Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier
LIRMM

UNIVERSITÉ MONTPELLIER 2
SCIENCES ET TECHNIQUES

Académie de Montpellier
Sciences et Techniques du Languedoc

# Mémoire de stage 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**

**Date de soutenance : 9 juillet 2013**

Sous la direction de **Roland DUCOURNAU**

# Table des matières

## Remerciements

Je tiens tout d'abord à remercier mon tuteur de stage *Roland Ducournau* pour sa présence et ses conseils avisés tout au long du stage. Je tiens également à remercier trois autres stagiaires qui travaillaient sur des sujets très proches : *Baptiste Saleil*, *Rabah Laouadi* et *Walid Benghabrit*. Merci également à un autre stagiaire de l'équipe : *Adel Ferdjoukh* pour les longues discussions et débats. Merci à eux pour l'entraide, le partage et l'ambiance pendant le stage. Merci également à *Floréal Morandat* pour ses conseils. Merci à Pauline pour son soutien et sa présence.

# 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 utilisent en général un système d'exécution basé sur la compilation (C++, Java, C#) d'autres fonctionnent avec un système basé sur l'interprétation (Python, Ruby, CLOS). Ces deux catégories ne sont pas figées, il est toujours possible de compiler du CLOS [1] mais le cas général est plutôt l'interprétation. 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. CLOS signifie *Common Lisp Object System*, forme objet du Lisp

## 1.1 Qu'est ce qu'une Machine Virtuelle ?

Une machine virtuelle (abrégée VM dans la suite) est un concept qui est apparu 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 qui utilise un mélange des deux approches [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*). Le monde ouvert encourage la réutilisabilité, c'est à dire qu'une classe est conçue indépendamment de ses usages futurs (en termes d'héritage par exemple). Mais ces avantages ont un coût, le monde fermé de C++ permet intrinsèquement de meilleures performances. En effet, la connaissance lors de la compilation de l'ensemble du programme permet un niveau d'optimisation plus élevé. L'idée de la compilation JIT a des origines assez lointaines, d'après [Ayc03] le premier article évoquant l'idée concerne le LISP, il est écrit par McCarthy en 1960. Un autre article (en 1966) énonce explicitement l'idée du chargement dynamique pendant l'exécution. Ces idées se sont ensuite 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 évolué 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 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 complexifié, 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 effectue les autres tâches. D'après les auteurs, les optimisations en parallèle pour des langages tels que C et Java 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. Les opérations constantes, arithmétiques, manipulation de piles en passant par les chargements/saut/comparaisons sont traités du côté matériel, 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'essouffle aujourd'hui bien que des recherches continuent.

Le rapport est structuré de la manière suivante : la première section un état de l'art des techniques existantes en présentant des concepts communs à toutes les machines virtuelles et quelques techniques d'implémentation du test de sous-typage et de la sélection de méthode.

Ensuite nous présentons plusieurs implémentations concrètes de la JVM. Puis nous comparons ces implémentations selon plusieurs critères.

Nous présentons ensuite la faisabilité de l'intégration du hachage parfait à une JVM et nous choisissons une machine virtuelle pour le stage.

La partie sur le stage présente essentiellement le déroulement de l'implémentation du hachage parfait ainsi que les résultats, le bilan et les perspectives. Enfin, nous présentons l'étude de l'adaptation d'un langage à objets à typage statique et à héritage multiple sur la JVM.

# 2 État de l'art des techniques existantes

La section suivante décrit l'architecture générale d'une machine virtuelle Java. Sont également présentées quelques techniques d'implémentations du test de sous-typage et de l'appel de méthode.

## 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 problème de transformation, 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, mais quelques composants sont récurrents tels que : le compilateur, le *Garbage Collector*, le chargeur de classes ou encore l'analyseur syntaxique et lexical. Ces composants et leur relation sont illustrés dans la figure 10 dans le cas d'une machine virtuelle précise.

