne en anglais la partie entière inférieure

méthode `lcps2d`[8] calculant le signe du produit vectoriel entre deux vecteurs de dimension 2 dont les coordonnées sont des entiers 64-bits (long) en gérant les cas de dépassement arithmétique (*overflow*) lors du calcul du produit vectoriel. Les dépassements arithmétiques surviennent comme les coordonnées représentent des formes planes de l'ordre du nanomètre carré sur des surfaces de l'ordre du décimètre carré, ce qui peut nécessiter plus de 32-bits d'encodage.

6.2.1 Partie entière d'un double

La méthode `floorD2I` calcule la partie entière inférieure d'un flottant double précision. Elle prend un double en entrée et retourne un entier en sortie. L'optimisation appliquée à cette méthode peut être étendue à d'autre type d'arrondi (entier supérieur) et pour des flottants simple-précision.
Cette méthode constitue un exemple de méthode pouvant être optimisée simplement à l'aide d'instructions machine adaptées. Par ailleurs sa très faible granularité implique un coût d'intégration quasi-nul afin de bénéficier de ces instructions. Ces caractéristiques en font un bon cobaye d'expérimentation et ont motivées son intégration sous forme d'intrinsic. Dans l'application étudiée les calculs d'arrondis constituent des points chauds de faible intensité avec une tendance d'exécution diffuse (cf. Section 6.1.1.1). Notons par ailleurs qu'il ne s'agit pas de méthode application-spécifique du fait de son caractère générique. La Section 6.2.1.1 présente la version Java compilée par le JIT et la Section 6.2.1.2 la version native intrinsifiée.

6.2.1.1 Version Java et code généré

L'implémentation Java de la méthode est montrée Figure 6.9. La méthode utilise des branchements conditionnels. Ainsi le code généré par le JIT pour cette méthode dépendra de la distribution des données d'entrées qui détermine les fréquences d'exécution de chaque branche profilée par le runtime (cf. Section 3.3.2.2). Comme vu Section 6.1.2.2, la spécialisation du code aux donnée profilées doit être considérée dans les tests de performance. Néanmoins, comme expliqué dans les sections suivantes, la réorganisation des branches n'entraîne pas de gain de performance comme les chemins exécutés restent similaires à la version générique.

Jeux de données considérés L'implémentation Figure 6.9 définit trois chemins d'exécution. Le premier correspond aux doubles positifs, le second aux doubles strictement négatifs avec une partie fractionnaire non nulle (notés FSN) et le troisième aux doubles strictement négatifs avec une partie fractionnaire nulle (notés ESN). On définit ainsi trois jeux de données d'entrée, un jeu de données correspondant à un type de double vu précédemment, exécutant exclusivement l'un des trois chemins et pour lesquels le code généré est

[8] Dans `lcps2d`, le préfixe "l" fait référence à "long", "cps" est l'acronyme de *cross-product sign* (en français "signe de produit vectoriel") et le suffixe "2d" fait référence à "2 dimensions"

Versions\données	Double positif	Double FSN	Double ESN
Générique	ALL→RET	ALL→SN→FSN→RET	ALL→SN→RET
Spécialisée aux doubles positifs	ALL→RET	ALL→deopt_trap	ALL→deopt_trap
Spécialisée aux doubles FSN	ALL→deopt_trap	ALL→SN→FSN→RET	ALL→SN→FSN→deopt_trap
Spécialisée aux doubles ESN	ALL→deopt_trap	ALL→SN→deopt_trap	ALL→SN→RET

TAB. 6.4: Chemins d'exécution empruntés pour les différentes versions compilées par C2 en fonction du type de double en entrée. Une ligne correspond à une version du code compilée par C2 et une colonne à un type de double en entrée. Les couples code/donnée en vert sont ceux n'exécutant pas de branche de désoptimisation et considérés pour les tests de performance. FSN (resp. ESN) désigne les doubles strictement négatifs avec partie fractionnaire non-nulle (resp. nulle)

spécialisé. De plus on considère un jeu de données aléatoire pour lequel les trois chemins sont exécutés de manière équiprobable (ce dernier étant le plus fidèle quant aux conditions d'utilisation réelles). Pour ce jeu de données, le code généré est générique (i.e. non spécialisé) et similaire à celui produit en désactivant la réorganisation des blocs de base[9] relativement aux fréquences d'exécution profilées.

Versions générées par C2 pour l'implémentation Java La Table 6.5 montre les différentes versions générées par le JIT pour les jeux de données définis précédemment. La Table 6.4 montre les différents chemins exécutés pour chacune des versions en fonction du type de double en entrée. Les chemins sont décrits avec les blocs définis dans le code assembleur.
Le bloc `ALL`, exécuté pour tous les doubles, effectue le test de positivité. Le bloc `SN`, exécuté uniquement pour les doubles strictement négatifs, test la nullité de la partie fractionnaire. Enfin le bloc `FSN`, exécuté uniquement pour les doubles strictement négatifs avec une partie fractionnaire non nulle, décrémente la valeur de retour.
Les versions spécialisées générées par C2 correspondent à une spécialisation du code à l'un des chemins empruntés dans la version générique. Les branches n'étant jamais empruntées sont remplacées par une branche de désoptimisation (repérée par le label `deopt_trap` dans le code).
La première portion du chemin avant exécution du bloc `ALL` est considérée dans le paragraphe suivant. Comme précisé dans ce paragraphe, les valeurs extrêmes emmenant à l'exécution du bloc `VE` ne sont pas considérées dans les tests de performance, ainsi, l'exécution du bloc `ALL` est uniquement précédée de l'exécution du bloc `ENTRY`.

Comportement aux valeurs extrêmes La première opération effectuée par la méthode est la troncature (ou *cast*) de la valeur d'entrée de double à entier (cf. Figure 6.9). Cette troncature correspond au bytecode `d2i` dans la version bytecode et correspond à

[9]Cette désactivation se fait via l'option `-XX:-BlockLayoutByFrequency`

```java
public static int floorD2I(double a){
    int ia = (int) a;
    if (a < 0.) {
        double dia = (double) ia;
        if (dia != a)
            return ia - 1;
    }
    return ia;
}
```

FIG. 6.9: Implémentation Java de la méthode `floorD2I` retournant la partie entière inférieure d'un double (*floor*). La méthode utilisant des branchements conditionnels, le code généré par C2 dépend de la fréquence d'exécution des différentes branches et donc du jeu de données en entrée. Le troncage du double `a` vers l'entier `ia` correspond au bytecode `d2i` respectant certaines spécifications aux valeurs extrêmes

l'exécution des blocs `ENTRY` ou `ENTRY→VE` dans le code assembleur Figure 6.5.
Cette troncature introduit un branchement conditionnel (vers le bloc `VE`) pour les valeurs extrêmes afin de respecter les spécifications de `d2i`[10]. Ces valeurs extrêmes sont $\pm\infty$, `NaN` et celles pour lesquelles la troncature est inexacte[11].
La compilation de `d2i` génère l'instruction `CVTTSD2SI` effectuant la troncature. L'instruction `CVTTSD2SI` renvoie la valeur hexadécimale `0x80000000`[12] dans les cas de troncatures particulières évoquées précédemment. Si cette valeur est retournée le code appelle la routine `d2i_fixup` (bloc `VE`) chargée de retourner les valeurs requises par les spécifications.
Les tests de performance ne considèrent pas les cas d'exécution de la branche exceptionnelle. Par ailleurs son comportement dans ces cas d'exécution n'est pas défini par les spécifications de la méthode comme précisé dans la section suivante. L'opération de troncature constitue un exemple où les spécifications du langage Java peuvent contraindre la génération de code plus performant.

6.2.1.2 Version native optimisée

Le code correspondant est montré Table 6.5. La version native optimisée utilise l'instruction `ROUNDSD` de SSE4.1[13]. L'instruction est utilisée avec un préfixe VEX pour éviter la transition AVX/SSE (comme vu Section 6.1.3.2). L'instruction `ROUNDSD` calcule l'arrondi d'un double selon le mode voulu paramétré par un opérande immédiat. Si l'opérande vaut 0, l'arrondi se fait à l'entier inférieur (*floor*). Ii il vaut 1 l'arrondi se fait à l'entier supérieur

[10] Selon les spécifications [60] `d2i` renvoie l'entier 0 si l'entrée vaut `NaN`. Il renvoie `Integer.MIN_VALUE` (resp. `Integer.MAX_VALUE`) si l'entrée vaut $-\infty$ (resp. $+\infty$) ou si la troncature est trop petite (resp. trop grande)

[11] Les troncatures inexactes correspondent aux cas où la valeur flottante arrondie est trop grande ou trop petite pour être encodée dans un entier

[12] Équivalente à `Integer.MIN_VALUE`

[13] SSE4.1 est un sous ensemble de l'extension SSE4 (pour *Streaming SIMD Extensions version 4*) de Intel64 apparue dans l'architecture Intel Nehalem pour améliorer les performances des algorithmes numériques notamment multimédia

(*ceil*). La sémantique de cette instruction permet de définir des intrinsics pour d'autres signatures et modes d'arrondi en bénéficiant des même gains de performance.
Cette instruction n'est pas générée par C2 et n'est pas déclarée dans la librairie assembleur de HotSpot[14]. Sa définition préalable a donc été un pré-requis dans l'implémentation de l'intrinsic. La seconde instruction utilisée est `CVTTSD2SI` qui, à l'instar de la version Java, est utilisée pour tronquer l'arrondi flottant en entier.

Divergence fonctionnelle avec la version Java La version native optimisée et la version Java compilée n'ont pas le même comportement pour les valeurs extrêmes. La version Java renvoie l'entier 0 si l'entrée vaut `NaN`. Elle renvoie `Integer.MIN_VALUE` (resp. `Integer.MAX_VALUE`) si l'entrée vaut $-\infty$ (resp. $+\infty$) ou si la troncature est trop petite (resp. trop grande). La version native, elle, renvoie systématiquement la valeur `Integer.MIN_VALUE` pour ces valeurs d'entrée particulières. Ainsi les spécifications de la méthode intrinsifiée doivent préciser que le comportement aux valeurs extrêmes est indéterminé. Cette spécification se justifie lorsque son usage applicatif se limite à des valeurs non-extrêmes.

6.2.2 Signe de produit vectoriel avec support de dépassement

La seconde méthode intrinsifiée est la méthode `lcps2d` dont le prototype est montré Figure 6.10. Cette méthode retourne le signe de l'expression $l0 \times l1 - r0 \times r1$ qui correspond au produit vectoriel entre les vecteurs $(l0, r0)$ et $(l1, r1)$. La méthode est considérée dans un contexte d'exécution particulier, problématique en terme de performance, qui est explicité Section 6.2.2.1. La version Java et la version native intrinsifiée sont ensuite décrites Section 6.2.2.2.

6.2.2.1 Cas d'exécution problématiques

La méthode doit garantir que le résultat retourné est valide même en cas de dépassement arithmétique. La Figure 6.10 montre une implémentation Java obsolète de la méthode qui ne considère pas les risques de dépassement arithmétique pouvant fausser le résultat. Le dépassement arithmétique peut survenir lors du calcul des termes $l = l0 \times l1$ ou $r = r0 \times r1$ dont les résultats peuvent excéder le maximum encodable sur 64-bits.
La méthode peut être optimisée en exploitant les cas où le signe du produit vectoriel est déterminé (sous-entendu sans calcul explicite de l et de r). C'est le cas si le signe de l est opposé au signe de r (les signes de l et r pouvant être connus à partir de ceux des paramètres d'entrée), le signe du produit vectoriel est alors le signe de l. Notons que d'autre critères spécifiques permettent de déterminer le signe du produit vectoriel sans calcul de l et r, néanmoins leur pertinence dépend du jeu de données considéré.
Les implémentations de `lcps2d` se placent dans les cas problématiques où le signe est indéterminé, à savoir lorsque l et r sont de même signe, et qui nécessite le calcul de leur valeur

[14] Les instructions assembleurs connues et pouvant être générées par C2 sont déclarées dans le header *cpu/x86/vm/assembler_x86.hpp* dans les sources de HotSpot

```java
public static int lcps2d(long l0, long l1, long r0, long r1){
    long l = l0*l1;
    long r = r0*r1;
    int sign = (l>r)?1:((l<r)?-1:0);
    return sign;
}
```

FIG. 6.10: Implémentation Java obsolète de la méthode `lcps2d` retournant le signe du produit vectoriel entre les vecteurs $(l0, r0)$ et $(l1, r1)$. Le résultat peut être faux en cas de dépassement arithmétique dans le calcul des valeur `l` et `r`

par multiplication. Elles correspondent ainsi à la branche problématique de la version générique de `lcps2d`. Le gain de performance, si l'on regarde la méthode générique, est alors à relativiser par rapport à la proportion de branches problématiques exécutées.
Pour contourner les problèmes de dépassement arithmétique la solution retenue a été d'effectuer les calculs $l = l0 \times l1$ et $r = r0 \times r1$ sur 128 bits.

6.2.2.2 Version Java et version native optimisée

Version Java L'implémentation Java utilise la classe `SignedInt128` de la librairie Java *Apache Hive* [2]. Une instance de cette classe est un objet variable[15] représentant un entier 128 bits par un tableau primitif d'entiers 32-bits de taille 4, chaque entier étant une partie de l'entier 128 bits. La classe `SignedInt128` fournie la méthode `multiplyDestructive` permettant de multiplier deux entiers 128 bits sans allouer une nouvelle instance de `SignedInt128`. Les instances utilisées par la méthode et contenant les paramètres d'entrée sont allouées à l'initialisation de la classe définissant la méthode `lcps2d` et réutilisées à chaque appel de la méthode afin de garantir des performances optimales dans les tests.
La méthode utilisait initialement la classe `BigInteger` du JDK. Cette dernière s'avère néanmoins bien moins performante, un facteur 2,3 est observé en faveur `SignedInt128` sur les tests de performance effectués (cf. Section 6.3). Cette différence s'explique pour au moins deux raisons. D'une part les calculs se font en précision précision infinie pour `BigInteger` et sur 128-bits pour `SignedInt128` ce qui permet dans le second cas de spécialiser les algorithmes. D'autre part les instances de `BigInteger` sont immuables[16]. Ainsi à chacune des multiplications, une nouvelle instance de `BigInteger` est allouée pour contenir le résultat. Par ailleurs, les paramètres d'entrée doivent également être convertis en instance de `BigInteger`, ce qui ne peut être effectué en modifiant des instances pré-allouées du fait de leurs caractères immuables. Ainsi au moins 6 allocations (4 pour les conversions et 2 pour les multiplications) sont nécessaires à chaque appel ce qui pénalise grandement les performances.

[15] Un objet immuable (resp. variable) est un objet dont l'état peut être (resp. ne peut pas être) modifié après avoir été instancié. Contrairement aux instances de `BigInteger` qui sont immuables, les instances de `SignedInt128` sont variables. Les objets immuables peuvent poser des problèmes de performance comme ils entraînent des allocations d'instances à chaque modification

[16] Voir note de bas de page 15

Contrairement à la méthode `floorD2I` vue Section 6.2.1, l'analyse du code source et les tests de performance pour cette méthode ne montrent pas de spécialisation du code sur les différents jeux de données considérés.

