Académie de Montpellier
Université Montpellier II
Sciences et Techniques du Languedoc

Étude Bibliographique de Master 2

effectué au Laboratoire d'Informatique de Robotique
et de Micro-électronique de Montpellier

Spécialité : **AIGLE**

Étude de machines virtuelles Java existantes et adaptation au hachage parfait

par **Julien PAGÈS**

4 mars 2013

Sous la direction de **Roland DUCOURNAU**

Table des matières

1 Introduction

La problématique générale est l'étude des machines virtuelles Java, et leur possible adaptation au hachage parfait. Java fait partie des langages à objets par opposition aux autres paradigmes de programmation (fonctionnelle, impérative, logique...). La programmation par objets est depuis quelques années le paradigme de programmation le plus utilisé. Le modèle objet permet une bonne modélisation et introduit de nombreux concepts (héritage, encapsulation, classes...) qui facilitent la réutilisation et la structuration des applications. Parmi les langages à objets, un certain nombre sont des langages compilés (C++, Java, C#) d'autres sont plutôt interprétés (Python, Ruby, CLOS). Dans les langages à objets, une sous-catégorie a émergé : les langages fonctionnant avec une machine virtuelle. Cela consiste à transformer un langage source en un langage de plus bas niveau qui sera lui même interprété par une **machine virtuelle**. Cette dernière est une sorte d'émulation d'ordinateur qui interprète ce langage de plus bas niveau. L'avantage principal de ce fonctionnement réside dans la portabilité, le problème de la diversité des architectures est déplacé sur la machine virtuelle plutôt que sur le programme. De plus, plusieurs langages sources peuvent fonctionner avec la même machine virtuelle. Cela permet aussi d'avoir des propriétés intéressantes telles que l'introspection, la compilation *Just-In-Time* et le chargement dynamique de classes, qui seront décrites ultérieurement.

Java et C# sont les principaux langages modernes et grand public fonctionnant avec ces critères (objets, machine virtuelle, chargement dynamique). Dans cette étude sera traité le cas spécifique de Java. Comme C#, Java est un langage à héritage simple (une seule superclasse directe) mais il est possible de définir des interfaces pour avoir une sorte d'héritage multiple qui sera appelé sous-typage multiple. Une interface est une sorte de squelette qu'il est possible d'étendre et qui aide à la réutilisabilité. Dans le monde Java, une classe ne peut hériter que d'une classe mais peut implémenter plusieurs interfaces.
La machine virtuelle qui interprète le code produit à partir de Java est appelée Java Virtual Machine. Il existe plusieurs implémentations de cette machine : certaines sont commerciales (comme Hotspot, celle d'Oracle anciennement Sun), d'autres sont destinées à la recherche.
Le contexte général est donc le langage Java [GJS+12], les machines virtuelles de recherches et en particulier l'implémentation du test de sous-typage et de la sélection de méthodes des objets qui sont typés par des interfaces.
Le sujet du stage est de faire l'état de l'art de ces machines virtuelles de recherches, de choisir celle qui paraît la plus adaptée et d'implémenter le hachage parfait pour le test de sous-typage ainsi que l'envoi de message. Le problème de la sélection de méthode est loin d'être trivial surtout en héritage multiple. Or, en Java la présence des interfaces revient justement à avoir de l'héritage multiple.

1.1 Qu'est ce qu'une Machine Virtuelle ?

Une machine virtuelle (abrégée VM dans la suite) est un concept qui a emmergé dans les années 70 ([Gou01]). Il s'agit de gagner en portabilité en compilant un programme dans un langage intermédiaire qui sera ensuite interprété par la machine virtuelle. L'idéal est bien sûr d'être totalement indépendant de la plateforme cible. De plus, la machine virtuelle assure qu'un même programme fonctionne identiquement sur plusieurs machines, et que ce programme est correct à travers de nombreuses vérifications lors de l'exécution du programme. Le code de la machine virtuelle est produit depuis un langage source (Java, C#) qui est transformé en un langage intermédiaire de plus bas niveau. Ce langage est ensuite interprété par une machine virtuelle spécifique, langage qui dans le monde Java est appelé *bytecode*. Il y a plusieurs types de compilations possibles pour le *bytecode* : le code peut être compilé à la volée (*Just-In-Time*) vers du code machine, ou interprété directement. Une autre approche intermédiaire existe, c'est le cas de HotSpot par exemple [KWM+08].

L'origine de Java est issue des années 90 : Sun conçoit le langage Java et la Java Virtual Machine, Microsoft l'imite en 2000 avec les langages fonctionnant avec .NET et le CLR (Common Language Runtime).
Les performances de ces deux machines virtuelles sont assez proches [Sin03]. La différence réside dans la philosophie de conception. La JVM n'est pas vraiment ouverte à d'autres langages que Java ni à des paradigmes différents. Le CLR a été prévu pour supporter plusieurs langages, y compris ceux venant de la programmation impérative (manipulations de pointeurs) et fonctionnelle (traitement des listes). Les deux plateformes restent toutefois assez proches et constituent probablement l'avenir des langages de programmation de par leur portabilité et les avantages qu'elles offrent. Ces deux systèmes sont très réactifs, ils chargent des classes pendant l'exécution et les compilent à la volée : Compilation *Just-In-Time* (abregé JIT). Ce type de fonctionnement est appelé en **monde ouvert** (*Open World Assumption*) en opposition au monde fermé de C++ (*Closed World Assumption*). L'idée de la compilation JIT a des origines assez lointaines, d'après [Ayc03] le premier article évoquant l'idée est sur LISP, écrit par McCarthy en 1960. Un autre article (en 1966) énonce explicitement l'idée du chargement dynamique pendant l'exécution. Ces idées se sont ensuites répandues et considérablement améliorées pour être maintenant largement utilisées en particulier avec Java et .NET.

1.2 Java virtual Machine (JVM)

Initialement la JVM [LYBB12] a été conçue pour exécuter du *bytecode* issu lui-même du Java. Puis quelques années plus tard certains concepteurs ont l'idée d'utiliser la JVM avec d'autres langages comme par exemple SCALA [Ode09]. Cette machine virtuelle prévue initialement pour des langages à objets à typage statique a évoluée avec la multiplication des langages fonctionnant sur la JVM. Une instruction spéciale pour appeler des méthodes dynamiquement a été rajoutée il y a peu. Ceci constitue à peu près la seule ouverture du Java à d'autres paradigmes, les spécifications de la JVM restant assez peu changées depuis l'origine. L'essence de la JVM est une spécification, il y a donc plusieurs implémentations existantes. Il y a aussi des versions pour les clients et d'autres pour les serveurs avec des problématiques différentes.
Ses caractéristiques essentielles sont :

- Machine abstraite à pile
- Machine sécurisée (typage sûr, pas de manipulations de pointeurs, vérifications multiples)
- Gestion automatique de la mémoire (*Garbage Collector*)
- Instructions orientées objet
- Chargement dynamique
- Multitâches

Étant donné que la JVM est une sorte de processeur en plus complet, des chercheurs se sont penchés sur l'implémentation d'un processeur dédié pour faire exécuter des instructions *bytecode*, ceci dans le but d'accélérer l'exécution de Java. Sun a travaillé sur le sujet avec picoJava [OT97]. Dans [KS00] les auteurs développent l'idée d'un processeur dédié à exécuter des instructions *bytecode* qui viendra compléter le processeur classique qui effectuent les autres tâches. D'après eux les optimisations pour des langages tels que C et Java en même temps sont très compliquées, il est plus simple d'avoir un processeur dédié au Java uniquement.
Le langage évoluant, ils choisissent de travailler avec une puce reconfigurable. La JVM est décrite comme ayant deux grandes composantes :

- les instructions de bas niveau
- le chargement de classes, ramasse-miettes, flot de contrôle...