Le fonctionnement général de Java est le suivant :

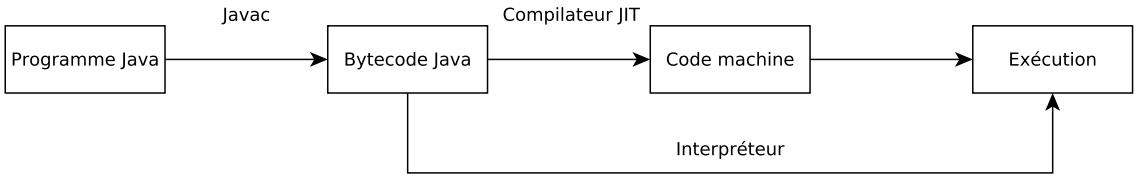


FIGURE 1 – Fonctionnement général de Java.

Un programme écrit en Java est compilé en plusieurs fichiers *.class*. Chaque fichier contient une seule classe (ou interface). Ce fichier est constitué d'une suite d'instructions dans un langage intermédiaire appelé le *bytecode* Java. Ce code intermédiaire est chargé par la machine virtuelle qui peut le recompiler en code natif (assembleur) à la volée. Elle peut également interpréter directement le *bytecode* mais généralement l'interprétation est plus lente que la compilation. Dans cette étude, nous nous intéresserons plus particulièrement aux JVM qui utilisent uniquement la compilation.

### 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 invoquée statiquement (méthodes privées, constructeurs, appel à *super() invokespecial*, invocation statique avec un receveur
- Méthode virtuelle *invokevirtual* introduite dans une classe
- Méthode d'une interface *invokeinterface*, implique de trouver la bonne implémentation de la méthode

Une autre instruction *bytecode invokedynamic* n'est pas utilisée pour Java mais est présente 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];
}
```

Les fichiers de *bytecode* Java contiennent des informations générales sur la classe contenue ainsi que le code des méthodes. Le *bytecode* utilise des références pour accéder aux différents éléments. Ces références sont dans le **constant pool**.

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 éléments (méthodes, noms des variables). Manipuler des entiers est plus efficace que des chaînes de caractères. De plus, ce système permet une résolution paresseuse de certains éléments qui peuvent être chargés uniquement quand c'est nécessaire. C'est pourquoi la JVM utilise un tel système, qui est par ailleurs présent dans une forme proche dans la machine virtuelle .NET.

Après le *constant pool* le fichier de *bytecode* contient le code des différentes méthodes. Voici le code compilé de la méthode *main* de l'exemple précédent 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)
13: astore_1
14: aload_1
15: invokevirtual #6  // Method Personne.afficher:()V
18: return
```

Ici une instance de *Personne* est créée avec les valeurs suivantes : nom = Dupond, prénom = Jean, âge = 25. Ce code permet d'observer deux types d'appels de méthode 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, ceci entraîne un surcoût mais garantit un typage plus sûr. Le composant de la JVM qui gère le chargement des classes à l'exécution est appelé *Class Loader*. Ce composant prend en entrée un fichier *.class* et retourne l'objet *Class* correspondant.

En plus d'être paresseux, le chargement entraîne un test pour s'assurer que le type est définissable et provoque une erreur s'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 (celles 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 s'agit d'un programme parcourant la mémoire et libérant les objets qui ne sont plus référencés. Cela permet au programmeur de ne pas gérer les aspects mémoires mais entraîne un surcoût à l'exécution.

La machine virtuelle Hotspot utilise un système par générations. Les objets sont déplacés dans des segments mémoires correspondant à leur âge, c'est à dire le nombre de cycles de collectage depuis leur création. Ce système est souvent repris par d'autres machines virtuelles pour ses bonnes performances. Dans Hotspot, il y a trois segments mémoires différents [KWM+08] :

– Young generation : objets venant d'être alloués
– Old generation : objets alloués depuis longtemps
– Permanent generation : structures internes

Ce fonctionnement par générations est appelé *ramasse-miettes générationnel*. 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 généralement en trois étapes ( [C+03]) : la première étape consiste à 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. Ce système par générations est assez courant dans les machines virtuelles, bien que plusieurs autres types de *Garbage collector* existent. 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 *bytecode* Java, il est déjà possible d'optimiser l'exécution future du programme. Une analyse de la hiérarchie de classes (*Class Hierarchy Analysis*, abrégée en *CHA*) permet par exemple de détecter certains sites d'appels monomorphes.