Version native optimisée La version native optimisée et instrinsifiée utilise l'instruction machine `IMUL`. Cette dernière effectue les calculs des valeurs l et r sur 128 bits ce qui permet d'éviter les problèmes de dépassement arithmétique. Elle place respectivement les 64 bits supérieurs et les 64 bits inférieurs dans les registres `rdx` et `rax`. Les parties supérieures de l et r sont d'abord comparées si l'une des deux est strictement plus grande le signe est retourné. Si elle sont égales, les parties inférieures sont comparées et le signe est retourné. La multiplication sur 128-bits est ici effectuée par une instruction machine ce qui est la source principale du gain de performance par rapport à la version Java.

Version générique

```
#Input is in xmm0
#Output is in eax
ENTRY:
 vcvttsd2si %xmm0, %eax
 cmpl $0x80000000, %eax
 jne ALL
VE:
 subq $0x8, %rsp
 vmovsd %xmm0, (%rsp)
 call d2i_fixup
 pop %rax
ALL:
 vxorpd %xmm1,%xmm1,%xmm1
 vucomisd %xmm0,%xmm1
 jbe RET
SN:
 vcvtsi2sd %eax,%xmm1,%xmm1
 vucomisd %xmm0,%xmm1
 jp FSN
 je RET
FSN:
 decl %eax
RET:
 ret
```

Version spécialisée aux doubles positifs

```
#Input is in xmm0
#Output is in eax
ENTRY:
 vcvttsd2si %xmm0,%eax
 cmp $0x80000000,%eax
 jne ALL
VE:
 sub $0x8,%rsp
 vmovsd %xmm0,(%rsp)
 call d2i_fixup
 pop %rax
ALL:
 vxorpd %xmm1,%xmm1,%xmm1
 vucomisd %xmm0,%xmm1
 ja deopt_trap
RET:
 ret
```

Version spécialisée aux doubles FSN

```
#Input is in xmm0
#Output is in eax
ENTRY:
 vcvttsd2si %xmm0,%eax
 cmp $0x80000000,%eax
 jne ALL
VE:
 sub $0x8,%rsp
 vmovsd %xmm0,(%rsp)
 call d2i_fixup
 pop %rax
ALL:
 vxorpd %xmm1,%xmm1,%xmm1
 vucomisd %xmm0,%xmm1
 jbe deopt_trap
SN:
 vcvtsi2sd %eax,%xmm1,%xmm1
 vucomisd %xmm0,%xmm1
 jp FSN
 je deopt_trap
FSN:
 dec %eax
RET:
 ret
```

Version spécialisée aux doubles ESN

```
#Input is in xmm0
#Output is in eax
ENTRY:
 vcvttsd2si %xmm0,%eax
 cmp $0x80000000,%eax
 jne ALL
VE:
 sub $0x8,%rsp
 vmovsd %xmm0,(%rsp)
 call d2i_fixup
 pop %rax
ALL:
 vxorpd %xmm1,%xmm1,%xmm1
 vucomisd %xmm0,%xmm1
 jbe deopt_trap
SN:
 vcvtsi2sd %eax,%xmm1,%xmm1
 vucomisd %xmm0,%xmm1
 jp deopt_trap
 jne deopt_trap
RET:
 ret
```

Version intrinsifiée

```
#Input is in xmm0
#Output is in eax
ENTRY:
 roundsd %xmm0, %xmm0
 cvttsd2sil %xmm0, %eax
 ret
```

TAB. 6.5: Versions compilées par C2 et version intrinsifiée de la méthode `floorD2I` (cf. Figure 6.9). Les versions compilées spécialisées génèrent une branche de désoptimisation (désignée par le label `deopt_trap`) à la place des blocs non exécutés. Le code intrinsifié est lui indépendant du jeu de donnée. FSN (resp. ESN) désigne les doubles strictement négatifs avec partie fractionnaire non-nulle (resp. nulle). Pour les versions compilées par C2 le code de création et de restauration de la frame n'est pas montré

6.3 Résultats de performance

Cette section expose les résultats de performance obtenus. La Section 6.3.1 décrit succinctement la méthode de mesure employée ainsi que les différentes implémentations mises en œuvre pour l'analyse des résultats. La Section 6.3.2 expose puis discute les résultats de performance obtenus.

6.3.1 Mesures de performance

Pour les deux méthodes considérées, `floorD2I` et `lcps2d` (cf. Section 6.2), les performances sont mesurées comme un débit d'appel en Op/s (opération par seconde), où une opération désigne un exécution de la méthode. L'environnement utilisé pour les expérimentations est détaillé Figure 2.8 Chap. 2.5.
Les performances sont mesurées via la méthodologie décrite Chapitre 3. Une méthode, dénommée `compute` Chapitre 3, est chargée d'appeler les méthodes testées dans une boucle et de renvoyer la durée d'exécution. Ces méthodes sont montrées Figure 6.6. La performance mesurée vaut alors n/dt Op/s où n est le nombre d'itération de la boucle et dt la durée d'exécution retournée par `compute`. La méthode `compute` prend en paramètre un jeu de données qui est, dans les cas présents, un tableau primitif. La taille du tableau primitif est choisie de manière à loger dans le cache L1 pour limiter le coût de chargement des paramètres en registre et garantir la justesse des mesures (cf. Section 3.3.3.2). Concernant la méthode `floorD2I` le code dépendra du jeu de données comme vu Section 6.2.1. Les différentes implémentations considérées pour analyser les performances sont les suivantes :

- **Java inliné** : la version Java de la méthode est inlinée par le JIT ;
- **Intrinsic sous-routine** : la méthode Java est intrinsifiée. Le JIT génère un appel vers une sous-routine contenant le code natif optimisé (cf. Section 6.1.3.2) ;
- **Intrinsic inliné** : la méthode Java est intrinsifiée. le JIT inline le code natif optimisé (cf. Section 6.1.3.3) ;
- **JNI** : la méthode Java est native. Le JIT génère un appel JNI vers une fonction native contenant le code natif optimisé.

6.3.2 Résultats et observations

La Figure 6.11 montre les performances obtenues pour les différentes implémentations de la routine `floorD2I` (définie Section 6.2.1). Comme rappelé Section 6.1.2, les comparaisons de performance avec les méthodes Java compilées par le JIT doivent prendre en compte la spécialisation du code aux données profilées. Sans cette précaution, le gain de performance obtenu avec la version intrinsifiée peut être dépendant du jeu de données utilisé lors des tests et inexistant dans les cas d'exécution réels.
Pour la méthode `floorD2I`, le code généré par C2 est générique dans le cas d'un jeu de données aléatoire en entrée et spécialisé dans les autres cas. Le gain de performance provenant de la spécialisation du code est cependant nul, les versions spécialisées empruntant

floorD2I	lcps2d

`static int CONSUME;`
`static long`
`compute(double[] data){`
`int acc = 0;`
`int n = data.length;`
`long t0 = System.nanoTime();`
`for(int i=0;i<n;++i)`
`  acc += floorD2I(data[i]);`
`long t1 = System.nanoTime();`
`long dt = (t1-t0);`
`CONSUME = acc;`
`return dt;`
`}`
	`static int CONSUME;`
	`static long`
	`compute(double[] data){`
	`int acc = 0;`
	`int n = data.length/4;`
	`long t0 = System.nanoTime();`
	`for(int i=0;i<n;++i)`
	`  acc+=lcps2d(data[4*i],`
	`  data[4*i+1],`
	`  data[4*i+2],`
	`  data[4*i+3]);`
	`long t1 = System.nanoTime();`
	`long dt = (t1-t0);`
	`CONSUME = acc;`
	`return dt;`
	`}`

TAB. 6.6: Méthodes `compute` mesurant les performances des méthodes `floorD2I` et `lcps2d`. La performance mesurée vaut `n/dt`. Concernant l'implémentation, des accumulateurs (`acc`) servent à prévenir l'élimination de code mort et la boucle d'itération est nécessaire pour garantir la justesse de la mesure (cf. Chap. 3). Les performances sont mesurées dans différents cas où la méthode appelée est (i) en Java pure, (ii) native ou (iii) intrinsifiée

les mêmes chemins d'exécution que la version générique comme vu Section 6.2.1.1.
La performance de la version Java, qui contient des branchements conditionnels, dépend du jeu de données d'entrée contrairement aux performances des autres versions (*Intrinsic sous-routine*, *Intrinsic inliné* et *JNI*) basées sur du code natif optimisé sans branchements conditionnels.
Pour les versions Java spécialisées, ces différences de performance s'expliquent par les différents chemins d'exécution empruntés (montrés Table 6.4) possédant des longueurs différentes. Pour le jeu de données aléatoire il faut ajouter la baisse de performance provenant de la prédiction de branchement.
On observe ainsi un facteur d'accélération de $\times 1,5$ entre l'implémentation *Java inliné* et l'implémentation *Intrinsic inliné* pour des doubles positifs en entrée contre un facteur d'environ $\times 2,8$ pour les autres jeux de données.
La Figure 6.12 montre les performances obtenues pour les différentes implémentations de la routine `lcps2d` (définie Section 6.2.2). Pour cette méthode, les performances obtenues sont indépendantes du jeu de données et on observe un facteur d'accélération d'environ $\times 16$ entre la version *Java inliné* et la version *Intrinsic inliné*.

Impact de la granularité Comme rappelé en introduction du chapitre, la granularité des méthodes est une motivation essentielle pour l'implémentation des intrinsics considérant le coût d'intégration élevé via JNI. Les résultats obtenus valident ces assertions. On

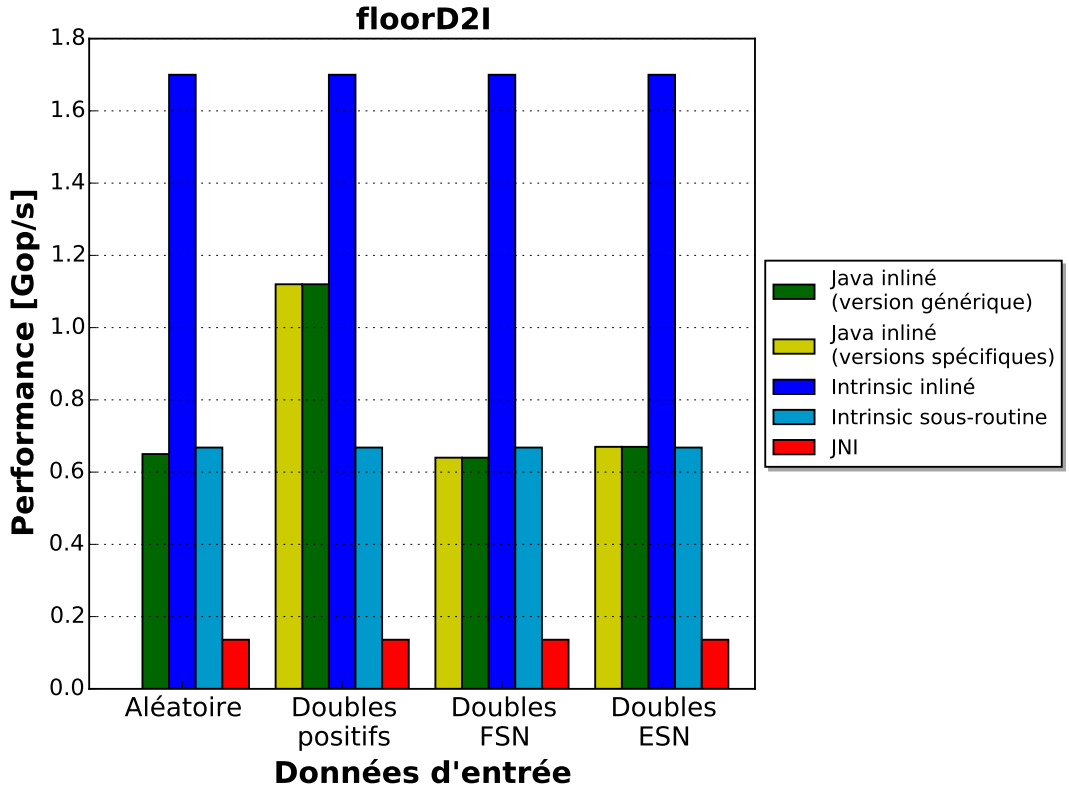


FIG. 6.11: Performance des différentes implémentations de la méthode `floorD2I` pour différents jeux de données en entrée. La performance est mesurée comme un débit d'opérations (Op/s) où une opération correspond à une exécution de la méthode. Concernant les données d'entrée, FSN (resp. ESN) désigne les doubles strictement négatifs avec partie fractionnaire non nulle (resp. nulle). Les versions spécifiques, générées par C2 pour les données d'entrée spécifiques non aléatoires, n'entraînent pas de gain de performance. Pour ces jeux de données, la version générique est obtenue via l'option `-XX:-BlockLayoutByFrequency`. Le facteur d'accélération moyen obtenu par l'implémentation *Intrinsic inliné* contre l'implémentation *Java inliné* est de $\times 2,5$

peut en effet observé une différence de performance importante entre les implémentations *JNI* et *Intrinsic sous-routine* générant toutes les deux un appel vers la sous routine contenant le code natif optimisé (cf. Section 6.1.3.2). Cette différence de performance est d'un facteur $\times 5$ pour la routine `floorD2I` et d'un facteur $\times 3$ pour la routine `lcps2D`. Elle est plus importante pour la routine `floorD2I` qui possède une granularité plus faible.

Comme vu Section 6.1.3, deux techniques d'implémentation des intrinsics sont proposées donnant lieu aux versions *Intrinsic sous-routine* et *Intrinsic inliné*.

Concernant la méthode `floorD2I`, dont la durée d'exécution unitaire (estimée avec l'inverse du débit en Op/s) est inférieure à la nanoseconde, la version *Intrinsic sous-routine* effectuant un appel de sous-routine demeure trop coûteuse pour observer un gain de performance significatif. Les performances sont similaires à celles obtenues avec la version *Java inliné* (i.e. environ 0,6 Gop/s).