Le premier point peut être implémenté côté matériel tandis que le deuxième est trop complexe pour ceci. Ils choisissent finalement de traiter côté matériel des opérations constantes, arithmétiques, manipulation de piles en passant par les chargements/saut/comparaisons, plus généralement tout ce qui existe déjà dans les processeurs actuels. Ils rajoutent à ceci la création d'objet, la synchronisation, les appels de méthodes et l'accès aux attributs dans des cas simples qui n'entraînent pas de chargement dynamique de classes. Il est également expliqué que des classes sont stockées dans les caches du processeur, il s'agit des classes courantes correspondant à l'API de Java.
L'idée de la machine virtuelle va donc assez loin, les optimisations actuelles (*inlining*...) étant compatibles avec une implémentation matérielle de la JVM. Toutefois, cette implémentation matérielle semble être une idée qui s'essoufle aujourd'hui bien que des recherches continuent.

1.3 Hachage parfait

Le hachage parfait [Duc08] permet de réaliser la sélection de méthode et le test de sous-typage. Il satisfait également plusieurs exigences pour le test de sous-typage :

- Temps constant
- Espace linéaire
- Compatible avec l'héritage multiple
- Compatible avec le chargement dynamique
- Compatible avec l'*inlining*

D'autres techniques de test de sous-typage existent. Il y a par exemple la coloration, mais elle n'est pas compatible avec le chargement dynamique bien qu'elle permette de conserver les invariants de position [1] et de référence. D'autres tests de sous-typages existent mais ils ne remplissent pas les cinq conditions citées plus haut.

1. L'invariant de position signifie que la position des attributs et méthodes ne dépend pas du type dynamique de l'objet. Cette position sera donc identique dans les sous-classes.

2 État de l'art

2.1 Architecture de la JVM en général

La JVM est une machine abstraite à pile. Les processeurs fonctionnent avec des registres, la compilation de *bytecode* vers du code machine orienté registre est donc généralement faite en plusieurs étapes. Les JVm en mode interprété n'ont pas ce problèmes mais ont des performances moins élevées. Le Java est compilé dans des fichiers *.class* qui contiennent des instructions *bytecode*. Ces dernières sont au nombre de 148 d'après les spécifications de la JVM.

L'architecture globale varie selon les implémentations, quelques composants sont récurrents tels quel : le compilateur, le *Garbage Collector*, le chargeur de classes ou encore l'analyseur syntaxique et lexical. Ces composants et leur relation sont illustrés 3.

2.1.1 Bytecode Java

Le *bytecode* contient des instructions orientées objets, elles sont au nombre de 148 d'après les spécifications 2012 de la JVM. Il y a quatre types d'invocations des méthodes en Java ([LYBB12]). Ces invocations sont présentes sous forme d'instructions *bytecode* :

- Méthode statique *invokestatic*
- Méthode virtuelle *invokevirtual*
- Méthode d'une interface $invokeinterface$
- Méthode virtuelle invoquée statiquement (méthodes privées, constructeurs, appel à *super*()) *invokespecial*

Une autre instruction *bytecode invokedynamic* n'est pas utilisée pour Java mais sert pour implémenter un langage dynamiquement typé sur la JVM.

2.1.2 Fichiers de bytecode

La structure globale du fichier est décrite dans [GJS+12] :

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

Le *bytecode* utilise des références pour accéder aux différents éléments. Ces références sont dans le **constant pool**.

Pour illustrer voici une classe minimale *Personne* :

```
public class Personne
{
    private String nom;
    private String prenom;
    private int age;

    public Personne(String n, String p, int a)
    {
        nom = n;
        prenom = p;
        age = a;
    }

    public void afficher()
    {
        System.out.println("Nom : "+nom+" prenom : "+prenom+" age : "+age);
    }
}
```

Après compilation en *bytecode*, le *constant pool* de la classe *Personne* contient par exemple :

```
Constant pool:
   #1 = Methodref          #16.#30        //  java/lang/Object."<init>":()V
   #2 = Fieldref           #15.#31        //  Personne.nom:Ljava/lang/String;
   #3 = Fieldref           #15.#32        //  Personne.prenom:Ljava/lang/String;
   #4 = Fieldref           #15.#33        //  Personne.age:I
```

Il en va de même pour tous les autres élements (méthodes, noms des variables). La JVM travaille donc avec ce système pour des raisons d'efficacité.

Après le *constant pool* le fichier de *bytecode* contient le code des différentes méthodes. Voici le code compilé de la méthode *main* qui permet de tester le programme.

```
  public static void main(java.lang.String[]);
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=5, locals=2, args_size=1
 0: new           #2  // class Personne
 3: dup
 4: ldc           #3  // String Dupond
 6: ldc           #4  // String Jean
 8: bipush        25
10: invokespecial #5  // Method Personne."<init>":(Ljava/lang/String;Ljava/lang/String;I)V
13: astore_1
14: aload_1
15: invokevirtual #6  // Method Personne.afficher:()V
18: return
```

Ici une instance de *Personne* est crée avec les valeurs suivantes : nom = Dupond, prénom = Jean, âge = 25. Ce code permet d'observer deux types d'appels de méthodes au niveau du *bytecode* :
- *invokespecial* pour le constructeur de Personne
- *invokevirtual* pour l'appel normal à la méthode afficher

2.1.3 Chargement dynamique

La JVM est caractérisée par le chargement dynamique de classes à l'exécution [LB98]. Celui-ci est décrit par :
- Chargement paresseux
- Vérification des types au chargement
- Extensible par le programmeur
- Multiples espaces de nom

La JVM doit assurer un certain niveau de qualité du code. Pour cela, elle doit vérifier la correction des types. Les types ne doivent pas forcément être toujours corrects mais un mauvais type doit entraîner une erreur. Faire des vérifications de types à l'exécution est très coûteux, c'est pourtant l'approche choisie par SmallTalk, Lisp et Self. Java et la JVM font ces vérifications lors du chargement d'une classe. Le composant de la JVM qui gère le chargement des classes à l'exécution est appelé *Class Loader*. Son but est de prendre en entrée un fichier *.class* et de retourner l'objet *Class* correspondant.

En plus d'être paresseux, le chargement implique un test pour s'assurer que le type est définissable et provoquer une erreur si il ne l'est pas. Le type d'une classe est défini par une paire formée par **(Nom de la classe, Chargeur)**. Le chargeur maintient une structure appelée *loaded class cache* contenant les différentes paires. Cela lui permet de ne pas recharger une classe déjà chargée. Il est possible de faire des manipulations avancées avec le chargeur, comme par exemple le redéfinir. Cela peut être utile pour charger des classes depuis un endroit spécifique, ou encore effectuer des vérifications particulières après le chargement.
Néanmoins, pour les classes de bases de Java (celle de la bibliothèque standard) ce n'est pas possible et le chargeur standard sera toujours appelé.

2.1.4 Gestion de la mémoire

La JVM utilise un *Garbage Collector* (GC, également nommé ramasse-miettes) pour gérer sa mémoire. Il y a trois segments mémoires dans la JVM Hotspot [KWM+08] :
- Young generation : objets venant d'être alloués
- Old generation : objets alloués depuis longtemps
- Permanent generation : structures internes

Ce fonctionnement par génération est appelé *ramasse-miettes générationnel*. Il est assez courant dans les machines virtuelles, bien que plusieurs types de *Garbage collector* existent. Les objets dans la *young generation* viennent d'être alloués, ils sont collectés quand il n'y a plus de référence sur eux. Si après plusieurs passages du ramasse-miettes les objets de la *young generation* sont toujours là, ils sont déplacés en *old generation* dans lequel ils peuvent être compactés entre eux avec un algorithme dédié. *Permanent generation* est un segment contenant notamment les classes et données statiques.
Dans ces différents segments la gestion de la mémoire est différente et plusieurs algorithmes s'appliquent. Plusieurs autres VM utilisent ce fonctionnement par génération.
De plus, il existe plusieurs ramasses-miettes dans les JVM, par exemple Hotspot en possède trois et effectue un test pour choisir le meilleur au lancement d'un programme. Il existe :
- Serial GC : léger, monoprocesseur
- Parallel GC : bon temps de réponse, multiprocesseurs

– Concurrent GC : faible latence sur plusieurs processeurs

La libération de la mémoire fonctionne en trois étapes ([C+03]) : la première étape est de trouver les références directes sur les objets depuis le programme. Pour gérer ces références sur les objets, un $objects - maps$ est construit lors de la compilation, il doit être maintenu en temps réel. Généralement, quelques bits sont positionnés dans l'entête des objets pour conserver les informations pour le GC. La deuxième étape consiste à trouver les objets atteignables depuis ces références. La troisième étape libère la mémoire, les objets non trouvés dans les deux premières phases sont désalloués, libérant ainsi de l'espace. Le parcours de la mémoire est une opération très coûteuse, c'est pourquoi les GC ont un rôle central dans les performances d'une VM.