**Définition 1.** *Site d'appel monomorphe* : pour un site d'appel, une seule méthode est éligible pour l'appel de méthode

Lors de la détection d'un tel site d'appel, il y a deux possibilités d'optimisations :
– Si la méthode est courte elle peut être inlinée
– Si la méthode est trop longue, un appel statique peut être fait avec l'instruction *bytecode* *invokespecial*.

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, il est possible d'effectuer un *inlining* 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 a priori. Mais la plupart utilisent les deux approches combinées.

L'article [vdB06] détaille quelques optimisations utilisées dans la JVM Hotspot. Certaines sont des techniques d'optimisations qui sont présentes également dans la compilation classique. Par exemple les optimisations suivantes relèvent de la compilation classique (des langages pas forcément objets) :

- Inlining : enlever le coût d'un appel de méthode
- Propagation de constantes : remplacement des variables par leurs valeurs
- Suppression du code mort : élimination du code jamais exécuté
- Sortir des boucles des calculs récurrents
- *Peephole optimizations* : optimisation de l'ordre des instructions générées

D'autres optimisations ont aussi pour objectifs d'améliorer le nombre des optimisations précédentes. Dans les JVM, la production de code natif n'est pas faite en une seule étape. Toutes les optimisations ne sont pas faites dans la même phase de production de code.

Des optimisations typiques aux JVM (ou autres systèmes similaires) sont aussi réalisées. Par exemple, lors d'un appel de méthode, la JVM s'assure que le receveur n'est pas nul. Mais cette vérification doit être réalisée perpétuellement et dans la mesure du possible elle est supprimée. Entre deux affectations de la valeur *NULL*, l'objet a normalement une autre valeur, il est donc possible de supprimer tous les tests entre ces deux affectations. Pour des questions de sûreté du code, les JVM doivent aussi tester les bornes des tableaux pour ne pas dépasser.

Certaines JVM adoptent des systèmes hybrides faits d'analyse pré-exécution et de profilage pendant l'exécution. C'est la voie suivie par Hotspot et Jikes notamment. Ces systèmes sont qualifiés d'adaptatifs car ils s'adaptent en fonction et au cours de l'exécution du programme.

C'est par exemple le cas du système adaptatif dans la JVM Jikes. Les travaux de l'équipe sont assez poussés dans ce domaine précis. Le système est rapidement décrit dans la présentation de cette machine virtuelle.

### Inlining et désoptimisation

**Définition 2.** *Inlining* : le code de la méthode appelée est placé à l'endroit de l'appel.

De manière générale, une analyse statique du code permet de détecter un certain nombre de sites monomorphes. Bien entendu, les appels statiques et les classes méthodes d'une classe finale sont éligibles à l'*inlining*. Il est aussi possible d'inliner les sites provisoirement monomorphes voire inliner un site qui n'est pas monomorphe mais qui se révèle l'être lors de l'exécution.

Ce type de décision entraîne un appel statique ou un *inlining*. Par contre, il est possible que l'optimisation réalisée rentre en conflit avec le chargement dynamique. En particulier, si un appel statique ou inlining est réalisé (cette méthode est considérée finale donc non-redéfinie), le chargement ultérieur d'une classe peut invalider cette hypothèse. Il faut donc annuler cette optimisation et pouvoir revenir au code précédent. Cette opération est délicate, surtout si elle doit être faite pendant l'exécution de la méthode en question dont l'optimisation devient invalide. C'est possible si cette méthode entraîne le chargement d'une sous-classe qui redéfinie cette même méthode. Ce procédé est appelé désoptimisation.