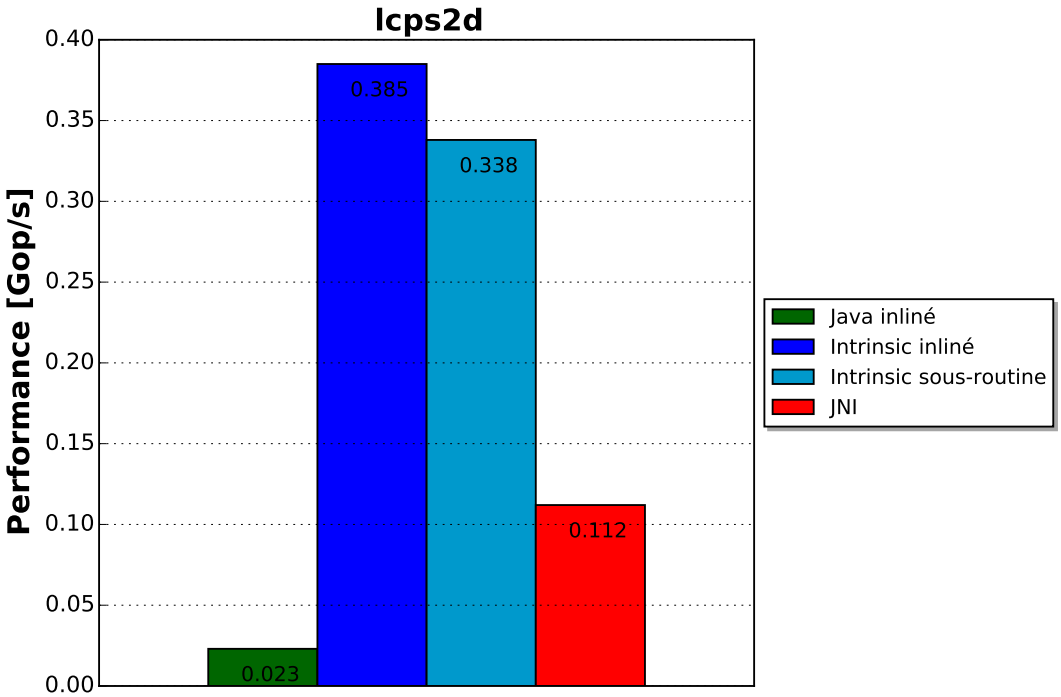


FIG. 6.12: Performance des différentes implémentations de la méthode `lcps2d` (cf. Section 6.2.2). La performance est mesurée comme un débit d'opérations (Op/s) où une opération correspond à une exécution de la méthode. Le facteur d'accélération obtenu par l'implémentation *Intrinsic inliné* contre l'implémentation *Java inliné* est d'environ $\times 16,7$

Concernant la méthode `lcps2d`, dont la durée d'exécution unitaire avoisine les 10 nanosecondes, la version *Intrinsic sous-routine* permet également d'observer un gain de performances significatif, même si sa performance est d'environ 12% inférieure à celle de la version *Intrinsic inliné*.

6.4 Conclusion

Ce chapitre expose les expérimentations d'ajout d'intrinsic effectuées au sein du compilateur C2 de la JVM HotSpot. Ces intrinsics concernent des méthodes de faible granularité pour lesquelles le code généré par C2 sur l'architecture Intel64 a été établi comme sous optimal en analysant l'état du code généré et en exposant des optimisations utilisant des instructions spécifiques.
Les résultats de performance, obtenus par micro-benchmark, valident l'inefficacité de l'utilisation de JNI pour des méthodes de faible granularité. Ils ont montré par ailleurs des accélérations significatives, mesurées à l'aide de micro-benchmark, provenant de l'implémentation des intrinsics comparativement au code généré par C2 (de $\times 2$ à $\times 16$).
La méthodologie utilisée pour implémenter les intrinsics a été exposée. Elle donne des éléments de réponse sur la nature des méthodes à intrinsifier et quelques détails techniques sur leur implémentation.

Deux méthodes d'implémentation sont décrites. L'une, similaire à JNI mais réduisant largement le coût d'invocation, permet une implémentation simple des intrinsics mais néanmoins sous-optimale. L'autre, plus élaborée, permet d'inliner le code natif optimisé dans le contexte d'appel et de bénéficier d'optimisation locale. Elle est à privilégier pour des méthodes de très faible granularité ou ayant un fort potentiel d'optimisation à l'inlining.
Cette étude montre que l'implémentation d'instrinsics côté utilisateur est un moyen efficace, moyennant une bonne connaissance de la JVM HotSpot, d'améliorer les performances de code applicatif critique, particulièrement dans un contexte HPC où la performance est une contrainte majeure.
La prise en compte des informations profilées et du contexte d'exécution dans la génération du code des intrinsics permettrait de générer un code encore plus performant pour certaines méthodes. Par ailleurs, le code généré par le JIT pourrait également être amélioré sans recourt aux intrinsics, mais avec une amélioration du back-end de C2 afin qu'il puisse générer les instructions spécifiques plus adaptées. Néanmoins même si des opportunités d'optimisation sont identifiées coté utilisateur du runtime, ce type d'implémentation doit se faire côté développeur du runtime comme il nécessite des compétences métier plus approfondies. De ce point de vue, le compilateur C2 possède encore un fort potentiel d'amélioration.

Chapitre 7

État de l'art : une analyse comparative

Ce chapitre propose un état de l'art focalisé essentiellement sur les techniques d'optimisation de code applicatif Java pour architectures généralistes du type x86-64.
On regarde en particulier les techniques pouvant être mises en œuvre côté utilisateurs du langage Java et de la JVM (*user-end*). On ignore les techniques hardwares i.e. l'utilisation d'architectures matérielles permettant d'exécuter du bytecode nativement [49, 150]. Dans [87], Kazi et Al. ont montré un panorama global des différentes techniques d'exécution de programme Java avec la performance comme critère central de comparaison.
Les différentes techniques sont comparées en regardant quatre critères majeurs liés au développement d'application d'application déployée en production :

1. **Prix/licence**. Il s'agit d'un critère industriel qui touche à la rentabilité ou à la propriété intellectuelle. Il peut être bloquant concernant l'utilisation de JVM propriétaires haute-performances comme la JVM Zing [5] ou l'utilisation de librairies sous licence contraignante [158] comme la licence GPL ;
2. **Pérennité/robustesse**. Il s'agit d'un critère central qui garantit que la solution employée est robuste et pérenne. Ce critère regarde par exemple si la solution est toujours maintenue et distingue les projets de recherche, pouvant être de nature expérimentale, des solutions intégrées en production ;
3. **Difficulté d'intégration**. Ce critère prend en considération la compatibilité de la solution avec le code existant et les diverses difficultés techniques d'intégration pouvant survenir ;
4. **Efficacité**. Ce critère regarde l'impact sur les performances de la solution employée. Le critère prend notamment en compte le caractère local ou global du gain i.e. si ce dernier impacte une section de code en particulier ou bien toute l'application. Il peut être relatif i.e. dépendre d'un certain contexte d'exécution ou non.

Les différentes techniques analysées sont classées en deux catégories :

- Les techniques d'optimisations statiques de code applicatif discutées Section 7.1 ;
- Les techniques d'optimisations dynamiques du code applicatif discutées Section 7.2.

7.1 Optimisations statiques de code applicatif

Les optimisations statiques sont, comme leur nom l'indique, les optimisations apportées à l'application mises en œuvre statiquement. Elles sont divisées en plusieurs catégories :

- Les optimisations de code Java par une approche dite JIT-Friendly, discutées Section 7.1.1 ;
- Les optimisations reposant sur des optimiseurs de bytecode, discutées Section 7.1.2 ;
- Les optimisations basées sur l'utilisation de code natif, discutées Section 7.1.3.1 ;
- Les optimisations basées sur la compilation statique du code applicatif, discutées Section 7.1.4.

7.1.1 Approche JIT-friendly

La solution la plus simple et privilégiée par les développeurs pour l'amélioration des performances de code applicatif consiste à rester en Java autrement dit d'effectuer des transformations source à source. Elle offre en effet pérennité et facilité d'intégration. Une des approches possibles est l'approche JIT-Friendly.
Une programmation **JIT-Friendly** est une programmation qui prend en considération les optimisations et/ou la politique de compilation du JIT afin de produire un code source plus performant.

7.1.1.1 Positionnement des contributions

L'étude conduite Chapitre 4 vise à quantifier les performances des bytecodes `invokevirtual` et `invokeinterface` servant à la mise en œuvre du polymorphisme et d'identifier des alternatives plus performantes en particulier l'usage de dispatch explicite. Les optimisations identifiées étant en partie relatives à la JVM HotSpot, cette démarche s'inscrit donc dans une approche JIT-Friendly. Les expérimentations menées peuvent être étendues à d'autres bytecodes.
Les gains potentiels sont limités aux zones ciblées à savoir les zones critiques par l'utilisation du polymorphisme caractérisée dans le chapitre correspondant. La valeur du gain est difficile à établir puisqu'elle dépend fortement du contexte d'exécution (i.e. les méthodes ciblées, le potentiel d'optimisation à l'inlining...).
Les optimisations pourraient néanmoins avoir un impact global avec une prise en charge au niveau du runtime notamment en intégrant des optimisations architecture-spécifiques relatives à la prédiction de branchement [107].
Dans le Chapitre 5, Section 5.4.2.1, l'approche JIT-friendly est également utilisée dans l'analyse des patterns vectorisés par le JIT afin de produire la meilleure implémentation.

Cette considération permet d'obtenir un gain de performance allant jusqu'à un facteur $\times 2$ sur la routine implémentant l'algorithme d'Horner.

7.1.1.2 Usages tiers de l'approche JIT-friendly

Cette approche a été explorée dans [9] afin d'améliorer les performances d'une `DGEMV`[1] en jouant sur le facteur de déroulage de boucle et en observant le code généré. Elle montre un gain de performance allant jusqu'à +35%. Elle est cependant conduite avec la version de HotSpot intégrée dans Java 7 dans laquelle l'émission d'instruction AVX par le JIT n'est pas supportée ce qui restreint considérablement les marges de manœuvre.
L'approche JIT-Friendly peut également être utilisée pour adapter la granularité des méthodes aux heuristiques de compilation afin de garantir l'inlining [40]. Cette technique est utilisée dans la librairie Java haute-performance HikariCP[2] [191] qui prend en compte le seuil d'inlining systématique de 35 bytes[3] (portant sur la taille des méthodes au niveau bytecode) afin d'adapter la taille des méthodes critiques. L'outil JITWatch [114] favorise ce type d'optimisation en fournissant des informations utiles à partir des logs de compilation. Il intègre l'outil JarScan d'analyse statique de bytecode qui permet par exemple de lister l'ensemble des méthodes excédant 325 bytes[4] qui est le seuil d'inlining pour les sites d'appel chauds.
Différents guides de bonnes pratiques de programmation orientés performance ont une approche en partie JIT-Friendly. Dans [143] par exemple, les techniques sont adaptées au compilateur dynamique TR-JIT [162] de la JVM J9 d'IBM [161] et concernent par exemple la granularité des méthodes, l'usage privilégié des intrinsics de TR-JIT ou encore l'implémentation des boucles pour faciliter l'élimination du bound-checking à la compilation.

7.1.1.3 Limitations de l'approche

Si l'approche JIT-Friendly est l'approche la plus simple à mettre en œuvre pour les développeurs Java, elle a néanmoins plusieurs défauts.
Le premier est que l'amélioration des performances qu'elle permet reste limitée et requiert une connaissance plus ou moins poussée du fonctionnement interne du JIT. En effet comme l'amélioration reste au niveau bytecode, le code généré par le JIT peut quand même s'avérer sous-optimal. Autrement dit cette approche est inefficace, ou au mieux très fastidieuse, pour faire de l'optimisation bas-niveau.
Le second est que les optimisations JIT-Friendly sont par définition JVM-dépendantes. Ainsi les performances de l'application peuvent elles-mêmes le devenir. Cet argument peut

[1] `DGEMV` est une routine définie dans l'API d'algèbre linéaire BLAS qui effectue l'opération $y \leftarrow \alpha Ax + \beta y$ où A est une matrice, y et x des vecteurs et α, β des scalaires
[2] HikariCP est un pool de connections pour l'interface JDBC
[3] Ce seuil est fixé par l'option `MaxInlineSize` de HotSpot dont la valeur par défaut est 35. Les méthodes dont la taille est inférieure à ce seuil sont systématiquement inlinées. Au delà, l'inlining dépend d'autres facteurs comme la température du site d'appel
[4] Ce seuil est fixé par l'option `FreqInlineSize` de HotSpot qui vaut par défaut 325

néanmoins être mitigé en considérant que les optimisations des compilateurs dynamiques des différentes JVM peuvent être similaires. C'est par exemple le cas pour les méthodes intrinsifiées qui sont souvent similaires d'une JVM à l'autre (comme dans le cas d'HotSpot et J9).
Enfin, considérant que le compilateur dynamique au sein d'une JVM de production est en amélioration continue, certaines optimisations JIT-Friendly peuvent devenir inutiles ou invalides ce qui peut complexifier le code et ajouter un effort de maintenance.

7.1.2 Optimiseurs de bytecode

Les optimiseurs de bytecode permettent d'améliorer les performances de code applicatif avec des transformations source à source. Cependant, contrairement aux techniques citées dans la section précédente, ces derniers n'utilisent pas une approche JIT-Friendly.
Ils effectuent des optimisations statiques non-effectuées par le compilateur `javac` qui compile le code source Java en bytecode. Ces optimisations sont par exemple la propagation de constante, l'élimination de code mort, la factorisation de code redondant, la suppression de champs ou de méthodes non utilisés, l'inlining, le remplacement des appels virtuels par des appels statiques lorsque c'est possible... Beaucoup de ces optimisations sont redondantes avec celles effectuées par le JIT.
Parmi ces optimiseurs on peut citer Soot [178] ou Proguard [63] (le dernier étant utilisé pour l'application étudiée). Dans [178], l'utilisation de Soot montre un gain de +21% sur le benchmark SPECjvm98. Néanmoins ces résultats, basés sur la version Java 1.2 (première version intégrant un compilateur à la volée), sont à mitiger étant donner les améliorations apportés au compilateur dynamique depuis.

Avantages et limitations Les avantages des optimiseurs de bytecode sont leur simplicité de mise en œuvre, leur impact global à toute l'application ainsi que leur potentiel d'optimisation important sur le chargement de classe ou l'interprétation du bytecode autrement dit sur le temps de démarrage de l'application.
Néanmoins ces derniers ont des limitations.
On peut par exemple citer leur faible impact sur des sections de code applicatifs critiques (autrement dit sur les performances asymptotiques), du fait d'une part des optimisations manuelles effectuées par les développeurs sur ce type de code ou encore du fait de la redondance des optimisations par rapport à celles du JIT. De ce point vue l'approche JIT-Friendly offre un plus grand potentiel de gain.
On peut également citer le fait que, tout comme l'approche JIT-Friendly, il demeure impossible d'effectuer des optimisations bas-niveau comme par exemple la vectorisation. Une des alternatives pour contrer cette limitation peut être l'usage de code natif comme vu dans la section suivante.

7.1.3 Interopérabilité avec du code natif

L'idée d'utiliser JNI afin de contourner les problèmes de performance provenant du langage Java est apparue très tôt [57] et demeure d'actualité, considérant l'évolution rapide des architectures et l'écart, toujours existant, entre compilateurs statiques et compilateurs dynamiques.
La Section 7.1.3.1 référence quelques solutions basées sur JNI en les comparant aux contributions apportées.
Même si l'effort, ces dernières années, a été d'avantage porté sur l'amélioration du compilateur dynamique, des solutions ont également été proposées afin d'améliorer l'interopérabilité entre code Java et code natif notamment au regard des performances. Ces dernières offrent d'importantes perspectives et sont présentées Section 7.1.3.2.

7.1.3.1 Techniques basées sur JNI

JNI permet d'améliorer les performances en permettant l'appel à des librairies natives optimisées comme exposé Chapitre 5, Section 5.4.1. Cette alternative a été décrite dès les premières versions de Java afin de bénéficier des performances de librairies optimisées comme BLAS ou MPI [57, 13] ou encore afin d'accélérer le calcul d'éléments finis [116].
L'alternative JNI a, par ailleurs, été particulièrement étudiée pour pallier les limitations du JIT concernant la vectorisation qui demeure une optimisation critique compte tenu des ressources de temps limitées dont dispose le compilateur dynamique [117].