2.1.5 Optimisations

Lors de la phase de compilation du Java, il est déjà possible d'optimiser l'exécution du programme. Une analyse de la hiérarchie de classes (*Class Hierarchy Analysis*, abrégée en *CHA*) permet de détecter certains sites d'appels monomorphes [2] qui sont caractérisés par l'unicité de la méthode éligible.
Dans ces cas là, il y a deux possibilités :

– Si la méthode est courte elle peut être inlinée
– Si la méthode est assez lourde, un appel statique peut être fait avec l'instruction *bytecode* *invokestatic*.

Dans [PVC01] sont décrites des optimisations pouvant être faites en deux temps : lors de la compilation avec de l'analyse statique et lors de l'exécution avec du *profiling*. En effet, Hotspot interprète et fait de la compilation JIT. Si, lors de la compilation, le site n'est pas monomorphe mais qu'il s'avère l'être lors de l'exécution, ils choisissent d'effectuer un *inlining* [3] ou un appel statique.
De plus, lors de cette compilation, des analyses sont faites pour permettre des optimisations, par exemple l'analyse des structures des classes. Certaines JVM utilisent uniquement de l'analyse à priori. Mais la plupart utilisent les deux approches combinées.

Inlining et désoptimisation

De manière générale, une analyse statique du code permet de détecter un certain nombre de sites monomorphes (une seule méthode éligible pour ce site d'appel). Bien entendu, les appels statiques et les classes méthodes d'une classe finale sont éligibles à l'*inlining*. Dans ce cas-là un appel statique ou un *inlining* peuvent être effectués. Si pendant l'exécution une classe est chargée et invalide l'optimisation précédente, il faut revenir en arrière pour garantir un programme correct. Ce procédé est appelé désoptimisation.

Dans le cas de Hotspot, ils reviennent en arrière en repassant en mode interprété lors d'une désoptimisation.

De manière générale, ce procédé entraîne l'utilisation du *On-Stack Replacement*, c'est-à-dire des changements sur la pile pour maintenir un code correct. Le code de la méthode est remplacé par le code de la nouvelle, et l'adresse de retour de l'ancienne méthode est elle-même remplacée.

2. Un site d'appel monomorphe a une seule méthode éligible (méthode d'une classe finale de la hiérarchie par exemple).
3. Le code de la méthode appelée est placée à l'endroit de l'appel.

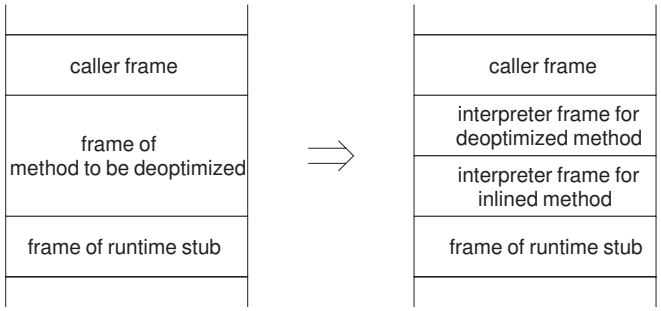

FIGURE 1 – Schéma du *On-Stack Replacement* dans Hotspot d'après [KWM+08]

Il est possible d'éviter d'utiliser ce mécanisme, qui est d'ailleurs assez coûteux, en utilisant la propriété de pré-existence. C'est ce que proposent les auteurs de [DA99], c'est-à-dire inliner quand on est sûr que cela n'entraînera pas de *On-Stack Replacement*. Au delà du coût intrinsèque du mécanisme, maintenir les informations permettant de revenir en arrière sur une optimisation est également complexe.

2.2 Test de sous-typage et sélection de méthode

La programmation par objets introduit la notion de type et donc de sous-typage. Le type statique d'un objet est déterminé à la compilation mais son type dynamique n'est connu qu'à l'exécution. Il peut donc être nécessaire de tester si un objet est instance d'une classe. Ou encore, si le type de cet objet est sous-type d'un type donné.

L'adresse de la méthode à exécuter dépend du type dynamique de l'objet receveur (sur lequel s'applique la méthode). La sélection de la bonne méthode, également appelée liaison tardive ou envoi de message est donc un vrai problème. En héritage simple, les méthodes et attributs sont à position invariante dans la mémoire. Un accès direct est donc possible, par contre, en héritage multiple cette position invariante n'existe plus. Ce problème est encore complexifié avec le chargement dynamique car l'ensemble de classes grandit à l'exécution.

Plusieurs tests de sous-typages et plusieurs mécanismes de sélection de méthode existent. Il est possible de mesurer leur efficacité par le nombre de cycles de processeur qu'ils utilisent.

Le but est d'en avoir un qui remplit les critères suivants [Duc08], déjà énoncés précédemment :

1. Temps constant
2. Espace linéaire
3. Compatible avec l'héritage multiple
4. Compatible avec le chargement dynamique
5. Compatible avec l'*inlining*

Le premier point est important car le test de sous-typage est très souvent utilisé (implicitement ou explicitement). L'implémentation en temps constant d'un mécanisme souvent utilisé est donc un gage d'efficacité si ce temps est raisonnable. L'espace linéaire n'est pas vraiment possible au sens strict du terme étant donné que certaines données des superclasses sont recopiées dans les sous-classes. Le but est d'avoir un espace avec une croissance linéaire par rapport au nombre de classes. Certaines implémentations sont quadratiques par rapport au nombre de classes dans le pire des cas, bien que linéaires dans la taille de la relation de spécialisation.

L'héritage multiple est une nécessité pour la modélisation. [Duc08] cite l'exemple des ontologies qu'il est difficile de modéliser sans héritage multiple. Une forme acceptable demeure néanmoins dans le sous-typage multiple comme Java et C#. Une autre preuve de sa nécessité est la rareté de langages à sous-typage simple. Les langages ne possédant pas de sous-typage multiple sont souvent en phase de mutation pour l'acquérir (c'est le cas de Ada). Le chargement dynamique est devenu assez courant aujourd'hui, l'implémentation du test de sous-typage doit donc en tenir compte. Une manière a priori simple de répondre à ce problème est de proposer une implémentation incrémentale. Une autre approche consistant à tout recalculer a déjà été expérimentée mais dans le pire des cas tout doit être recalculé.
Enfin, l'*inlining* est une optimisation très largement utilisée aujourd'hui. Le test de sous-typage car il est très souvent utilisé, doit être suffisamment petit pour pouvoir être inliné. Seront donc comparés quelques implémentations avec ces critères.

2.2.1 Test de sous-typage dans Hotspot

Le mécanisme de sous-typage de Hotspot est décrit dans [CR02]. Le test utilisé est le test de Cohen [Coh91] avec l'utilisation de caches.
Il est d'abord rappelé le fonctionnement de Java. Le test de sous-typage peut être :

- Explicite (*instanceof*, *checkcast*)
- Implicite (utilisation des tableaux)

En effet, les tableaux sont covariants et polymorphes en Java. Chaque fois qu'on ajoute un élément à un tableau un test de sous-typage est effectué pour savoir si son type est compatible avec le type de déclaration du tableau. Ce mécanisme entraîne énormément de tests de sous-typage, celui-ci est donc crucial pour les performances.

Hotspot utilise un test de sous-typage qui est très rapide dans des conditions favorables. Dans quelques rares cas, comme par exemple le test de sous-typage par rapport à une interface, le test n'est plus en temps constant. Les hiérarchies de classes sont stockées dans un tableau de taille fixe (8). Une liste secondaire est utilisée pour des hiérarchies plus longues. Au pire, une recherche linéaire est effectuée dans cette liste secondaire contenant les supertypes de la classe.
C'est pourquoi ils utilisent un système de caches pour éviter d'être trop souvent dans ces cas défavorables.