Dans le cas de Hotspot, l'optimisation est défaite en repassant en mode interprété lors d'une désoptimisation. De manière générale, ce procédé entraîne souvent l'utilisation du *On-Stack Replacement* dans les JVM. La méthode en cours d'exécution est invalidée, il faut donc modifier les valeurs de retour et son code dans la pile d'exécution. Cela implique d'arrêter l'exécution pendant ce remplacement et de garder en mémoire le code avant l'optimisation. Ainsi, le code continue à être correct malgré l'optimisation effectuée précédemment.

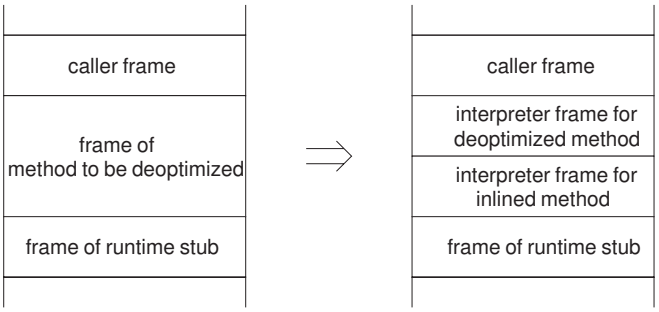


FIGURE 2 – Schéma du *On-Stack Replacement* dans Hotspot d'après [KWM+08]

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 très complexe.

## 2.2 Méta-modèle et définitions

L'article [DP11] propose une sémantique de l'héritage multiple basée sur la méta-modélisation. Ce modèle permet que chaque nom dans le code du programme ne représente qu'une et une seule instance du méta-modèle. Ce méta-modèle permet une sémantique propre en plus de fournir du vocabulaire efficace pour décrire l'implémentation des langages à objets.

Le méta-modèle proposé possède trois entités :
- Classe
- Propriété globale
- Propriété locale

Une classe possède un certain nombre de méthodes. Parmi celles-là, il y a des méthodes qui ne sont pas redéfinies et d'autres qui le sont. Une propriété qui n'est pas encore définie est dite introduite par la classe.

Un propriété globale représente un ensemble d'une même méthode. Une même propriété globale regroupera différentes propriétés locales. C'est à dire plusieurs redéfinitions de la même méthode globale.

Une propriété locale appartient à une classe donnée et correspond à une propriété globale. Ce méta-modèle est illustré dans la figure suivante :

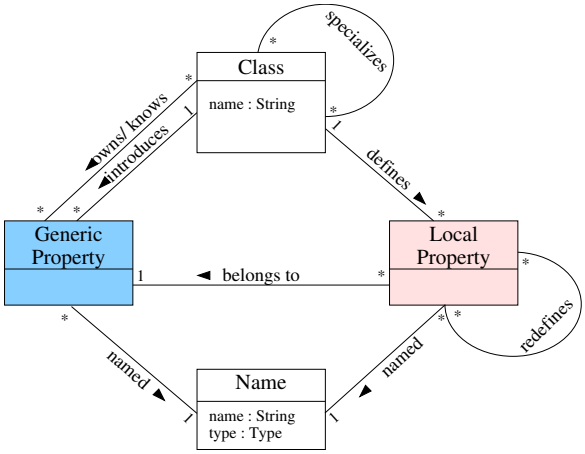


FIGURE 3 – Méta-modèle des classes et propriétés d'après [DP11]

Dans le cadre de Java et de ses interfaces, les signatures de méthodes déclarées dans les interface Java sont dites introduites par l'interface. Il s'agit donc de l'introduction d'une propriété. La méthode correspond donc à une propriété globale. L'implémentation concrète de cette méthode dans les sous-classes correspond à une propriété locale dans chaque classe où elle est redéfinie.

Dans les langages à objets, le type dynamique du receveur d'un appel de méthode n'est connu qu'à l'exécution. Il s'agit de sélectionner la propriété locale d'une propriété globale appelée. Cette propriété locale est connue par le type dynamique du receveur.