JNI pour la vectorisation Afin de remédier aux limitations du Java concernant la vectorisation du code (cf. Section 5.4.2.1 Chap. 5) de nombreuses propositions basées sur JNI ont émergé [110, 6, 136] au même titre que les contributions apportées [65].
La plupart d'entre elles reposent sur la définition d'une API regroupant des méthodes Java natives servant d'interface avec des fonctions vectorisées machine-spécifiques.
Dans [6] Albert et Al. présente, en plus de l'API Java *Vector-Library*, un outil d'auto-vectorisation basé sur Aparapi[5], agissant au niveau du code source Java. Ce dernier remplace les blocs vectorisables par des appels vers les méthodes définies dans *Vector-Library*. La gamme des patterns vectorisés reste cependant étroite et se limite à des opérations de base comme par exemple l'addition de tableaux.
Dans [136], Parri et Al. expose la conception de l'API jSIMD permettant de fournir un ensemble de méthodes vectorisées aux développeurs sur différentes architectures.
Ces solutions ont cependant des défauts et leurs résultats sont à mitiger. Dans un premier temps, aucune des deux solutions ne fait l'usage de sections critiques (cf. Section 5.2.2 Chap. 6) permettant d'éviter les copies ou l'usage de mémoire native. L'API *Vector-Library* fait usage de mémoire native et son utilisation peut nécessiter des copies entre la Java-heap et

[5] Aparapi est, à l'instar de Rootbeer, un outil de parallélisation automatique permettant de générer des noyaux de calcul pour GPU à partir du code Java [35]

la mémoire native qui ne sont pas prises en compte dans les tests de performance. Concernant jSIMD, les copies sont toujours nécessaires, rendant l'API peu intéressante pour une utilisation à grain fins.
Contrairement aux contributions, les solutions évoquées ne considèrent pas d'autres paramètres comme l'exécution out-of-order et l'alignement des données en combinaison de la vectorisation. Elles ne prennent pas en compte les caractéristiques intrinsèques aux routines comme l'intensité arithmétique dans l'analyse des performances. Enfin pour finir, les expérimentations sont conduites avec des versions inférieures à Java 8 dans lesquelles l'auto-vectorisation avec le support des instructions AVX n'était pas effectif.

7.1.3.2 Alternatives et améliorations de JNI

Les sur-coûts induits lors de l'utilisation de JNI ont été quantifiés dans [95], à la fois pour le coût d'appel via JNI et le coût des fonctions de rappel JNI. Différentes alternatives ou améliorations de l'interface ont été proposées.

Inlining des fonctions natives Dans [160] Stepanian et Al. ont proposé une stratégie afin d'inliner les méthodes natives à la compilation pour le compilateur TR-JIT de la JVM J9. Ils proposent de surcroît de remplacer les fonctions de rappel JNI par des constantes de compilation exprimées également dans la représentation intermédiaire. En dépit des accélérations observées dans le prototype implémenté, la solution n'a pas été implémentée dans la JVM J9 du fait des difficultés techniques qu'elle introduit (concernant par exemple l'inlining de code natif contenu dans des modules externes ou la prise en charge de toutes les fonctions de rappel JNI).

Alternatives à JNI : GNFI et FFI Dans [62] Grimmer et Al. présente GNFI (Graal Native Function Interface), une alternative à JNI implémentée dans le compilateur Graal[6] [169, 193]. GNFI est, du point de vue utilisateur, similaire à JNA [43]. Elle gère automatiquement l'interface vers les fonctions natives externes ce qui permet aux développeurs de rester en Java sans avoir à programmer en C (notons que JNA reste tout de même basée sur JNI). Le fonctionnement de GNFI diffère de celui de JNI par le fait que, dans son cas, l'interface native est gérée coté Java dans des `NativeFunctionHandle` et optimisée par le compilateur Graal, ce qui permet d'améliorer les performances. Les résultats exposent des accélérations pouvant atteindre un facteur $\times 1,9$ par rapport à JNI sur des appels de méthodes vides. Une des restrictions de GNFI est que l'exécution des fonctions natives est bloquante pour le ramasse miette, ce qui peut nécessiter de se rabattre sur JNI.
Le projet OpenJDK Panama [170], qui a pour objectif d'améliorer l'interconnexion entre

[6] Le compilateur Graal est, à l'instar de C2, un compilateur dynamique pour la JVM HotSpot (HotSpot munit du compilateur Graal est aussi qualifié de Graal VM). La particularité de Graal est d'être écrit en Java, comme tout autre librairie du JDK, ce qui lui permet d'exposer des fonctionnalités internes aux utilisateurs et offre de nombreuses perspectives (cf. Section 7.2.1.4). À l'heure actuelle (Java 8) Graal est toujours en évolution et n'est pas utilisé en production.

code Java et code natif, propose également une nouvelle interface appelée Foreign Function Interface (FFI) [129] dont les principes de base afin d'améliorer les performances sont similaires à ceux de GNFI mais pour la version standard de HotSpot. L'interface sera gérée par des `MethodeHandle` [146]. Elle permettra notamment, par l'intermédiaire de structures de données Java spécifiques, d'écrire le code machine associé à une méthode native directement en Java. L'arrivée de FFI est prévue pour Java 10.

Amélioration pour des cas particuliers : l'API Critial Native Afin d'améliorer les performances de fonction natives de hachage cryptographique[7] pour l'architecture SPARC T4, l'API interne *Critical Native* a été ajoutée dans HotSpot. Cette dernière consiste en la modification des conventions d'appel JNI pour les méthodes statiques ayant pour paramètres d'entrée uniquement des types primitifs ou des tableaux primitifs tout comme les méthodes considérées dans l'étude Chapitre 5 (cf. en particulier Section 5.4). Pour ces méthodes les appels aux fonctions JNI `{Get/Release}PrimitiveArrayCritial` (cf. Section 5.2.1.2) sont effectués directement dans la méthode native générée par le JIT, évitant le surcoût de ces fonctions. Par ailleurs le coût d'appel de la méthode Java appelante vers la fonction native est aussi réduit comme il n'y a ni exception, ni synchronisation. Cette nouvelle convention d'appel pour ce type de méthode permet d'améliorer considérablement les performances. Les gains sont similaires à ceux obtenus lors de l'intrinsification avec appel de sous-routine, à savoir un facteur $\times 30$ pour une méthode vide avec un tableau primitif comme paramètre d'entrée (cf. Section 5.2.1.1), mais sans l'effort d'implémentation d'intrinsic. Néanmoins, c'est actuellement une API privée du JDK comme son usage n'est pas entièrement sécurisé. Aucune documentation ou spécification officielle n'est disponible. Son activation dans HotSpot est contrôlé par l'option `-XX:+CriticalJNINatives`. La JVM J9 utilise de son côté un mécanisme propriétaire nommé *Direct2JNI* similaire à *Critical Native* [160].

7.1.4 Compilation AOT

7.1.4.1 Limitations par rapport à la compilation JIT

La compilation AOT (Ahead-Of-Time) est, par opposition à la compilation Just-In-Time, effectuée avant l'exécution de l'application. Ses deux avantages majeurs concernent d'une part la durée allouée à l'optimisation du code qui n'est pas limitée (important dans le cas d'optimisations très coûteuses comme la vectorisation) et d'autre part son gain de performance systématique par rapport à l'interprétation qui permet d'accélérer le démarrage des applications dans un environnement avec compilateur dynamique.
Elle souffre cependant de différentes contraintes limitant fortement son usage par rapport à la compilation dynamique. La première contrainte, et la plus importante dans le contexte

[7] Il s'agit plus précisément de fonctions contenues dans le package `com.oracle.security.ucrypto` comme par exemple `NativeDigest.nativeUpdate`

d'étude, est qu'elle n'est pas adaptée à la compilation du bytecode. Ce dernier nécessitant, pour être optimisé, d'information dynamique comme vu dans l'étude sur le polymorphisme Chapitre 4. Par ailleurs, elle doit être utilisée conjointement à un interpréteur pour supporter le chargement de classes externes (i.e. non liées statiquement) et à un ramasse-miette pour la gestion automatique de la mémoire. La compilation statique limite les performances lors des interactions avec ces différents composants.

7.1.4.2 Solutions existantes

Précédemment à l'avènement de la compilation dynamique, de nombreuses solutions basées sur la compilation AOT, avec des techniques diverses, avaient été proposées afin d'améliorer les performance d'application Java [75, 45, 111, 140].
Actuellement, la compilation dynamique semble s'être imposée tout du moins dans le domaine de l'informatique non-embarquée et encore d'avantage dans le domaine du HPC ou les temps de démarrage sont négligeables par rapport au temps d'exécution asymptotique. L'environnement GCJ (GNU Compiler for Java) [15, 53] est entièrement basé sur la compilation AOT. Son principal avantage est qu'il permet de bénéficier d'une vectorisation avancée par rapport aux capacités du JIT [118] comme il est basé sur l'optimiseur et le back-end de GCC lors de la compilation du code source Java ou bien du bytecode. Néanmoins ses évolutions semblent bloquées depuis 2009[8] et les dernières versions de Java ne sont plus supportées. La compilation AOT est d'avantage mise à profit dans le domaine de l'informatique embarquée compte tenu des ressources limitées dont disposent les applications. On peut citer l'environnement Java propriétaire Excelsior JET [106, 41], pour lequel les principaux bénéfices revendiqués sont, un, la protection contre les décompilateurs et, deux, l'accélération du démarrage des applications. Un autre exemple, cette fois concernant le système Android, est l'environnement ART (Android Runtime). Basé sur la compilation AOT, il a été privilégié à la machine virtuelle Dalvik, basée sur la compilation JIT, afin un, d'accélérer le démarrage des applications, deux, de réduire la consommation d'énergie, et trois, de réduire l'empreinte mémoire [8].

7.1.4.3 Compilation mixte JIT/AOT

L'usage de la compilation AOT reste pertinent dans un contexte de compilation mixte JIT/AOT afin de diminuer l'empreinte du compilateur dynamique et d'accélérer le démarrage des applications tout en maintenant des performances crêtes optimums grâce à la compilation dynamique. Cette combinaison a été envisagée dès l'apparition de la compilation dynamique en Java [151]
Cette solution est déjà utilisée dans certains environnements d'exécution comme Mono [108], une implémentation open source de l'environnement Microsoft .NET. L'environnement Android ART intègrera également un compilateur dynamique dans la version Andoid

[8] Constat relevé le 30 Mai 2016 depuis le site web du projet [53]

N, faisant de lui un environnement mixte AOT/JIT [142].
Dans le cadre du projet Graal il semblerait que l'apport de la compilation AOT dans HotSpot afin d'accélérer le démarrage des applications soit une piste envisagée [26]. La JVM Zing (cf. Section 2.2.1.2 Chap. 2) implémente déjà une technologie baptisée *ReadyNow !* [4]. Cette dernière permet de réutiliser du code préalablement optimisé à travers différentes instances d'application via des directives au compilateur dynamique. Cette technique permet de diminuer le warmup et le coût de la désoptimisation. Le profile de compilation d'une instance d'application peut être sauvé à la fin de l'exécution et réutilisé par des instances ayant un profil de compilation similaire (i.e. même points chauds et même profiling).

7.2 Techniques d'optimisations dynamiques

La seconde famille d'optimisations, après les optimisations statiques vues Section 7.1, sont les optimisations dynamiques. Ces dernières sont caractérisées par une prise en charge au niveau du compilateur dynamique.
Dans cette section, et comme rappelé en introduction de chapitre, on s'intéresse essentiellement aux optimisations pouvant être effectuées coté développeurs Java. On ignore donc l'état de l'art sur les techniques et algorithmes d'optimisations intégrés au sein des compilateurs dynamiques[9], qui constitue un domaine de recherche actif mais dépasse le cadre cette étude.
Les techniques d'optimisation dynamique sont divisées en deux catégories. La première, traitée Section 7.2.1 inclut les techniques permettant d'avoir un contrôle direct sur le code généré, la seconde, traitée Section 7.2.2, inclut les techniques permettant de régler le comportement du compilateur dynamique afin d'améliorer le code généré.

7.2.1 Contrôle du code généré

7.2.1.1 Implémentation d'intrinsics du compilateur dynamique

Le Chapitre 6 propose l'implémentation d'intrinsic dans le compilateur dynamique C2 de la JVM HotSpot. Les intrinsics sont des méthodes particulières pour lesquelles, le code généré par le compilateur est prédéfini et n'est pas à la charge de ce dernier. Leur mécanisme est ainsi, comme étayé dans [50], une des modalités permettant de réconcilier programmation haut niveau et programmation bas niveau. L'implémentation d'intrinsic est très efficace pour trois raisons :

1. Elle permet un contrôle niveau assembleur du code généré et donc de produire un code optimum sur l'architecture sous-jacente ;
2. Elle permet l'inlining du code dans le contexte courant et donc des optimisations locales (propagation de constante, allocation de registre...) ;

[9] On pense par exemple à l'allocation de registre [186], la vectorisation [98], la recherche de sous-expression commune [28], l'analyse de localité [156] ou encore la dés-optimisation [88]

3. Elle offre la possibilité de spécialiser le code aux données profilées par le runtime.

Cette technique a permis d'obtenir un facteur d'accélération de $\times 2,5$ sur une méthode générique de très faible granularité (trois instructions machine) ainsi qu'un facteur $\times 16,7$ sur une méthode application-spécifique de granularité plus élevée (de l'ordre d'une dizaine d'instructions).
Le troisième potentiel d'amélioration n'est cependant pas exploité dans les expérimentations conduites du fait, d'une part, des méthodes considérées, dont le potentiel de spécialisation au données profilées est nul, mais également des difficultés techniques d'implémentation induites.
Le principal inconvénient de l'implémentation des intrinsics concerne l'effort de développement et de maintenance supplémentaire. Elle nécessite, en effet, une connaissance du fonctionnement interne du JIT notamment concernant sa représentation intermédiaire, ses mécanismes de gestion et la définition d'instructions dans le back-end.
Afin de limiter cette contrainte, une implémentation basique par appel de sous-routine a été proposée. Cependant cette dernière n'est pas optimum et entraîne un sur-coût important en particulier sur les routines de faible granularité. Ainsi aucune accélération n'est observée sur la routine de faible granularité considérée dont la performance demeure similaire à celle de la version générée par C2. Une différence de 13% avec l'implémentation optimum est observée sur la routine de plus forte granularité.
Afin de permettre aux développeurs Java de contrôler le code généré différentes techniques facilitant le développement sont présentées Section 7.2.1.4.

7.2.1.2 Solutions similaires

Intrinsics au sein des JVM de production Beaucoup d'intrinsics sont déjà implémentés dans des JVM de production telles que HotSpot ou J9 et leurs performances reposent en partie dessus [68, 119, 61]. Leur nombre ne fait que croitre compte tenu des nouveaux jeux d'instructions intégrés dans les nouvelles architectures. Par exemple la méthode `Math.fma`[10] est l'un des intrinsics introduit dans la version Java 9 de HotSpot [122] afin de bénéficier de l'extension FMA[11] présente dans les architectures Intel à partir de la génération Haswell [66]
Néanmoins, concernant les environnements d'exécution, deux contraintes se posent sur la nature des intrinsics :

- Ils doivent correspondre à des méthodes génériques afin qu'ils soient bénéfiques à toutes les applications ;
- Ils ne doivent pas être, contrairement aux intrinsics de compilateur statique comme GCC, spécifiques à une architecture particulière afin de respecter la portabilité Java.