Pour résumer, ce test est :

- Compatible avec l'*inlining*, car court
- Compatible avec le chargement dynamique
- Espace linéaire
- Compatible avec le sous-typage multiple

Les cas les plus défavorables sont ceux impliquant des classes avec une grande hiérarchie.
Ils disent avoir choisi d'optimiser ces cas pour l'espace mémoire plutôt que le temps car cela reste rare. Une condition non remplie est donc le temps constant.
De plus, cette liste de taille 8 a été établie il y a longtemps. Les programmes actuels ont fréquemment une hiérarchie de classe plus grande, et donc, effectuent plus souvent des recherches linéaires dans la seconde liste.

2.2.2 Implémentation des interfaces dans Jikes RVM

Dans [ACFG01] est décrite l'implémentation des interfaces utilisée dans Jalapeño (ancêtre de JikesRVM).

Les interfaces avaient mauvaise réputation en Java car réputées lentes. En effet, avoir des interfaces revient à un contexte d'héritage multiple avec les problèmes que cela engendre. Les auteurs identifient trois sources de mauvaises performances des interfaces :

- Test de type dynamique à l'exécution
- Sélection de méthodes dans le sous-typage multiple
- Optimisations faibles du compilateur pour ces appels

Pour le test de sous-typage un tableau contient les identifiants des interfaces. Cette table est présente pour chaque classe. Étant donné que le test de sous-typage est assez lourd, un cache contient le résultat d'un test de sous-typage pour une paire donnée de classes.

Pour la sélection de méthodes ils utilisent une *Interface Method Table* (IMT) de taille fixe. C'est une sorte de table de méthode pour les interfaces. À l'intérieur sont hachées les signatures des méthodes des interfaces. Cette table de hachage est indexée depuis l'objet *class* de la classe en question.

L'article décrit aussi quelques optimisations pour améliorer les performances des interfaces. Ils utilisent dans Jalapeño la technique sur l'*inlining* des méthodes virtuelles décrite dans [DA99]. Le concept de *virtualisation* est aussi introduit. Il s'agit de transformer un appel de type $invokeinterface$ en un $invokevirtual$.

2.2.3 Hachage parfait

Le hachage parfait est une technique s'appuyant sur une table de hachages. Les identifiants des interfaces et des méthodes sont hachés. Cette technique garantit une table de hachage avec peu de collisions et un espace occupé qui reste faible.

Si on veut tester si la classe B est sous-type de I, on hache I. On regarde ensuite dans la table de hachage de B. Si l'identifiant de I est trouvé, le test renverra vrai, sinon faux. Le hachage parfait remplit les cinq exigences pour le test de sous-typage. Dans le cadre du **test de sous-typage**, le hachage parfait reste néanmoins deux fois plus lent que le test de Cohen. Si le receveur du test est une classe, ce dernier sera donc préféré. Par contre, pour les interfaces dans un contexte d'héritage multiple le hachage parfait s'applique. Les tables de hachages contiennent donc les identifiants des interfaces uniquement.

Pour un test dynamique, une fonction peut s'appliquer et exécuter le bon test selon que le receveur est une classe ou une interface.

Pour **l'appel de méthode** le hachage parfait est aussi utilisé. La table de hachage contient aussi les identifiants des méthodes introduites depuis des interfaces. Si le receveur est typé par une classe, l'appel de méthode est comme en héritage simple.
S'il est typé par une interface il n'y a plus l'invariant de position dans les tables de méthodes. Le hachage parfait s'applique dans ce cas là.

Cette table de hachage fait donc office de table de méthodes. Elle est placée aux indices négatifs de la table de méthodes de la classe implémentant cette interface. Il est donc possible d'utiliser le hachage parfait pour le test de sous-typage et pour l'appel de méthode.

Le hachage parfait est donc une solution efficace, qui remplit les cinq conditions. Elle demeure quand même coûteuse, et dans le cadre de Java il vaut mieux utiliser le test de Cohen quand le receveur est une classe. D'après [DM11], il faudrait plus de recherches avant d'utiliser exlusivement le hachage parfait pour les attributs et donc fournir une implémentation complète basée juste sur le hachage parfait.

2.3 Critères de comparaison des JVM

Type de compilation

Le principal discriminant technique des JVM est leur type de compilation : certaines interprètent le *bytecode*, d'autres le compilent et enfin certaines ont une approche mixte.
L'interpréteur a certains avantages : il est souvent plus léger et plus simple à écrire et moins dépendant de la plateforme cible. En contrepartie, il est beaucoup plus lent. Dans [GEK01], les auteurs développent un interpréteur efficace de *bytecode*. Leur travail est greffé sur la JVM Cacao. Ils utilisent un code intermédiaire pour simplifier les instructions *bytecode* et effectuer des optimisations. Les résultats montrent, dans le benchkmark le plus favorable, que l'interpréteur est deux fois plus lent que le compilateur JIT (certaines optimisations sont beaucoup plus efficaces en touchant directement au code machine). Pire encore, dans certains cas l'interpréteur est de 15 à 20 fois plus lent, ils concluent eux-mêmes que cette approche n'est pas possible pour les applications scientifiques effectuant beaucoup de calculs.

Langage d'écriture de la VM

Certaines VM sont écrites en Java. Elles sont qualifiées de méta-circulaires. Pour celles-ci la réalisation de l'amorçage, appelé *bootstrap*, n'a rien d'évident. Par exemple dans Jikes RVM [RZW08], le *bootstrap* est réalisé avec une image mémoire obtenue préalablement via l'introspection dans une autre VM. Cette image est ensuite rechargée en mémoire au lancement de la machine virtuelle. Généralement cette image mémoire est chargée via un petit programme en C [Mat08, RZW08].

D'autres VM sont écrites complètement en C/C++, c'est par exemple le cas de J3 et CacaoVM. Le langage utilisé pour la VM a des conséquences sur les implémentations possibles, c'est donc un critère important pour la suite de cette étude.

Optimisations

Il y a deux types approches pour les optimisations qui interviennent à des moments différents :
- optimiser certains aspects lors de la compilation en faisant une analyse de la hiérarchie de classe (CHA)
- effectuer du *profiling* pendant l'exécution, par exemple en comptant les appels de méthodes et en optimisant de manière poussée celles qui sont souvent appelées.

Complexité

Certaines JVM sont des produits commerciaux, donc par nature très complexes. D'autres sont destinées à la recherche donc en théorie plus facilement modifiables. Plusieurs critères entrent en jeu pour mesurer la complexité mais un aspect essentiel est la taille du code de la machine virtuelle. Par exemple travailler avec une machine virtuelle aussi complexe (et inaccessible) que Hotspot n'est pas réalisable. D'autres sont plus accessibles (documentation, articles) en permettant de changer certaines parties plus facilement. C'est donc une de ces dernières qu'il faudra utiliser lors de la mise en pratique.

2.4 Présentation de quelques JVM

Il y a tout d'abord un certain nombre de VM commerciales. Elles ne seront pas choisies à cause de leur trop grande complexité mais méritent d'être étudiées en termes d'implémentation.
D'autres sont destinées à la recherche :
- Jikes RVM : JVM d'IBM destinée à la recherche

– Maxine : JVM d'Oracle pour la recherche
– J3 : JVM intégrée à vmkit
– Cacao VM : développée par l'université de Vienne à l'origine, elle est devenue un projet libre
– Open Runtime Platform : développée par Intel pour la recherche

Bien sûr, cette liste n'est pas exhaustive. Il existe beaucoup d'implémentations de la JVM et toutes ne peuvent pas être passées en revue (Moxie JVM...). Par contre, il y a des catégories récurrentes. Par exemple, les JVM écrites en Java ou encore celles qui interprètent le *bytecode*. Cette étude est destinée à fournir un aperçu de quelques implémentations avec quelques grandes catégories des JVM représentées.