[10] La méthode retourne la valeur de l'opération $a \times b + c$ (*fused multiply-add*) avec a,b,c pour paramètres
[11] FMA (Floating-point Multiply Add) est une extension du jeu d'instruction x86 fournissant des instructions SIMD du type *fused multiply-add* (cf. note 10)

La première contrainte a motivé l'intégration d'intrinsic application spécifique. La seconde contrainte interdit, par exemple, d'exposer une API fournissant des instructions AVX.

Java Vectorization Interface Dans [115], Nie et Al. ont néanmoins cassé la seconde contrainte en implémentant une API Java nommée JVI (Java Vectorization Interface). Cette dernière expose aux développeurs Java des types vectoriels ainsi que des méthodes pour exploiter les jeux d'instructions SSE et AVX (les méthodes et types sont d'ailleurs définis dans un package nommé `com.intel.jvi` qui marque explicitement la dépendance du code à l'architecture). JVI ne repose pas sur une extension du langage Java pour la définition des types vectoriels qui correspondent à des classes. Néanmoins un vecteur n'est jamais instancié et JVI repose un support de l'interpréteur et du JIT. De ce point de vue, son usage est hors-spécification comme le code utilisant l'API ne peut pas fonctionner sur d'autres JVM. Le support du JIT utilise le mécanisme des intrinsics. La solution a été expérimentée dans le contexte de la JVM Apache Harmony (intégrant le compilateur dynamique Jittrino) qui n'est plus maintenue depuis 2011 [166]. Les résultats de performance montrent des gains de respectivement +55% et +107% sur les benchmarks `scimark.fft` et `scimark.lu` [168] exécutés sur un cœur Intel Nehalem exposant uniquement l'extension SSE (les gains seraient à priori supérieurs sur une architecture supportant l'extension AVX).

7.2.1.3 Limitations pour la génération de code dynamique

Deux autres inconvénients, qui dépendent néanmoins du compilateur dynamique, peuvent être relevés concernant l'implémentation d'intrinsics :

- Le premier concerne la gamme des informations dynamiques utilisables qui sont relativement limitées (cf. Section 3.3.2.2 Chap. 3), et peuvent être insuffisantes pour produire un code spécialisé optimum. Par exemple, hormis si ces dernières sont des constantes de compilation auquel cas il est possible de bénéficier de la propagation de constante, les valeurs des paramètres d'entrée ne peuvent pas être prises en compte pour spécialiser le code ;
- Le second concerne l'émission du code optimisé qui est inliné dans la méthode appelante. Cet aspect peut limiter le bénéfice des optimisations optimistes comme, en cas d'invalidation du code, la dés-optimisation, qui est un mécanisme coûteux, affecte toute la méthode appelante.

Ces deux inconvénients limitent l'utilisation des intrinsics pour faire de la génération de code dynamique. La génération de code dynamique consiste, dans le cas d'un langage compilé statiquement, à remplacer un site d'appel par un générateur de code. Ce générateur est en charge, un, de générer un code spécialisé aux données d'entrée, et deux, d'appeler le code optimisé. Différentes stratégies peuvent être utilisées pour supporter l'invalidation des optimisations allant du dispatch explicite vers différentes versions à la régénération complète du code.

Cette technique est efficace pour des méthodes de granularité assez forte pour couvrir le temps de génération de code (qui dans le pire cas, a lieu à chaque appel) et ayant un fort potentiel de gain par spécialisation aux données d'entrée. Elle s'avère au contraire impraticable pour des routines de très faible granularité comme celles considérées Chapitre 6 Section 6.2 pour l'expérimentation sur les intrinsics.
L'outil deGoal [24] permet d'embarquer, au sein des applications, des générateurs dynamiques de code appelés *compilettes*. L'usage des compilettes expose, sur des processeurs embarqués STxP70, des temps de génération de code au moins 10 fois inférieurs à ceux des JIT traditionnels tout en garantissant un code généré optimum. L'accélération obtenue pour le temps de génération de code est néanmoins à relativiser pour l'usage des intrinsics, dont le mécanisme s'apparente à de la génération de code mais au niveau de la représentation intermédiaire du JIT au lieu du niveau code machine.
Actuellement l'outil deGoal ne supporte pas le langage Java. Par ailleurs un tel mécanisme, dans un contexte de compilation dynamique, doit logiquement être pris en charge par le compilateur dynamique qui définit la politique de compilation. On peut cependant imaginer d'utiliser des générateurs dynamiques de bytecode implémentés en Java (dont l'usage et courant) et effectuant la spécialisation, le bytecode spécialisé sera ensuite compilé dynamiquement et optimisé par le JIT. Mint [182] est une extension de Java permettant d'implémenter et d'utiliser des générateurs de bytecode dans du code applicatif. Il est basé sur la programmation MSP (*Multi-Stage Programming*) [165]. Également basé sur la programmation MSP, l'outil LMS (*Lightweight Modular Staging*) [144], sur lequel sont basés différents projets [120, 163, 145], permet d'implémenter des générateurs de bytecode mais dans le langage Scala. De nombreuses APIs Java (comme Javassist [84] ou Byte Buddy [189]) permettent également de générer du bytecode à la volée.

7.2.1.4 Alternatives

Le compilateur Graal [169] offre des perspectives importantes pour réconcilier la programmation Java et la programmation bas-niveau [138, 193]. Sa particularité est d'être écrit en Java, ce qui permet par exemple d'étendre sa représentation intermédiaire en définissant de nouveaux nœuds ainsi que les optimisations locales associées [37].
Le compilateur Graal implémente le concept de *snippets* [155]. Les snippets sont des méthodes statiques Java repérées par l'annotation `@Snippets` dont la sémantique définit à la fois le bytecode correspondant mais également la représentation intermédiaire à générer à la compilation. Ils reprennent le mécanisme des intrinsics mais facilitent leur implémentation comme elle s'effectue en Java. Graal introduit également des méthodes natives spéciales appelées *node intrinsics* qui permettent de faire référence à des nœuds particuliers dans la représentation intermédiaire. Les snippets offrent de surcroit la possibilité de forcer la spécialisation du code via des annotations portant sur les paramètres d'entrée (comme `@ConstantParameter` forçant la spécialisation du code à la valeur d'entrée) et de contrôler la désoptimsation du code. Cette possibilité est un pas en avant vers la prise en charge de

la génération de code dynamique par le compilateur.
Un exemple d'utilisation des snippets concerne la conversion `double` vers `int`. Comme vu Chapitre 6, Section 6.2.1, les spécifications du bytecode `d2i` effectuant cette conversion peuvent limiter les performances. La classe `AMD64ConvertSnippets` définit ainsi le snippet `d2i` effectuant cette conversion avec les spécifications de l'instruction x86 `CVTTSD2SI` (utilisée pour l'optimisation des performances de la méthode `floorD2I` Section 6.2.1).

Les snippets ne permettent cependant pas, contrairement aux intrinsics, d'inliner de l'assembleur. Leur efficacité suppose une couverture exhaustive des instructions machines par les nœuds de la représentation intermédiaire de Graal ce qui n'est pas garanti étant donnée la portabilité de cette dernière. Ainsi, si une instruction n'est pas définie dans le back-end du compilateur, comme c'était le cas pour l'instruction `ROUNDSD` (cf. Section 6.2.1), les snippets ne permettront pas d'en faire usage.

7.2.2 Réglages du compilateur dynamique

Certaines techniques ne permettent pas d'avoir un contrôle direct sur le code généré mais un contrôle sur le comportement du compilateur dynamique.
Le contrôle du compilateur dynamique a pour objectif d'améliorer son efficacité (cf. Section 2.3.2.1 Chap. 2) en définissant statiquement des directives par l'intermédiaire de fichier de configuration ou bien d'annotations dans le code applicatif. On distingue deux types de directives au compilateur dynamique :

1. Les directives globales qui affectent la politique de compilation dynamique (cf. Section 7.2.2.1) ;
2. Les directives localisées qui affectent des sites applicatifs sélectionnées (cf. Section 7.2.2.2).

7.2.2.1 Directives globales

Politique de compilation Dans [74], Hoste et Al. ont souligné l'importance des réglages du JIT pour les performances et ont proposé un outil pour la recherche automatique d'optimums. Les expérimentations menées sont spécifiques à la machine virtuelle Jikes RVM[12].
L'étude distingue deux paramètres à déterminer pour une méthode donnée.

- Les heuristiques de compilation qui déterminent quelle méthode compilée, quand et avec quelle niveau de compilation. Jikes RVM possède plus précisément trois niveaux de compilation affecté selon la criticité de la méthode à compiler ;

[12] Jikes RVM (pour *Research Virtual Machine*) [173] est un environnement d'exécution Java, libre et open-source destiné à la recherche. Ses particularités sont d'être méta-circulaire, à savoir écrit en Java, et, contrairement à d'autres environnements méta-circulaires, de ne pas utiliser d'interpréteur mais uniquement des compilateurs [7]

- Les plans d'optimisations correspondant à chaque niveau de compilation. Un plan de compilation est plus précisément défini par un ensemble d'options, 33 options booléennes permettant d'activer ou non une optimisation particulière et 10 paramètres à fixer.

L'outil proposé permet d'identifier dans un premier temps les plans Pareto-optimaux (i.e. tels qu'il n'en existe pas d'autres qui fassent mieux en termes de durée de compilation et de qualité du code). Dans un second temps il affecte ces plans aux différents niveaux de compilation puis recherche les heuristiques de compilation optimales pour une affectation donnée. Des techniques d'optimisation combinatoire sont utilisées considérant les vastes domaine de recherche.
Cette approche reste néanmoins très limitée. D'une part les résultats montrent qu'elle ne permet pas d'obtenir des gains de performance en moyenne sur une suite de benchmark par rapport à la configuration standard réglée à la main. Les gains, de l'ordre de +10%, ne sont obtenus qu'en spécialisant la configuration à une instance de benchmark donnée sans prise en compte de la dépendance aux jeux de données d'entrée.
Par ailleurs la recherche d'optimum est très coûteuse. Les résultats montrent 150 heures de recherche pour une suite de 16 benchmarks sur une machine contenant suffisamment de ressources pour effectuer les recherches en parallèle.
Enfin la limitation majeure, inhérente au caractère global des directives, concerne le caractère générique des plans de compilation. L'optimum est recherché en considérant qu'un plan de compilation est global à toutes les méthodes compilées. Cette restriction limite largement le potentiel d'optimisation de code applicatif. Ainsi si les directives globales sont efficaces pour réduire le temps de démarrage de l'application, les directives localisées offre un potentiel d'optimisation plus important comme elle peuvent cibler en particulier les points chauds applicatifs et améliorer les performances asymptotiques de l'application.

Heuristiques d'inlining D'autres études similaires ont été menées afin de déterminer les heuristiques d'inlining optimums. Dans [22] Cavazos et Al. ont utilisé des algorithmes génétiques afin de déterminer les heuristiques optimales pour l'environnement Jikes RVM. Les expérimentations ont montré une réduction du temps d'exécution de 37% pour la suite de benchmarks Dacapo sur une architecture Intel NetBurst.
Par ailleurs, l'outil JMH propose, dans sa suite de microbenchmark, un microbenchmark (`JMHSample_25_API_GA`) permettant de déterminer le réglage de l'inlining optimal pour la JVM HotSpot à l'aide de l'algorithme génétique de Layman.
Plus récemment, Kulkarni et Al. ont utilisé des algorithmes d'apprentissage automatiques pour construire des heuristiques d'inlining optimums [93]. Les résultats exposent un gain de 15% en moyenne sur différents benchmarks pour la JVM HotSpot.
Ces techniques souffrent des mêmes limitations soulignées dans le paragraphe précédent concernant la politique de compilation. À savoir, d'une part, la dépendance du réglage au code testé et aux jeux de données, d'autre part, le caractère général du réglage à toutes les

méthodes et enfin le temps de recherche élevé.

7.2.2.2 Directives localisées

Actuellement, les étapes de compilations effectuées par un compilateur comme le compilateur C2 de la JVM HotSpot sont communes à toutes les méthodes, on dit que la compilation est générique par opposition à spécifique. Les directives localisées permettent de régler les paramètres de compilation pour une méthode ou pour un type de méthode donnés. Elles peuvent ainsi conduire à des gains de performance à la fois en réduisant le temps de compilation (par exemple en supprimant des optimisations inefficaces) et en améliorant la qualité d'exécution (en activant certaines optimisations). Les directives localisées sont particulièrement utiles comme elles peuvent cibler les points chauds applicatifs et améliorer la performance asymptotique des applications.

Techniques basées sur l'apprentissage automatique Beaucoup d'études sont basées sur des techniques d'apprentissage automatique [83, 23, 148] afin d'identifier des heuristiques de sélection d'optimisations. L'apprentissage automatique a pour objectif d'identifier et d'associer une sélection d'optimisation à un type de méthode défini par un vecteur de caractéristiques.
La finalité est ensuite d'intégrer ces heuristiques au sein des compilateurs dynamiques. Dans [83] par exemple, l'intégration d'heuristiques pour la sélection d'optimisation dans le compilateur C2 de HotSpot a entrainé un gain de performance de +6,2% en moyenne (+40% au maximum) sur une suite d'algorithmes génétiques.
Néanmoins, même si elles valident le potentiel du recourt à des directives locales, ces techniques d'apprentissage sont à ranger dans la catégorie amélioration de la compilation que l'on distingue de celle de l'optimisation de code applicatif comme vu en introduction du chapitre.

Annotation du code applicatif D'autres techniques sont basées sur l'annotation du code applicatif. Ces techniques ont par exemple été proposées aux débuts de la compilation dynamique comme une alternative à une politique de compilation gérée par l'environnement d'exécution [10]. L'usage d'annotation a par ailleurs été étudié pour des optimisations spécifiques comme l'élimination du bound-checking [197] qui parfois n'est pas résolu dynamiquement ou encore l'allocation de registres, qui est une optimisation coûteuse au runtime [196].
Ces techniques se rapprochent de la compilation mixte (cf. Section 7.1.4.3) comme les annotations sont générées automatiquement par un compilateur source à source puis utilisées par le JIT au runtime.

Contrôle de la compilation Une initiative prometteuse pour l'usage de directives de compilation localisées est une amélioration intégrée dans la version Java 9 de HotSpot

baptisée *compiler control* [128].
L'objectif est de permettre le contrôle du compilateur dynamique à grain fin et de manière méthode-spécifique. Même si la motivation initiale est plutôt de faciliter l'écriture de benchmarks ou de tests, elle pourra permettre également, comme démontré dans différentes études [23, 148], d'améliorer les performances de l'application. L'ensemble des options contrôlables est cependant trop limité actuellement pour envisager de telles améliorations.
Le contrôle de la compilation est intégré dans le cadre de la technologie ReadyNow ! de la JVM propriétaire Zing (cf. Section 7.1.4.3).

7.3 Conclusion

La Table 7.1 propose une synthèse des différentes techniques d'optimisation de code applicatif Java décrites dans ce chapitre. Les différents critères de comparaison sont définis en introduction. Le critère *prix/licence* n'étant pas déterminant pour la majorité des solutions, il ne figure pas dans la table.
Contrairement au critère *efficacité*, les critères *robustesse/pérennité* et *difficulté d'intégration* peuvent être bloquant dans le choix d'une solution. L'efficacité d'une solution repose sur quatre attributs principaux qui sont :

- Son niveau d'optimisation (bytecode ou bien code natif) ;
- Son niveau d'impact : impact sur les performances asymptotiques ou bien le temps de démarrage de l'application (qui traduit également la capacité à cibler les points chauds applicatif) ;
- Son caractère dynamique ou statique (notamment la capacité de spécialisation du code aux données) ;
- Son impact spécifique à un type de code particulier (comme le polymorphisme) ou général (pouvant impacter tout type de code).

Considérant ces attributs la solution avec une efficacité optimale est donc celle permettant de faire de l'optimisation native et dynamique, de cibler les points chauds applicatifs, et d'être exploitable sur tout type de code.
Les contributions apportées sont caractérisées par un effort de développement croissant accompagné également d'une efficacité croissante. Une des contraintes qui a également guidée le choix de ces solutions et qui ne figure pas dans la table concerne la conservation de l'environnement d'exécution HotSpot pour des raisons plus larges que le critère de performance du code.
La meilleure alternative à court terme considérant les différentes techniques est l'utilisation de l'API *critical-native* permettant l'intégration de code natif à faible coût. Les autres solutions comme le projet Panama ou les Snippets de Graal étant des alternatives prometteuses à plus long terme.

Catégorie	Technique	Pérennité/ robustesse	Difficulté d'intégration	Efficacité
Approche JIT-Friendly	Optimisation du polymorphisme	▲HotSpot-dépendant	✔Java pur	✔ Performance asymptotique ▲ Impact spécifique (polymorphisme) ✘ Optimisations natives
Optimiseurs de byte-code	Proguard, Soot	✔En prod.	✔Automatique	✔ Temps de démarrage ✘ Performance asymptotique ✘ Optimisation native
Techniques basées sur JNI	Analyse JNI vs. Java avec approche JIT-friendly et mesure des points de croisement	▲Portabilité	▲Effort de dev.	✔ Optimisations natives ✔ Performance asymptotique (out-of-order et alignement) ▲ Impact spécifique (Alternative Java pour les méthodes de granularité assez forte) ✘ Optimisations dynamiques
	Aparapi vector-library		✘Mémoire native ✔Autovectorization	✔ Optimisations natives ▲ Performance asymptotique ✘ Méthode de faible granularité ✘ Optimisations dynamiques
	jSIMD		▲Effort de dev.	✔ Optimisations natives ✘ Performance asymptotique (copies systématiques) ✘ Méthode de faible granularité ✘ Pas d'optimisations dynamiques
Alternatives à JNI	GNFI	✘JVM r&d	▲Effort de dev.	✔ Optimisations natives ✔ Performance asymptotique ✔ Méthode de faible granularité ✘ Optimisations dynamiques
	FFI	✘Projet en dev.		
	Critical Native	▲API interne au JDK		
Compilation AOT pure	Excelsior JET	✔En prod.	✔Automatique	✘ Performance asymptotique ✔ Réduction temps de démarrage
	GCJ	✘Plus d'évol.		
Compilation mixte AOT-JIT	ART, Mono	✔En prod.	✘Hors Java	✔ Performance asymptotique ✔ Réduction temps de démarrage
Contrôle du code généré	Implémentation d'intrinsics	▲Portabilité	✘Effort de dev.	✔ Optimisations natives ✔ Optimisations dynamiques ✔ Performance asymptotique ✘ Spécialisation aux données limitée
	JVI	▲Portabilité ✘Hors spec. ✘Expérimental	✔Autovectorisation	
	Graal Snippets	✘JVM r&d	✔API Java	✔ Optimisations dynamiques ✔ Performance asymptotique ▲ Spécialisation aux données ✘ Optimisations natives
	deGoal (générateur de code binaire dynamique)	▲r&d	✘Hors Java ✘Hors JIT	✔ Optimisations natives ✔ Optimisations dynamiques ✔ Performance asymptotique ✔ Spécialisation aux données
	Mint (Générateur de bytecode dynamique)		✘Extension de Java ✘Hors JIT	✔ Optimisations dynamiques ✔ Performance asymptotique ✔ Spécialisation aux données ✘ Optimisations natives
Contrôle du JIT	Directives globales	✔Indépendant de l'application	✘Recherche d'optimum difficile	▲ Performance asymptotique ✔ Temps de démarrage ✘ Limité par les capacités du JIT
	Directives locales			✔ Performance asymptotique ✔ Temps de démarrage ✘ Limité par les capacités du JIT

TAB. 7.1: Analyse comparative multi-critère des techniques d'optimisation de code applicatif Java. Les contributions apparaissent sur fond gris

Chapitre 8

Conclusion et perspectives

8.1 Conclusion générale

Cette thèse s'est focalisée sur la problématique de l'optimisation de code Java pour des points chauds applicatifs de faible granularité dans le contexte d'une application Java industrielle.
Deux aspects distincts mais liés ont été adressés dans cette étude :

- Les limitations du langage Java et du compilateur dynamique pour fournir un code optimum ;
- Le coût d'interfaçage du code de faible granularité avec le reste de l'application.

Afin de fournir un code optimum 3 approches ont été explorées (cf. Figure 2.9 Section 2.5 Chap. 2) : l'optimisation du code Java, l'usage de JNI et l'extension du compilateur dynamique.
Ces 3 approches se placent dans une approche plus globale qualifiée de JIT-friendly reposant sur l'analyse préalable du code généré par le JIT afin d'évaluer son optimalité.
Concernant les coûts d'interfaçage, l'étude couvre tous les types d'appel rencontrés en Java[1] à savoir les appels polymorphiques et monomorphiques de méthodes Java, les appels statiques de méthodes Java pures et les appels statiques de méthodes Java natives avec ou sans les conventions d'appel JNI.

Optimisation du polymorphisme Lorsque le code est jugé sous-optimal la première approche suggère de modifier le code source Java afin que le JIT génère un code de meilleur qualité. Ce travail a été fait Chapitre 4 dans l'objectif d'augmenter le débit des appels polymorphiques vers du code de faible granularité. Le polymorphisme est intensivement utilisé dans les applications Java sans pour autant que son impact sur les performances ne soit connu. L'étude a permis d'identifier les différentes optimisations appliquées par le JIT et les conditions d'émission de ces optimisations. Elle a également permis de souligner les différents paramètres impactant et d'identifier différentes optimisations possibles. Les

[1] En dehors de `invokedynamic`

résultats suggèrent des améliorations potentielles significatives mais malheureusement non expérimentées sur des cas d'exécution réels.
Deux autres points d'amélioration sont à noter. Premièrement, l'utilisation de compteurs matériels permettrait d'étayer l'impact de la prédiction de branchement sur le débit d'appel de manière plus rigoureuse. Deuxièmement l'intérêt de l'optimisation du site d'appel au regard de l'inlining n'est pas mis en avant dans le micro-benchmark générique proposé

Utilisation de JNI La seconde approche suggère l'utilisation de JNI afin de fournir un code natif (développé en C) de meilleure qualité que celui généré par le JIT. L'étude s'est focalisée sur une gamme particulière de routines calculatoires à savoir des routines vectorisables de complexité linéaire. Le coût d'intégration via JNI est alors la principale contrainte à prendre en considération. Pour cela le point de granularité (ou quantité de calcul) à partir duquel l'usage de JNI n'est plus efficace est mesuré en traçant les profils de performance des routines (i.e. la performance en fonction de la quantité de calcul). Dans cette étude, l'approche JIT-friendly a consisté principalement en l'analyse des idiomes vectorisés par le JIT afin d'évaluer la pertinence d'un recourt à JNI. L'étude propose en outre une analyse des profils de performance au regard des caractéristiques de la micro-routine et fournit ainsi des éléments d'analyse. La gamme des idiomes couverte par l'étude s'avère cependant limitée et gagnerait à être étendue (par exemple afin de couvrir tous les idiomes vectorisés par GCC [1]).

Extension du compilateur dynamique La troisième approche explorée consiste en l'extension du compilateur dynamique par l'ajout d'intrinsics application-spécifiques. Cette technique permet d'inliner dynamiquement du code avec un contrôle niveau assembleur lorsque la compilation de la méthode appelant l'intrinsic est déclenchée. Cette approche est optimale d'un point de vue efficacité comme elle permet un contrôle au niveau binaire et de surcroit dynamique. Sa principale limitation vient de sa difficulté d'intégration qui requiert un travaille au niveau de la JVM. De ce point de vue, un travail pour faciliter l'intégration d'intrinsic au niveau de la JVM serait pertinent. Des solutions futures allant dans ce sens sont à prévoir (cf. Section 7.2.1 Chap. 7). Là encore, l'étude gagnerait à être enrichie par des méthodes ayant un potentiel d'optimisation dynamique important pouvant être exploité lors de la génération de code, le potentiel des méthodes considérées étant quasi-nul.

Apports pour la société Aselta Nanographics Différentes contributions ont pu être apportées directement ou indirectement au logiciel Inscale (cf. Section 1.2 Chap. 2) et ces dernières vont se poursuivent au delà de cette thèse dans le cadre d'un contrat à durée indéterminée ; la performance demeurant un critère central pour le logiciel. Le projet de thèse est d'abord né du besoin pour la société de pouvoir situer les performances de l'application, non pas nécessairement par rapport aux standards du HPC, mais d'avantage

par rapport à un développement Java qui se voudrait optimal en terme de performance. De ce point de vu, l'étude a permis de fournir différents retours d'information sur des aspects impactant les performances et pouvant dès lors être pris en compte par les développeurs. Ces aspects recouvrent principalement le polymorphisme mais également des retours sur diverses optimisations pratiquées par le JIT, le fonctionnement interne et le réglage de la JVM HotSpot ou encore les limitations mémoire liées à la localité. Le travail d'optimisation effectué sur différentes sections du logiciel s'est confronté aux limitations du langage Java et du compilateur dynamique et a permis de proposer les différentes approches résumées plus haut. Ces approches offrent aujourd'hui au logiciel Inscale des solutions exploitables pour l'optimisation des sections de code critiques.

8.2 Perspectives

À court terme les travaux en lien avec cette étude vont s'orienter vers l'exploitation des API existantes (ou à venir) permettant l'amélioration des performances à savoir principalement l'API Critical Native et les Snippets (cf. Section 7.1.3.2 et Section 7.2.1.4 Chap. 7). Par ailleurs l'exploitation des approches évaluées dans cette étude va se poursuivre (implémentation d'intrinsics et intégration de code natif). L'utilisation de générateurs de bytecode afin de spécialiser du code de forte granularité est également une piste de travail envisageable afin de maintenir un bon compromis spécialisation/généricité. À plus long terme deux pistes d'étude pertinentes peuvent être soulevées.

Approfondissement de l'étude sur le polymorphisme L'étude sur le polymorphisme réalisée laisse différents points d'investigation en exergue. Le compilateur ne reconnait qu'une gamme limitée de distributions, son extension afin de reconnaître et d'optimiser plus de distributions est une piste intéressante. Par ailleurs, l'analyse de hiérarchie de classe est utilisée uniquement pour optimiser les sites d'appel monomorphiques et pourrait être étendue aux sites d'appel bimorphiques. L'impact du polymorphisme a été analysé au niveau d'un site d'appel isolé, en pratique les sites d'appels peuvent être imbriqués et adjacents. Une analyse globale du polymorphisme permettrait d'approfondir la connaissance sur son impact au niveau des performances et d'identifier des opportunités d'optimisation plus poussées prenant en considération l'historique global et les ressources de l'unité de prédiction de branchement. Enfin l'optimisation du polymorphisme repose sur du profiling qui contient différentes failles (défauts et pollution du profiling). Ces failles peuvent être contournées en substituant le profiling à des données utilisateurs ou des données fournies par l'application au niveau d'un site d'appel et à destination du compilateur dynamique. L'usage de ces données par le compilateur dynamique afin de générer un site d'appel optimal est une piste d'étude intéressante.

Efficacité de la compilation dynamique Comme vu Section 2.3.2.1 Chap. 2, on peut définir une métrique d'efficacité de la compilation dynamique. Actuellement les politiques de compilation dynamique reposent sur des relevés de fréquences et supposent que la distribution des fréquences à l'instant t est une bonne approximation de la distribution des fréquences globale. En pratique cette hypothèse est mise à mal par certaines méthodes entrainant des baisses d'efficacité de la compilation dynamique. Afin d'améliorer l'efficacité de la compilation dynamiques différentes pistes pouvant être combinées attendent d'être explorées :

- La connaissance des courbes de criticité permet d'affecter à une méthode un niveau d'optimisation optimal (cf. Section 2.3.2.1 Chap. 2). Dans le cas d'application exécutant des instances similaires de manière redondante, la prise en compte de la courbe de criticité pour fournir un réglage optimal de la compilation dynamique est une piste intéressante ;
- La compilation AOT (dans le contexte de compilation hybride AOT/JIT) permet de masquer le coût de la compilation dynamique dans les cas où les points chauds sont déterminés statiquement et pour lesquels le potentiel d'optimisation dynamique est limité. Son exploitation permettrait d'améliorer l'efficacité de la compilation dynamique ;
- Le déclenchement de la compilation à l'aide de prédicat permettrait d'augmenter l'efficacité de la compilation dynamique en diminuant la phase de warmup. Ces prédicats pourraient être placés en des points spécifiques du code (entrée de méthode, site d'appel) et évalués par l'interpréteur afin de déclencher la compilation. Ils pourraient également constituer une alternative à l'OSR (cf. Section 2.3.2.2 Chap. 2) ;
- Le réglage méthode-spécifique du compilateur dynamique permet d'adapter le niveau d'optimisation d'une méthode à sa criticité. Actuellement la compilation est générique (i.e. commune à toutes les méthodes) et donc son efficacité est potentiellement sous-optimale.

Liste des Figures

Liste des Tables

Bibliographie

[1] Auto-vectorization in GCC. https://gcc.gnu.org/projects/tree-ssa/vectorization.html (dernière accès : 01-07-2016).

[2] The Apache Software Foundation. Apache Hive TM. https://hive.apache.org/ (dernière accès : 01-07-2016).

[3] Top 500. The List. http://www.top500.org/ (dernière accès : 01-07-2016).

[4] Zing ReadyNow ! : A "No Stalls" Java that starts up fast and stays fast. Technical report, AZUL Systems, 2015.

[5] Zing : The best JVM for the enterprise. Technical report, AZUL Systems, 2015. https://www.azul.com/files/ZingDataSheet_14_09_rev2.pdf (dernière accès : 01-07-2016).

[6] C. Albert, A. Murray, and B. Ravindran. Applying Source Level Auto-vectorization to Aparapi Java. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform : Virtual Machines, Languages, and Tools*, PPPJ '14, pages 122–132, New York, NY, USA, 2014. ACM.

[7] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al. The Jikes research virtual machine project : building an open-source research community. *IBM Systems Journal*, 44(2) :399–417, 2005.

[8] Android Open Source Project. ART and Dalvik. https://source.android.com/devices/tech/dalvik/ (dernière accès : 01-07-2016).

[9] S. Archibald. Towards Native Performance using Java : Optimizing a simplified DGEMV operation. Technical report, OpenGamma, 10 2011.