2.4.1 Cacao VM

Présentation

CacaoVM est à l'origine un projet de recherche développé par l'université de Vienne. Le projet a commencé en 1997, le but initial est de découvrir de nouvelles techniques d'implémentations de la JVM. À l'époque Hotspot faisait de l'interprétation uniquement. Cacao était donc particulièrement efficace par son approche uniquement basée sur la compilation. En 2004, le projet est passé sous licence libre et son développement continue.

Architecture globale

Cacao VM est une machine virtuelle écrite en C++. Elle utilise une approche compilée uniquement. La compilation est paresseuse c'est à dire que les méthodes sont compilées lorsqu'elles sont appelées. Ainsi une méthode jamais appelée ne sera jamais compilée.
Elle utilise un système pour générer du code natif en quatre étapes [KG97]. Ces quatre étapes sont :
– Détermination des blocs basiques
– Génération d'un code intermédiaire orienté-registre, chaque instruction intermédiaire $MOVE$ utilise un nouveau registre
– Allocation des registres
– Génération du code machine

Un autre aspect intéressant concerne la représentation des objets en mémoire : le *object-layout*. Les méthodes des interfaces sont placées aux indices négatifs par rapport au pointeur sur la classe de l'objet.

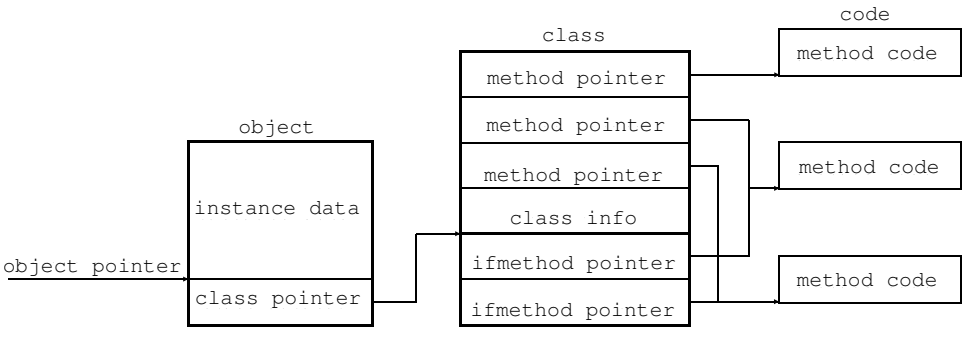

FIGURE 2 – Représentation mémoire des objets dans Cacao d'après [KG97]

Une amélioration de l'algorithme de génération de code machine est expliquée dans [Kra98], les performances sont bien meilleures qu'initialement, l'algorithme fonctionnant maintenant en trois étapes :

- Détermination des blocs basiques et génération d'instructions intermédiaires plus simples à traiter par la suite
- Analyse de la pile et génération d'une structure de pile statique
- Allocation de registres temporaires et génération du code machine

Test de sous-typage et sélection de méthode

La sélection de méthodes de Cacao utilise une sorte de table à accès direct. Le coup d'une instruction $invokeinterface$ est donc à peine plus élevé que celui de $invokevirtual$. Par contre, cette table contient de nombreux espaces vides car une classe implémente généralement assez peu d'interfaces. Le test de sous-typage de Cacao n'est pas décrit explicitement.

2.4.2 Hotspot

Présentation

Hotspot est la machine virtuelle officielle de Java, elle est déclinée dans une version client et serveur. Elle fonctionnait en mode interprété à l'origine. Hotspot utilise maintenant une approche mixte. Les méthodes souvent exécutées sont compilées pour plus d'efficacité. Elles sont qualifiées de "points chauds" (Hotspot) ce qui a donné le nom à la machine virtuelle. Elle n'est évidemment pas destinée à la recherche mais possède des propriétés intéressantes et sert d'étalon aux autres implémentations, notamment en ce qui concerne les optimisations.

Architecture générale

Hotspot est déclinée en une version client et une version serveur.
Globalement, la version client permet un temps de réponse rapide et un démarrage rapide. La version serveur quant à elle, n'a pas ces contraintes, elle est plus efficace.

Version client

La machine virtuelle fait deux types d'optimisations : de l'analyse statique avant la compilation et du *profiling* à l'exécution. Ces deux approches combinées permettent un gros gain de performances. Plusieurs optimisations différentes sont possibles une fois les informations récoltées. Plusieurs formes intermédiaires sont générées durant la compilation, elles permettent des optimisations spécifiques à chaque fois. Lors de la compilation, une forme intermédiaire du code nommée HIR (*High-level Intermediate Representation*) est produite. Cette forme permet plusieurs optimisations telles que l'élimination des $null - check$ [4]. Des analyses statiques du code sont aussi faites, CHA par exemple permet de détecter les sites d'appels monomorphes. Dans ces cas là, le compilateur effectue un *inlining* ou alors un appel statique selon la longueur du code. Ceci est aussi vrai pour les appel statiques et autres méthodes finales qui peuvent être directement inlinés. Une autre forme intermédiaire de plus bas niveau est aussi produite : LIR (*Low-level Intermediate Representation*). Cette forme sert par exemple à optimiser l'accès aux registres de la machine. Ces optimisations peuvent parfois devenir fausses avec le chargement d'une classe. Hotspot procède alors à une désoptimisation, à travers du *On-Stack Replacement* et un basculement en mode interprété. Lors de l'exécution le *profiling* entraîne les effets suivants :

4. Test permettant de savoir si le receveur n'est pas une valeur nulle avant d'appeler une méthode

– Si une méthode est souvent appelée alors elle sera compilée.
– Si une portion de code contient une boucle, elle peut aussi être compilée.

Version serveur

La version serveur est assez similaire à la version client.
Hotspot version serveur utilise aussi une approche mixte entre interprétation et compilation. Elle démarre plus lentement mais se montre plus performante à l'exécution. Le système utilise des optimisations adaptatives pour gagner du temps avec les portions de codes souvent exécutées.

Test de sous-typage et sélection de méthode

Le test de sous-typage est décrit dans [CR02], c'est un test de Cohen avec des caches, déjà décrit précédemment. L'implémentation de la sélection de méthodes n'est pas décrite explicitement.

2.4.3 Maxine

Présentation

Maxine [WHVDV+13] est une machine virtuelle développée par Sun dans un but de recherche.
C'est une VM développée en Java, donc métacirculaire. Elle est basée sur le JDK standard et couvre actuellement presque tout Java. D'après [WHVDV+13] la structure de Maxine s'approche par de nombreux points de celle de Jikes RVM. Ceci est vrai tant pour l'architecture que pour les représentations intermédiaires du code pour faire des optimisations.

Architecture générale

Étant donné qu'elle est destinée à la recherche les composants de la VM sont prévus pour être facilement changés. Les interfaces entre les composants de la VM sont clairement définies. Ils ont aussi développé un inspecteur permettant de voir l'état interne de la VM lors de l'exécution.
La machine est bootstrapée avec un substrat en C qui charge l'image en mémoire et le minimum pour permettre de démarrer. Deux compilateurs sont présents dans la machine virtuelle :
– T1X : compilateur de base
– C1X : compilateur optimisé
Le premier compilateur est forcément beaucoup plus rapide mais produit du code moins efficace.

Maxine réutilise les représentations intermédiaires présentes dans Hotspot. Leur machine virtuelle est disponible avec le JDK standard de Java et OpenJDK. Actuellement, la librairie standard de Java a quelques points d'interactions avec la machine virtuelle. Elle ne possède pas d'interface clairement définie pour ces points d'interactions et contient du code spécifique à Hotspot. Pour Maxine, ils ont donc dû modifier ces points d'interactions pour pouvoir utiliser OpenJDK et le JDK standard.
L'architecture de Maxine s'inspire d'autres JVM méta-circulaires, elle ressemble donc à JikesRVM sur de nombreux points. Pour le *bootstrap*, Maxine doit utiliser Hotspot car leur processus de génération de l'image de démarrage contient (encore) du code spécifique à Hotspot. Le *bootstrap* n'est pas encore total.

Il est précisé dans cet article que le projet est actif et que leur travaux futurs s'axeront sur les optimisations. Actuellement avec des benchmarks standard Maxine arrive à environ 67% des performances de Hotspot client pour 57% des performances de Hotspot serveur.

L'architecture de Maxine est décrite dans la Figure 3. Cela permet d'avoir une illustration de l'architecture d'une JVM métacirculaire.

JikesRVM sera par exemple très proche de Maxine bien que la plupart des JVM gardent ce même schéma global. Par exemple une JVM écrite en C n'aura pas besoin du même principe d'amorçage qu'une JVM écrite en Java.

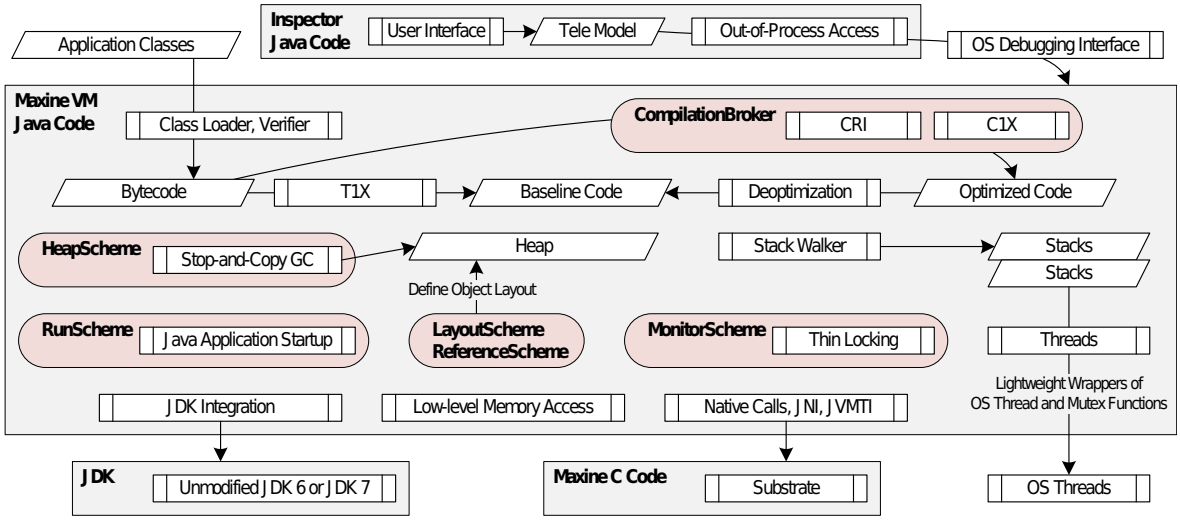

FIGURE 3 – Architecture interne de Maxine d'après [WD12].

Test de sous-typage et sélection de méthode

Maxine ne fournit pas d'informations précises sur son test de sous-typage. Mais puisqu'elle est développée par Oracle, on peut s'attendre à avoir les mêmes mécanismes qu'avec Hotspot. Il en va de même avec la sélection de méthodes.

2.4.4 Jikes RVM

Présentation

Jikes RVM est la continuation de Jalapeño [AAB+00], un projet de recherche d'IBM. À l'origine Jalapeño est un projet interne d'IBM mais le projet est passé en open-source et a été renommé en JikesRVM en 2001 [AAB+05]. C'est à l'origine une JVM spécifique pour les serveurs.

Architecture générale

Jalapeño répond aux besoins spécifiques des serveurs tels que : l'exploitation de processeurs puissants, la parallélisation ou encore la disponibilité pendant de longues durées. Cette machine virtuelle est optimisée pour une architecture précise qui était la cible initiale. Elle est écrite principalement en java sauf une toute petite partie servant à réaliser l'amorçage et les appels natifs.
Enfin, elle fonctionne en mode compilé uniquement. Trois compilateurs sont d'ailleurs implémentés :

- **baseline compiler**

– **optimizing compiler**
– **quick compiler**

Le premier est le compilateur de base qui est le premier qui a été écrit. Le compilateur optimisé utilise trois formes intermédiaires du code en réalisant à chaque fois des optimisations particulières. C'est évidemment le compilateur le plus lent. Enfin, le compilateur rapide produit du code peu optimisé mais de manière très rapide.

En 2001, Jalapeño devient JikesRVM.

Test de sous-typage et sélection de méthode

L'implémentation des interfaces est décrite dans [ACFG01]. Le test de sous-typage n'est pas décrit explicitement.

2.4.5 J3

Présentation

J3 [GTL+10] est la machine virtuelle Java intégrée à VMkit.
VMKit est un framework qui facilite la création de machines virtuelles. Il a été testé en fournissant la JVM J3 qui a été développée en parallèle de VMKIT. Elle est optimisée, et l'utilisation d'un framework permet d'avoir le coeur de la VM relativement petit.

Architecture générale

VMKit a des parties fixes de la VM tels que le *garbage collector* ou encore le compilateur JIT. Par contre, il n'impose pas le modèle objet, le système de types et surtout les appels de méthodes.
J3 contient donc le coeur de la machine virtuelle uniquement. J3 et VMKit fonctionnent avec les bibliothèques LLVM. Une partie de la machine virtuelle est donc destinée à faire la liaison vers LLVM [5]. Elle est également basée sur GNU Classpath et non pas sur openJDK. D'après les tests de performances effectués dans l'article, J3 a des performances similaires à Cacao. Elle demeure quand même plus lente que JikesRVM (facteur 1.5 à 3) et que Hotspot (facteur 1.5 à 5). Enfin, elle est composée d'environ 23000 lignes de code. Ceci est possible grâce à l'usage de composants externes définis dans VMKit et LLVM.

Test de sous-typage et sélection de méthode

J3 utilise le test de sous-typage de Hotspot [CR02] et l'implémentation des interfaces décrite dans [ACFG01].

2.4.6 Open Runtime Platform (ORP)

Présentation

C'est une JVM développée par Intel pour la recherche [C+03].
L'approche utilisée prône la réutilisation. En effet, leur but est de construire des interfaces entre les différents composants de la machine virtuelle. Cela permet donc de pouvoir facilement changer certaines parties et favorise donc les expérimentations.

5. LLVM contient son propre *bytecode*, et un compilateur pour produire du code machine à partir de ce code intermédiaire.

Architecture générale

Cette machine virtuelle fonctionne en mode compilé uniquement. Elle supporte Java et le CLI (langage intermédiaire de C#, analogue au *bytecode* de Java). Elle est divisée en trois composants principaux :

- Le coeur de la VM : chargement, compilation, informations pour la gestion mémoire, gestion des exceptions, threads et synchronisation
- Le compilateur JIT : compilation en instructions natives
- Le *garbage collector* : gestion du tas, allocation des objets et vidage de la mémoire

Dans le futur, ils indiquent vouloir créer un composant dédié à la gestion du multitâches et de la synchronisation. Les optimisations sont faites à la compilation et à l'exécution avec du *profiling*. La compilation paresseuse est aussi utilisée : lors du chargement d'une classe, l'espace est réservé pour le code de la méthode et elle est compilée à la première exécution.
Lors de la compilation une analyse statique du code avec CHA est faite. Cela permet d'enlever une partie des tests à *null*. Une autre partie de ces tests sont supprimés à l'aide du matériel, lors d'un test avec *null*, l'information est remontée jusqu'au coeur de la VM. Elle peut ensuite signaler une exception.
Une autre optimisation est l'accès aux tableaux, avec les tests de bornes. Dans certains cas le compilateur peut savoir si les bornes d'un tableau risquent d'être dépassées.
Il y a également de l'*inlining* et des appels statiques qui sont possibles grâce à CHA.

Toutes ces optimisations ont pour but de faire rapprocher les performances de cette machine virtuelle de produit commerciaux (Hotspot). Bien sûr les performances sont moins bonnes, mais l'idée était de prouver que le système de composants soit compatible avec de bonnes performances (2 fois plus lent dans le pire des cas).
Le découpage en composants impose par ailleurs des interfaces pour communiquer. Des sortes de structures propres sont donc partagées pour accéder aux objets en dehors du coeur de la VM. Le ramasse-miettes s'en sert par exemple pour gérer la mémoire.