[10] A. Azevedo, A. Nicolau, and J. Hummel. Java Annotation-aware Just-in-time (AJIT) Complilation System. In *Proceedings of the ACM 1999 Conference on Java Grande*, JAVA '99, pages 142–151, New York, NY, USA, 1999. ACM.

[11] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based Compilation in Execution Environments Without Interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010. ACM.

[12] C. Bentley, S. A. Watterson, D. K. Lowenthal, and B. Rountree. Implicit Java Array Bounds Checking on 64-bit Architecture. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 227–236, New York, NY, USA, 2004. ACM.

[13] A. J. Bik and D. B. Gannon. A note on native level 1 BLAS in Java. *Concurrency - Practice and Experience*, 9(11) :1091–1099, 1997.

[14] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and Realities : The Performance Impact of Garbage Collection. *SIGMETRICS Perform. Eval. Rev.*, 32(1) :25–36, June 2004.

[15] P. Bothner. Compiling Java with GCJ. *Linux Journal*, 2003(105) :4, 2003.

[16] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.*, 16(3) :428–455, May 1994.

[17] J. Buchanan. The GDSII Stream Format. *published online Jun*, 11, 1996.

[18] H. Bui and S. Tahar. Design and synthesis of an IEEE-754 exponential function. In *Electrical and Computer Engineering, 1999 IEEE Canadian Conference on*, volume 1, pages 450–455 vol.1, May 1999.

[19] A. M. Butters. Total cost of ownership : A comparison of C/C++ and Java. *Evans Data Corp-BEA Custom Research-White Paper*, 2007.

[20] D. Buytaert, A. Georges, L. Eeckhout, and K. De Bosschere. Bottleneck Analysis in Java Applications Using Hardware Performance Monitors. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 172–173, New York, NY, USA, 2004. ACM.

[21] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. De Bosschere. Using Hpm-sampling to Drive Dynamic Compilation. *SIGPLAN Not.*, 42(10) :553–568, Oct. 2007.

[22] J. Cavazos and M. F. P. O'Boyle. Automatic Tuning of Inlining Heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 14–, Washington, DC, USA, 2005. IEEE Computer Society.

[23] J. Cavazos and M. F. P. O'Boyle. Method-specific Dynamic Compilation Using Logistic Regression. *SIGPLAN Not.*, 41(10) :229–240, Oct. 2006.

[24] H.-P. Charles, D. Couroussé, V. Lomüller, F. A. Endo, and R. Gauguey. deGoal a tool to embed dynamic code generators into applications. In *Compiler Construction*, pages 107–112. Springer, 2014.

[25] H. Chen et al. Comparative Study of C, C++, C# and Java Programming Languages. 2010.

[26] D. Chuyko. Hotspot & AOT : Now it's time to compile. Technical report, Java SE Performance Team, 4 2016.

[27] M. Cierniak, B. T. Lewis, and J. M. Stichnoth. Open runtime platform : Flexibility with performance using interfaces. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, JGI '02, pages 156–164, New York, NY, USA, 2002. ACM.

[28] C. Click. Global Code Motion/Global Value Numbering. *SIGPLAN Not.*, 30(6) :246–257, June 1995.

[29] C. Click and M. Paleczny. A Simple Graph-based Intermediate Representation. *SIGPLAN Not.*, 30(3) :35–49, Mar. 1995.

[30] M. De Wael, S. Marr, and T. Van Cutsem. Fork/join parallelism in the wild : Documenting patterns and anti-patterns in java programs using the fork/join framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform : Virtual machines, Languages, and Tools*, pages 39–50. ACM, 2014.

[31] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, 1995. Springer-Verlag.

[32] E. D. Demaine. Cache-Oblivious Algorithms and Data Structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

[33] D. Dice, M. S. Moir, and W. N. Scherer III. Quickly reacquirable locks, Oct. 12 2010. US Patent 7,814,488.

[34] L. Djoudi, D. Barthou, P. Carribault, W. Jalby, C. Lemuet, and J.-T. Acquaviva. Exploring Application Performance : a New Tool for a Static / Dynamic Approach. In *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing*, ZIH, Dresden, September 2009. Springer.

[35] J. Docampo, S. Ramos, G. L. Taboada, R. R. Exposito, J. Tourino, and R. Doallo. Evaluation of Java for general purpose GPU computing. In *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, pages 1398–1404. IEEE, 2013.

[36] G. Duboscq, T. Würthinger, and H. Mössenböck. Speculation Without Regret : Reducing Deoptimization Meta-data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform : Virtual Machines, Languages, and Tools*, PPPJ '14, pages 187–193, New York, NY, USA, 2014. ACM.

[37] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM.

[38] M. Dukhan. PeachPy : A Python Framework for Developing High-Performance Assembly Kernels. November 2013.

[39] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Power challenges may end the multicore era. *Communications of the ACM*, 56(2) :93–102, 2013.

[40] B. Evans. Is Your Java Application Hostile to JIT Compilation ?, 11 2014. https://www.infoq.com/articles/Java-Application-Hostile-to-JIT-Compilation (dernière accès : 01-07-2016).

[41] Excelsior LLC. Excelsior JET. www.excelsiorjet.com (dernière accès : 01-07-2016).

[42] N. Faria, R. Silva, and J. Sobral. Impact of Data Structure Layout on Performance. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 116–120, Feb 2013.

[43] T. Fast, T. Wall, and L. Chen. Java Native Access (JNA). https://github.com/java-native-access/jna (dernière accès : 01-07-2016).

[44] S. J. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization : Feedback-directed and Runtime Optimization*, CGO '03, pages 241–252, Washington, DC, USA, 2003. IEEE Computer Society.

[45] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot : An optimizing compiler for java. Number MSR-TR-99-33, page 29. Wiley, March 2000.

[46] A. Fog. *Instruction tables : Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2014. http://www.agner.org/optimize/instruction_tables.pdf (dernière accès : 01-07-2016).

[47] A. Fog. *Optimizing subroutines in assembly language : An optimization guide for x86 platforms*, 2014. http://agner.org/optimize/optimizing_assembly.pdf (dernière accès : 01-07-2016).

[48] A. Fog. *The microarchitecture of Intel, AMD and VIA CPUs : An optimization guide for assembly programmers and compiler makers*, 2014. http://www.agner.org/optimize/microarchitecture.pdf (dernière accès : 01-07-2016).

[49] A. S. Fong, C. H. Yau, and Y. Liu. A hardware-software integrated design for a high-performance java processor. In *Information Technology : New Generations (ITNG), 2012 Ninth International Conference on*, pages 516–521, April 2012.

[50] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying Magic : High-level Low-level Programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[51] E. Francesquini. J-hwloc. https://launchpad.net/jhwloc (dernière accès : 01-07-2016).

[52] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, et al. The Sunway TaihuLight supercomputer : system and applications. *Science China Information Sciences*, pages 1–16, 2016.

[53] GCC Team. The GNU Compiler for the JavaTM Programming Language. www.excelsiorjet.com (dernière accès : 01-07-2016).

[54] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. VMKit : A Substrate for Managed Runtime Environments. *SIGPLAN Not.*, 45(7) :51–62, Mar. 2010.

[55] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.*, 42(10) :57–76, Oct. 2007.

[56] Georgia Institute of Technology. Yeppp ! http://www.yeppp.info (dernière accès : 01-07-2016).

[57] V. Getov, S. F. Hummel, and S. Mintchev. High-performance parallel programming in java : exploiting native libraries. *Concurrency : Practice and Experience*, 10(11-13) :863–872, 1998.

[58] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic : A garbage collector for big data on big numa machines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 661–673. ACM, 2015.

[59] J. Y. Gil, K. Lenz, and Y. Shimron. A Microbenchmark Case Study and Lessons Learned. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE'11, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 297–308, New York, NY, USA, 2011. ACM.

[60] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2015.

[61] N. Grcevski, A. Kielstra, K. Stoodley, M. G. Stoodley, and V. Sundaresan. Java Just-in-Time Compiler and Virtual Machine Improvements for Server and Middleware Applications. In *Virtual Machine Research and Technology Symposium*, pages 151–162, 2004.

[62] M. Grimmer, M. Rigger, L. Stadler, R. Schatz, and H. Mössenböck. An Efficient Native Function Interface for Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform : Virtual Machines, Languages, and Tools*, PPPJ '13, pages 35–44, New York, NY, USA, 2013. ACM.

[63] GuardSquare. ProGuard. https://www.guardsquare.com/proguard (dernière accès : 01-07-2016).

[64] J. Ha, M. Gustafsson, S. M. Blackburn, and K. S. McKinley. Microarchitectural Characterization of Production JVMs and Java Workloads. In *IBM CAS Workshop, Austin, TX*, Feburary 2008.

[65] N. Halli, H.-P. Charles, and J.-F. Méhaut. Performance comparison between Java and JNI for optimal implementation of computational micro-kernels. In *ADAPT 2015 : The 5th International Workshop on Adaptive Self-tuning Computing Systems*, Amsterdam, Netherlands, Jan. 2015.

[66] P. Hammarlund, A. J. Martinez, A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, et al. 4th generation Intel core processor, code-named haswell. In *Hot Chips*, volume 25, 2013.

[67] C. Häubl and H. Mössenböck. Trace-based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 129–138, New York, NY, USA, 2011. ACM.

[68] C. Häubl, C. Wimmer, and H. Mössenböck. Optimized Strings for the Java HotSpot&Trade ; Virtual Machine. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, PPPJ '08, pages 105–114, New York, NY, USA, 2008. ACM.

[69] C. Häubl, C. Wimmer, and H. Mössenböck. Context-sensitive Trace Inlining for Java. *Comput. Lang. Syst. Struct.*, 39(4) :123–141, Dec. 2013.

[70] M. Hertz and E. D. Berger. Automatic vs. explicit memory management : Settling the performance debate. Technical report, Citeseer, 2004.

[71] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[72] V. Horky. PAPI (Performance Application Programming Interface) library with bindings to Java. https://github.com/vhotspur/papi-java (dernière accès : 01-07-2016).

[73] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. DOs and DON'Ts of Conducting Performance Measurements in Java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 337–340, New York, NY, USA, 2015. ACM.

[74] K. Hoste, A. Georges, and L. Eeckhout. Automated Just-in-time Compiler Tuning. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 62–72, New York, NY, USA, 2010. ACM.

[75] C.-H. Hsieh, J. Gyllenhaal, and W.-M. Hwu. Java bytecode to native code translation : the Caffeine prototype and preliminary results. In *Microarchitecture, 1996. MICRO-29.Proceedings of the 29th Annual IEEE/ACM International Symposium on*, pages 90–97, Dec 1996.

[76] R. Hyde. The Fallacy of Premature Optimization. *Ubiquity*, 2009(February), Feb. 2009.

[77] H. Inoue. A Trace-based Java JIT Compiler for Large-scale Applications. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 1–2, New York, NY, USA, 2012. ACM.

[78] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. Adaptive Multi-level Compilation in a Trace-based Java JIT Compiler. *SIGPLAN Not.*, 47(10) :179–194, Oct. 2012.

[79] H. Inoue and T. Nakatani. How a Java VM Can Get More from a Hardware Performance Monitor. *SIGPLAN Not.*, 44(10) :137–154, Oct. 2009.

[80] H. Inoue and T. Nakatani. Identifying the Sources of Cache Misses in Java Programs Without Relying on Hardware Counters. *SIGPLAN Not.*, 47(11) :133–142, June 2012.

[81] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and Optimizing Java 8 Programs for GPU Execution. 2015.

[82] M. R. Jantz and P. A. Kulkarni. Exploring Single and Multilevel JIT Compilation Policy for Modern Machines. *ACM Trans. Archit. Code Optim.*, 10(4) :22 :1–22 :29, Dec. 2013.

[83] M. R. Jantz and P. A. Kulkarni. Performance Potential of Optimization Phase Selection During Dynamic JIT Compilation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 131–142, New York, NY, USA, 2013. ACM.

[84] Jboss-Javassist. Javassist : Java bytecode engineering toolkit since 1999. http://jboss-javassist.github.io/javassist/ (dernière accès : 01-07-2016).

[85] B. JRockit. Java for the Enterprise. *Technical White Paper*, 2003.

[86] M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. *ACM SIGOPS Operating Systems Review*, 34(5) :139–149, 2000.

[87] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja. Techniques for Obtaining High Performance in Java Programs. *ACM Comput. Surv.*, 32(3) :213–240, Sept. 2000.

[88] M. N. Kedlaya, B. Robatmili, C. Caşcaval, and B. Hardekopf. Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines. *SIGPLAN Not.*, 49(7) :103–114, Mar. 2014.

[89] C. Kirkpatrick. Intel AVX State Transitions : Migrating SSE Code to AVX. https://software.intel.com/en-us/articles/intel-avx-state-transitions-migrating-sse-code-to-avx (dernière accès : 01-07-2016).

[90] C. M. Kirsch, H. Payer, and H. Röck. Hierarchical PLABs, CLABs, TLABs in HotSpot. In *Proc. International Conference on Systems (ICONS)*, 2012.

[91] P. Kulkarni and J. Fuller. JIT Compilation Policy on Single-Core and Multi-core Machines. In *Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on*, pages 54–62, Feb 2011.

[92] P. A. Kulkarni. JIT Compilation Policy for Modern Machines. *SIGPLAN Not.*, 46(10) :773–788, Oct. 2011.

[93] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon. Automatic construction of inlining heuristics using machine learning. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–12, Feb 2013.

[94] R. V. Kumar, B. L. Narayanan, and R. Govindarajan. Dynamic Path Profile Aided Recompilation in a Java Just-In-Time Compiler. In S. Sahni, V. K. Prasanna, and U. Shukla, editors, *HiPC*, volume 2552 of *Lecture Notes in Computer Science*, pages 495–505. Springer, 2002.

[95] D. Kurzyniec and V. Sunderam. Efficient cooperation between Java and native codes - JNI performance benchmark. In *In The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[96] P. LaCour, A. J. Reich, K. H. Nakagawa, S. F. Schulze, and L. Grodd. New stream format : progress report on containing data size explosion. In *Advanced Microelectronic Manufacturing*, pages 214–221. International Society for Optics and Photonics, 2003.

[97] K. J. Langman and Z. L. Wang. Method and system for dynamic loop transfer by populating split variables, Sept. 25 2012. US Patent 8,276,131.

[98] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. *SIGPLAN Not.*, 35(5) :145–156, May 2000.

[99] D. Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3) :293–309, 2005.

[100] S.-W. Lee, S.-M. Moon, and S.-M. Kim. Enhanced Hot Spot Detection Heuristics for Embedded Java Just-in-time Compilers. *SIGPLAN Not.*, 43(7) :13–22, June 2008.

[101] S. Liang. *Java Native Interface : Programmer's Guide and Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[102] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[103] S. Maple. Developer Productivity Report 2015 : Java Performance Survey Results. Technical report, ZeroTurnarounds, 2015. http://pages.zeroturnaround.com/RebelLabs---All-Report-Landers_Developer-Productivity-Report-2015.html (dernière accès : 01-07-2016).

[104] F. Martínez, A. J. Rueda, and F. R. Feito. A New Algorithm for Computing Boolean Operations on Polygons. *Comput. Geosci.*, 35(6) :1177–1185, June 2009.