Cette JVM est écrite en C++ avec un peu d'assembleur par endroit. Elle contient environ 150 000 lignes de code.

Test de sous-typage et sélection de méthode

Il n'est pas précisé quel est le test de sous-typage utilisé. Par contre, dans un but de gain de performances, ils effectuent des *inlining* du test de sous-typage. Leur test remplit donc au moins la condition d'être compatible avec l'*inlining*. L'implémentation des interfaces n'est pas décrite explicitement.

2.4.7 SableVM

Présentation

SableVM [GH01] est une machine virtuelle qui a été abstraite en un framework. Ce dernier est destiné à être étendu mais utilise plusieurs techniques d'optimisations pour une JVM en mode interprété. SableVM est écrite en C.

Architecture générale

C'est une VM en mode interprété car, d'après les auteurs, il y a deux principaux problèmes avec la compilation JIT :

- La génération de code à la volée est coûteuse et ce temps est perdu pour l'exécution

– Le code compilé est dans la mémoire ce qui rajoute du travail au *Garbage Collector*

La construction de la VM est assez classique, elle contient cinq composants :

– Interpréteur
– Gestion de la mémoire (*Garbage collector*)
– Chargeur de classe
– Vérificateur de *bytecode*
– Lien avec l'interface native

La Vm gère bien sûr la synchronisation et les threads en plus.

Un des avantages principaux de cette VM est sa gestion de la mémoire. En effet, le GC doit parcourir la mémoire pour trouver les objets à désallouer. C'est principalement cette opération qui est la plus coûteuse en temps. Ce ramasse-miettes est donc qualifié de **traçeur**, car il parcourt la mémoire. C'est la forme la plus souvent choisie pour implémenter les GC dans les VM.

Leur solution est de grouper les références dans les objets pour que le GC puisse tester leur existence en même temps qu'il parcourt la mémoire. L'invariant de position est donc conservé dans l'objet. Ils mettent en plus dans l'entête de l'objet le nombre de références.
Le premier apport majeur de l'article est la forme des objets en mémoire. Ils introduisent un *bidirectionnal object layout*. Celui-ci permet de grouper les références et les informations de synchronisation pour faciliter le travail du *Garbage Collector*.

Test de sous-typage et sélection de méthode

Un deuxième apport de l'article concerne l'implémentation des interfaces. L'implémentation usuelle est d'avoir deux tables de méthodes :

– La première pour l'invocation normale
– La deuxième pour l'invocation de méthodes présentent dans l'interface

De plus, une table de méthode spécifique est présente pour chaque interface qu'implémente la classe.
Ce fonctionnement implique donc à chaque appel de méthode de trouver la bonne interface et ensuite la bonne méthode. Il y a donc une recherche supplémentaire par rapport à un appel de méthode sur une classe.

Leur approche tire partie du comportement de Java avec les interfaces. En effet, si une classe implémente deux interfaces qui contiennent la même signature de méthode, une seule sera conservée lors de l'implémentation dans la table de méthode.

Ils choisissent donc d'attribuer un seul indice à ces deux méthodes. Les tables de méthodes des interfaces sont positionnées dans les indices négatifs par rapport à la table de méthode de la classe. Ce positionnement est appelé *bidirectionnal layout*. Le coût de l'instruction $invokeinterface$ revient au même que $invokevirtual$. Ils utilisent donc des tables à accès direct, l'appel est rapide. Ceci forme une sorte de grande matrice avec les classes et les méthodes, l'accès est donc presque immédiat. En contrepartie il y de nombreux "trous" dans cette table. Leur solution (ingénieuse et téméraire à la fois), pour éviter de gaspiller trop d'espace est d'allouer des objets dans ces trous.

2.5 Comparaison des JVM

Dans cette section seront comparées les différentes machines virtuelles. Le but est d'avoir une vue globale qui permettra d'effectuer un choix sur celle à choisir pour la pratique.

2.5.1 Tableau comparatif

Nom	Langage	Lignes	Dernière MAJ	JDK	Compilation
Cacao	C++	230 000	4 septembre 2012	GNU Classpath/OpenJDK	Compilation
Hotspot	C/C++	250 000	×	OpenJDK	Mixte
J3	C++	23 000	février 2013	GNU Classpath	Compilation
JikesRVM	Java	275 000	12 février 2013	GNU Classpath	Compilation
Maxine	Java	550 000	janvier 2012	OpenJDK	Compilation
ORP	C++	150 000	2009	GNU Classpath	Compilation
SableVM	C	×	2007	GNU Classpath	Interprétation

FIGURE 4 – Tableau comparatif des JVM

2.5.2 Performances

La performance des JVM n'est pas forcément essentielle pour un but de recherche puisque il s'agit de comparer. Cela permet toutefois d'avoir une idée. En particulier, la différence entre les machines virtuelles de recherches et les commerciales apparaît clairement.

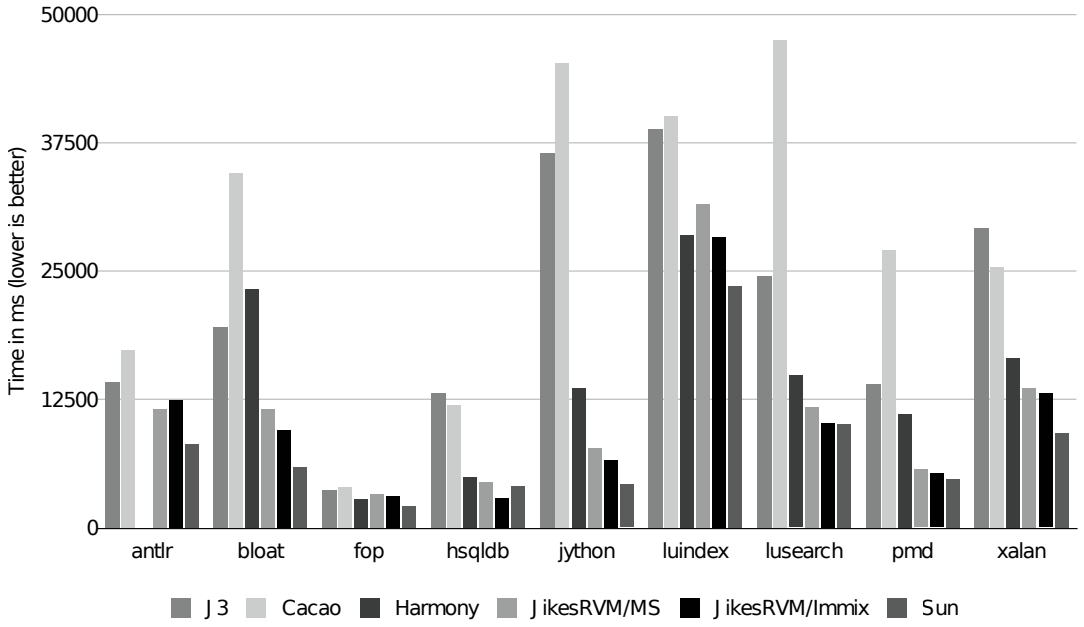

FIGURE 5 – Performances de quelques JVM d'après [GTL+10].

3 Intégration du hachage parfait aux JVM

3.1 Faisabilité

Le hachage parfait sera donc utilisé avec les contraintes énoncées plus haut. L'importance du hachage parfait pour les interfaces est cruciale. En Java, les interfaces sont maintenant souvent utilisées alors qu'elles avaient mauvaises réputation avant. De plus, des langages comme SCALA utilisent aussi beaucoup l'opération *bytecode invokeinterface*. Son intégration demande quelques prérequis dans la représentation des tables de méthodes de la VM [DM11] :

– Table bidirectionnelle
– Identifiants des interfaces groupés

Les tables de hachages étant de tailles variables, il est exclu de les placer dans les tables de méthodes. Une solution est donc de les positionner dans les indices négatifs des tables de méthodes. Il faut donc des tables de méthodes contiguës et qui puissent avoir des indices négatifs. Bien sûr, il est possible de simuler ces prérequis mais cela entraîne des indirections supplémentaires et donc un coût plus élevé.
Le schéma général de l'implémentation voulue est le suivant :

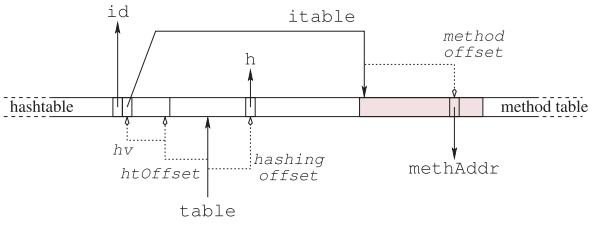

FIGURE 6 – Hachage parfait des interfaces pour Java d'après [DM11].

3.2 Choix de la JVM

Pour la partie pratique, il faut travailler sur une machine respectant ces prérequis pour le hachage parfait. Celles fonctionnant en mode interprété sont à exclure car assez anciennes et peu performantes. De plus, les machines virtuelles commerciales ne sont pas adaptées à des buts de recherches.

Un autre critère discriminant sera la dernière mise à jour. Travailler sur une machine virtuelle trop ancienne n'aurait pas de sens car les résultats seraient irréalistes.

Les représentations bidirectionnelles des tables de méthodes sont plutôt possibles dans des langages assez permissifs tels que C. De plus, [DM11] indique un début d'implémentation sur JikesRVM qui a été abandonné. Le gain de performance du hachage parfait aurait été éclipsé par la simulation de ces conditions requises pour le hachage parfait. Les JVM métacirculaires sont donc à exclure car le surcoût des tables de méthodes bidirectionnelles fausseraient les résultats.

Avec ces différents critères il est déjà possible d'éliminer SableVM (trop ancienne) ainsi que Open Runtime Platform pour la même raison. Cacao présente des caractéristiques intéressantes puisqu'elle est à l'origine destinée à la recherche. Elle utilise déjà une représentation bidirectionnelle. Par contre elle souffre de la comparaison avec J3. En effet, cette dernière se montre plus performante en plus d'avoir le coeur de son fonctionnement plus accessible.

Maxine et JikesRVM présentent l'avantage du langage de plus haut niveau ainsi que de nombreuses publications. Maxine a à son crédit l'inspecteur qui permet de visualiser son état pendant l'exécution.

Mais ces deux machines sont méta-circulaires et poseraient donc des problèmes pour implémenter le hachage parfait. J3 quant à elle a l'énorme avantage d'avoir les aspects mémoire, analyse lexicale et compilation en dehors du coeur de la machine virtuelle. Le coeur de la VM est composé de seulement 23000 lignes, ce qui fait beaucoup moins que ses concurrentes qui définissent le gestionnaire de mémoire et le compilateur en plus. J3 devrait donc être plus facilement compréhensible dans sa globalité. Il est bien précisé dans [GTL+10], que le modèle mémoire et le fonctionnement des appels de méthodes n'est pas décrit dans VMKit. Ces aspects là sont donc détachés de VMKit, ce qui indique qu'ils sont bien définis dans J3 en elle-même. J3 semble donc facile d'accès et réunit les conditions pour l'intégration du hachage parfait.

4 Conclusion

Au fil des années, les langages s'appuyant sur des machines virtuelles (C#, Java) se sont développés. La machine virtuelle et le chargement dynamique sont aujourd'hui des concepts très largement répandus. L'implémentation des machines virtuelles Java est presque toujours basée sur la compilation à la volée pour des raisons de performances. De plus, l'utilisation des interfaces est devenue très courante. Des réponses aux problèmes de performances de Java ont été apportées. En particulier l'implémentation des interfaces est aujourd'hui plus efficace qu'auparavant. Mais ces réponses ne sont pas optimales, en particulier, elles ne permettent pas de satisfaire les cinq exigences pour le test de sous-typage. La sélection de méthodes n'est pas réalisée de manière optimale non plus pour les méthodes introduites par une interface. Une solution serait d'utiliser le hachage parfait pour le test de sous-typage et la sélection de méthode pour des objets typés par des interfaces.
La suite de ce travail consistera à utiliser une machine virtuelle Java : J3, pour implémenter le hachage parfait et faire des tests de performances. Les machines virtuelles à sous-typage multiples étant largement utilisées, le hachage parfait est une solution pour gagner en performances. Il permet d'avoir une implémentation élégante et efficace tout en étant compatible avec des caractéristiques essentielles (héritage multiple, chargement dynamique). Ces caractéristiques sont probablement celles qui constitueront la plateforme idéale des prochaines années.

Références

[AAB+00] Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, J-D Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, et al. The jalapeno virtual machine. *IBM Systems Journal*, 39(1) :211–238, 2000.

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

[ACFG01] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of java interfaces : Invokeinterface considered harmless. *SIGPLAN Not.*, 36(11) :108–124, October 2001.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2) :97–113, 2003.

[C+03] Michal Cierniak et al. The open runtime platform : A flexible high-performance managed runtime environment. In *Intel Technology Journal*. Citeseer, 2003.

[Coh91] Norman H Cohen. Type-extension type test can be performed in constant time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4) :626–629, 1991.

[CR02] Cliff Click and John Rose. Fast subtype checking in the hotspot jvm. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pages 96–107, New York, NY, USA, 2002. ACM.

[DA99] David Detlefs and Ole Agesen. Inlining of virtual methods. *ECOOP'99—Object-Oriented Programming*, pages 668–668, 1999.

[DM11] Roland Ducournau and Floréal Morandat. Perfect class hashing and numbering for object-oriented implementation. *Softw. Pract. Exper.*, 41(6) :661–694, May 2011.

[Duc08] Roland Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6) :33, 2008.

[GEK01] D. Gregg, M. Ertl, and A. Krall. Implementing an efficient java interpreter. In *High-Performance Computing and Networking*, pages 613–620. Springer, 2001.

[GH01] Etienne M Gagnon and Laurie J Hendren. Sablevm : A research framework for the efficient execution of java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, volume 1, pages 27–39, 2001.

[GJS+12] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java language specification, java se, 2012.

[Gou01] K John Gough. Stacking them up : a comparison of virtual machines. *Aust. Comput. Sci. Commun.*, 23(4) :55–61, January 2001.

[GTL+10] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. Vmkit : a substrate for managed runtime environments. *SIGPLAN Not.*, 45(7) :51–62, March 2010.

[KG97] A. Krall and R. Grafl. Cacao - a 64-bit javavm just-in-time compiler. *Concurrency Practice and Experience*, 9(11) :1017–1030, 1997.

[Kra98] A. Krall. Efficient javavm just-in-time compilation. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 205 –212, oct 1998.

[KS00] Kenneth B Kent and Micaela Serra. Hardware/software co-design of a java virtual machine. In *Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on*, pages 66–71. IEEE, 2000.

[KWM+08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot™client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1) :7 :1–7 :32, May 2008.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. *SIGPLAN Not.*, 33(10) :36–44, October 1998.

[LYBB12] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java virtual machine specification : Java se 7 edition, 2012.

[Mat08] Bernd Mathiske. The maxine virtual machine and inspector. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 739–740, New York, NY, USA, 2008. ACM.

[Ode09] Martin Odersky. The scala language specification, version 2.8. *EPFL Lausanne, Switzerland*, 2009.

[OT97] J Michael O'connor and Marc Tremblay. picojava-i : The java virtual machine in hardware. *Micro, IEEE*, 17(2) :45–53, 1997.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot™server compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–1. USENIX Association, 2001.

[RZW08] I. Rogers, J. Zhao, and I. Watson. Boot image layout for jikes rvm. *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'08), Paphos, Cyprus. July*, 8, 2008.

[Sin03] J. Singer. Jvm versus clr : a comparative study. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 167–169. Computer Science Press, Inc., 2003.

[WD12] Christian Wimmer and Laurent Daynès. Maxine : A virtual machine for, and in, java. Technical report, Oracle, 2012.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine : An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4) :30 :1–30 :24, January 2013.