[105] M. Matz, J. Hubi, A. Jaeger, and M. Mark. *System V Application Binary Interface : AMD64 architecture processor supplement*, 10 2013. Draft Version 0.99.6.

[106] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, and A. Yeryomin. Overview of Excelsior JET, a High Performance Alternative to Java Virtual Machines. In *Proceedings of the 3rd International Workshop on Software and Performance*, WOSP '02, pages 104–113, New York, NY, USA, 2002. ACM.

[107] M. Milenkovic, A. Milenkovic, and J. Kulick. Microbenchmarks for Determining Branch Predictor Organization. *Softw. Pract. Exper.*, 34(5) :465–487, Apr. 2004.

[108] Mono Project. Mono : Cross platform, open source .NET framework. http://www.mono-project.com (dernière accès : 01-07-2016).

[109] J. E. Moreira, S. P. Midkiff, and M. Gupta. Supporting multidimensional arrays in java. *Concurrency and Computation : Practice and Experience*, 15(3-5) :317–340, 2003.

[110] J. E. Moreira, S. P. Midkiff, M. Gupta, P. Wu, G. Almasi, and P. Artigas. NINJA : Java for high performance numerical computing. *Scientific Programming*, 10(1) :19–33, 2002.

[111] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa : A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *COOTS*, pages 1–20, 1997.

[112] P. Murray, T. Smith, S. Srinivas, and M. Jacob. Performance issues for multi-language Java applications. In *In Proceedings of the 15 International Parallel and Distributed Processing Symposium 2000 Workshops*, pages 544–551. Springer, 2000.

[113] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the Accuracy of Java Profilers. *SIGPLAN Not.*, 45(6) :187–197, June 2010.

[114] C. Newland. JITWatch. https://github.com/AdoptOpenJDK/jitwatch (dernière accès : 01-07-2016).

[115] J. Nie, B. Cheng, S. Li, L. Wang, and X.-F. Li. Vectorization for Java. In *Proceedings of the 2010 IFIP International Conference on Network and Parallel Computing*, NPC '10, pages 3–17, Berlin, Heidelberg, 2010. Springer-Verlag.

[116] G. Nikishkov, Y. Nikishkov, and V. Savchenko. Comparison of C and Java performance in finite element computations. *Computers & Structures*, 81(24–25) :2401 – 2408, 2003.

[117] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD : Auto-vectorize Once, Run Everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 151–160, Washington, DC, USA, 2011. IEEE Computer Society.

[118] D. Nuzman and A. Zaks. Autovectorization in GCC–two years later. In *Proceedings of the 2006 GCC Developers Summit*, pages 145–158. Citeseer, 2006.

[119] R. Odaira, J. G. Castanos, and T. Nakaike. Do C and Java programs scale differently on Hardware Transactional Memory ? In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 34–43. IEEE, 2013.

[120] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in scala : towards the systematic construction of generators for performance libraries. In *Acm Sigplan Notices*, volume 49, pages 125–134. ACM, 2013.

[121] T. Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *ACM SIGPLAN Notices*, volume 44, pages 377–390. ACM, 2009.

[122] OpenJDK. Add fused multiply add to Java math library. https://bugs.openjdk.java.net/browse/JDK-4851642 (dernière accès : 01-07-2016).

[123] OpenJDK. Add support for AVX512. https://bugs.openjdk.java.net/browse/JDK-8076276 (dernière accès : 01-07-2016).

[124] OpenJDK. Compiler should only use verified interface types for optimization. https://bugs.openjdk.java.net/browse/JDK-6312651 (dernière accès : 01-07-2016).

[125] OpenJDK. Integer/FP scalar reduction optimization. https://bugs.openjdk.java.net/browse/JDK-8074981 (dernière accès : 01-07-2016).

[126] OpenJDK. JEP 142 : Reduce Cache Contention on Specified Fields. https://bugs.openjdk.java.net/browse/JDK-8046132 (dernière accès : 01-07-2016).

[127] OpenJDK. JEP 155 : Concurrency Updates. https://bugs.openjdk.java.net/browse/JDK-8046145 (dernière accès : 01-07-2016).

[128] OpenJDK. JEP 165 : Compiler Control. https://bugs.openjdk.java.net/browse/JDK-8046155 (dernière accès : 01-07-2016).

[129] OpenJDK. JEP 191 : Foreign Function Interface. https://bugs.openjdk.java.net/browse/JDK-8046181 (dernière accès : 01-07-2016).

[130] OpenJDK. JEP 266 : More Concurrency Updates. https://bugs.openjdk.java.net/browse/JDK-8132960 (dernière accès : 01-07-2016).

[131] OpenJDK. Support for vectorizing double precision sqrt. https://bugs.openjdk.java.net/browse/JDK-8135028 (dernière accès : 01-07-2016).

[132] OpenJDKwiki. MethodData. https://wiki.openjdk.java.net/display/HotSpot/MethodData (dernière accès : 01-07-2016).

[133] OpenJDKwiki. PowerPC/AIX Port. http://openjdk.java.net/projects/ppc-aix-port (dernière accès : 01-07-2016).

[134] M. Paleczny, C. Vick, and C. Click. The Java hotspotTM Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM '01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[135] V. Palomares. *Combining static and dynamic approaches to model loop performance in HPC*. PhD thesis, Université de Versailles-Saint Quentin en Yvelines, 2015.

[136] J. Parri, D. Shapiro, M. Bolic, and V. Groza. Returning Control to the Programmer : SIMD Intrinsics for Virtual Machines. *Queue*, 9(2) :30 :30–30 :37, Feb. 2011.

[137] E. Pelegrí-Llopart and S. L. Graham. Optimal Code Generation for Expression Trees : An Application BURS Theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 294–308, New York, NY, USA, 1988. ACM.

[138] T. Pool. New tricks of the GraalVM. 10 2015.

[139] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer : Seamlessly using GPUs from Java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 375–380. IEEE, 2012.

[140] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba : Java For Applications : A Way Ahead of Time (WAT) Compiler. Technical report, Tucson, AZ, USA, 1997.

[141] W. Pugh. Java Memory Model and Thread Specification Revision, 2004. JSR 133.

[142] K. Rahul, J.-C. Beyler, and H. Paul. Android : The Road to JIT/AOT Hybrid Compilation-Based Application User Experience. Technical report, Intel Corporation, 5 2016.

[143] P. Ramarao, J. Siu, P. Pamula, V. Sundaresan, P. Doyle, and G. Chapman. IBM Just-In-Time Compiler for Java. Best practices and coding guidelines for improving performance. Technical report, IBM Software Group, IBM Systems & Technology Group, 11 2008.

[144] T. Rompf and M. Odersky. Lightweight modular staging : a pragmatic approach to runtime code generation and compiled DSLs. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010.

[145] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical Precision JIT Compilers. *SIGPLAN Not.*, 49(6) :41–52, June 2014.

[146] J. Rose. Making native calls from the JVM. Technical Report 0.11, Oracle Corporation, 12 2014. http://cr.openjdk.java.net/~jrose/panama/native-call-primitive.html (dernière accès : 01-07-2016).

[147] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *ACM SIGPLAN Notices*, 41(10) :263–272, 2006.

[148] R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using machines to learn method-specific compilation strategies. In *Proceedings of the 9th Annual*

IEEE/ACM International Symposium on Code Generation and Optimization, pages 257–266. IEEE Computer Society, 2011.

[149] J. L. Schilling. The Simplest Heuristics May Be the Best in Java JIT Compilers. *SIGPLAN Not.*, 38(2) :36–46, Feb. 2003.

[150] M. Schoeberl. *JOP : A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[151] M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver : A Quasi-static Compiler for Java. *SIGPLAN Not.*, 35(10) :66–82, Oct. 2000.

[152] P. Sestoft. Numeric performance in C, C# and Java. *IT University of Copenhagen-Denmark, Version 0.9*, 1, 2010.

[153] S. R. Shenoy and A. Daniel. Intel architecture and silicon cadence : The catalyst for industry innovation. *Technology at Intel Magazine*, 5 :132, 2006.

[154] A. Shipilev. Java microbenchmark harness (the lesser of two evils). Devoxx '13, 2013.

[155] D. Simon, C. Wimmer, B. Urban, G. Duboscq, L. Stadler, and T. Würthinger. Snippets : Taking the High Road to a Low Level. *ACM Trans. Archit. Code Optim.*, 12(2) :20 :20 :1–20 :20 :25, June 2015.

[156] L. Stadler, T. Würthinger, and H. Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 165 :165–165 :174, New York, NY, USA, 2014. ACM.

[157] C. Staelin and L. McVoy. Mhz : Anatomy of a Micro-benchmark. In *USENIX Annual Technical Conference*, 1998.

[158] S. Steer and M. Fitzgibbon. Recueil de fiches explicatives de licences libres. Technical Report 2, Institut National de Recherche en Informatique et en Automatique, 2011.

[159] E. Steiner, A. Krall, and C. Thalinger. Adaptive Inlining and On-stack Replacement in the CACAO Virtual Machine. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 221–226, New York, NY, USA, 2007. ACM.

[160] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining Java Native Calls at Runtime. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 121–131, New York, NY, USA, 2005. ACM.

[161] K. Stoodley. IBM Java Technology. Technical report, Toronto Laboratory, IBM Corporation, 4 2003.

[162] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java just-in-time compiler. *IBM systems Journal*, 39(1) :175–193, 2000.

[163] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite : A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s) :134, 2014.

[164] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using Hardware Performance Monitors to Understand the Behavior of Java Applications. In *Proceedings of the third Usenix Virtual Machine Research and Technologie Symposium*, VM'04, pages 57–72, 2004.

[165] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[166] The Apache Software Foundation. Apache Armony. https://harmony.apache.org/ (dernière accès : 01-07-2016).

[167] The Apache Software Foundation. Commons Math. http://commons.apache.org/proper/commons-math/ (dernière accès : 01-07-2016).

[168] The Apache Software Foundation. SciMark 2.0. http://math.nist.gov/scimark2 (dernière accès : 01-07-2016).

[169] The HotSpot Group. Graal Project : a quest for the JVM to leverage its own J. http://openjdk.java.net/projects/graal/ (dernière accès : 01-07-2016).

[170] The HotSpot Group. Project Panama : Interconnecting JVM and native code. http://openjdk.java.net/projects/panama/ (dernière accès : 01-07-2016).

[171] The HotSpot Group. Project Sumatra. http://openjdk.java.net/projects/sumatra/ (dernière accès : 01-07-2016).

[172] The HotSpot Group. Project Valhalla. http ://openjdk.java.net/projects/valhalla/.

[173] The Jikes RVM project. Jikes RVM. http://www.jikesrvm.org/ (dernière accès : 01-07-2016).

[174] M. M. Tikir and J. K. Hollingsworth. NUMA-aware Java heaps for server applications. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 108b–108b. IEEE, 2005.

[175] TIOBE. TIOBE Index for June 2016. http://www.tiobe.com/tiobe_index (dernière accès : 01-07-2016).

[176] D. Ungar. Generation scavenging : A non-disruptive high performance storage reclamation algorithm. In *ACM Sigplan Notices*, volume 19, pages 157–167. ACM, 1984.

[177] B. Valentine. Introducing Sandy Bridge. https://www.cesga.es/pt/paginas/descargaDocumento/id/135 (dernière accès : 01-07-2016).

[178] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference*

of the Centre for Advanced Studies on Collaborative Research, CASCON '99, pages 13–. IBM Press, 1999.

[179] V. Venkatachalam. A Tutorial on Adding New Instructions to the Oracle Java HotSpot Virtual Machine. Technical report, AMD.

[180] I. Veresov. Tiered Compilation. Technical report, Oracle Corporation, 2013. http://cr.openjdk.java.net/~iveresov/tiered/Tiered.pdf (dernière accès : 01-07-2016).

[181] P. Wendykier. JTransforms. https://sites.google.com/site/piotrwendykier/software/jtransforms (dernière accès : 01-07-2016).

[182] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint : Java multi-stage programming using weak separability. *ACM Sigplan Notices*, 45(6) :400–411, 2010.

[183] Wikipedia. The Free Encyclopedia. List of Java Virtual Machines. https://en.wikipedia.org/wiki/List_of_Java_virtual_machines (dernière accès : 01-07-2016).

[184] Wikipedia. The Free Encyclopedia. List of JVM languages. https://en.wikipedia.org/wiki/List_of_JVM_languages (dernière accès : 01-07-2016).

[185] S. Williams, A. Waterman, and D. Patterson. Roofline : An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4) :65–76, Apr. 2009.

[186] C. Wimmer and M. Franz. Linear Scan Register Allocation on SSA Form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 170–179, New York, NY, USA, 2010. ACM.

[187] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine : An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4) :30 :1–30 :24, Jan. 2013.

[188] C. Wimmer and H. Mössenböck. Automatic Feedback-Directed Object Inlining in the Java HotSpot™ Virtual Machine. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 12–21. ACM, 2007.

[189] R. Winterhalter. Byte Buddy. http://bytebuddy.net (dernière accès : 01-07-2016).

[190] N. Wirth. A plea for lean software. *Computer*, 28(2) :64–68, 1995.

[191] B. Wooldridge. HirakiCP. https://brettwooldridge.github.io/HikariCP/ (dernière accès : 01-07-2016).

[192] Q. Wu and O. Mencer. Evaluating Sampling Based Hotspot Detection. In M. Berekovic, C. Müller-Schloer, C. Hochberger, and S. Wong, editors, *Architecture of Computing Systems – ARCS 2009*, volume 5455 of *Lecture Notes in Computer Science*, pages 28–39. Springer Berlin Heidelberg, 2009.

[193] T. Würthinger. Extending the Graal Compiler to Optimize Libraries. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 41–42, New York, NY, USA, 2011. ACM.

[194] T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination for the Java HotSpot client compiler. In *IN PPPJ '07 : PROCEEDINGS OF THE 5TH INTERNATIONAL SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA*, pages 125–133. ACM, 2007.

[195] T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination in the context of deoptimization. *Science of Computer Programming*, 74(5–6) :279 – 295, 2009. Special Issue on Principles and Practices of Programming in Java (PPPJ 2007).

[196] H. Yamauchi and J. Vitek. Combining Offline and Online Optimizations : Register Allocation and Method Inlining. In N. Kobayashi, editor, *Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 307–322. Springer Berlin Heidelberg, 2006.

[197] D. Yessick and J. Jones. Removal of bounds checks in an annotation-aware JVM. In *SoutheastCon, 2002. Proceedings IEEE*, pages 226–228, 2002.

[198] Z. C. H. Yu, F. C. M. Lau, and C.-L. Wang. Object Co-location and Memory Reuse for Java Programs. *ACM Trans. Archit. Code Optim.*, 4(4) :4 :1–4 :36, Jan. 2008.

[199] W. Zhu, C.-L. Wang, W. Fang, and F. Lau. High-performance computing on clusters : The distributed jvm approach. *High-Performance Computing : Paradigm and Infrastructure*, pages 459–480, 2005.

[200] W. Zhu, C.-L. Wang, and F. Lau. JESSICA2 : a distributed Java Virtual Machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 381–388, 2002.

[201] J. N. Zigman, R. Sankaranarayana, et al. djvm-a distributed jvm on a cluster. 2002.