Les propriétés globales et locales représentent aussi bien des attributs que des méthodes. Dans le cadre de cette étude nous nous focalisons essentiellement sur les méthodes mais ce méta-modèle est valable pour les attributs.

## 2.3 Test de sous-typage et sélection de méthode

La programmation par objets introduit la notion de type et donc de sous-typage. En particulier les langages à objets sont très dépendants d'une implémentation efficace de trois mécanismes de base :

- Accès aux attributs
- Appel de méthode

– Test de sous-typage

Pour l'appel de méthode, 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, l'implémentation est grandement facilitée.

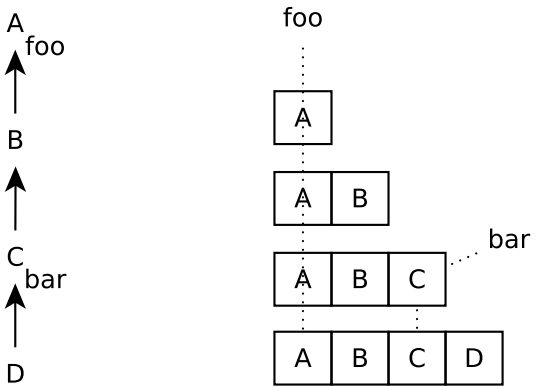


FIGURE 4 – Implémentation de l'héritage simple dans les langages à objets.

Pour construire les tables de méthodes, il suffit de concaténer les méthodes introduites dans chaque classe. Chaque méthode a donc une position unique dans la table de méthodes quel que soit la classe. On considère donc *l'invariant de position* pour les méthodes. Grâce à cet invariant, la sélection de la méthode est facilement réalisée car il suffit de connaître sa position. En cas de redéfinition dans une sous classe de la méthode, on lui attribue la position de la méthode qu'elle redéfinie.

Pour un langage en héritage multiple, cette position invariante n'existe plus.

**Définition 3.** *L'invariant de position* signifie que chaque méthode a une position dans la table de méthodes invariante par spécialisation.

**Définition 4.** *L'invariant de référence* signifie que les références à un objet sont indépendantes du type statique de la référence.

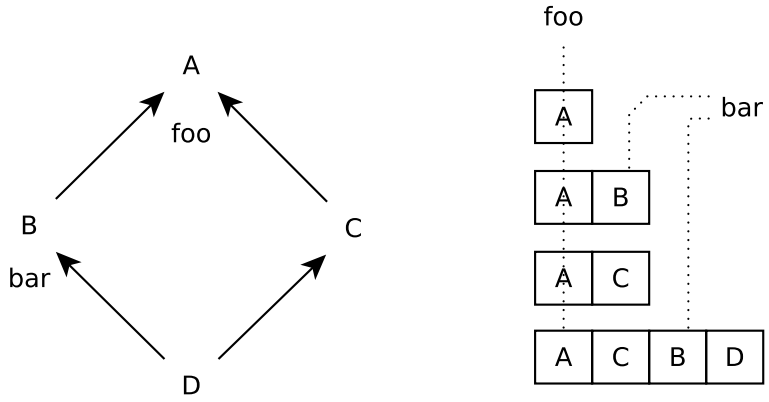


FIGURE 5 – Implémentation de l'héritage multiple dans les langages à objets.

Dans le schéma ci-dessus on peut observer un diagramme d'héritage en losange. L'implémentation des tables de méthodes par concaténation des méthodes introduites par chaque classe pose problème. En effet, dans $D$, on ne sait pas si on doit positionner les méthodes de $B$ avant celles de $C$ ou l'inverse. De manière générale les méthodes héritées de $B$ et de $C$ n'auront pas de positions invariantes dans $D$. Il est possible d'utiliser la coloration pour résoudre ce problème. En laissant des espaces vides dans les tables de méthodes les blocs de méthodes peuvent être à la même position invariante. Ce problème nécessite un mécanisme pour connaître la position des méthodes, qui ne peut plus être déduite simplement comme en héritage simple.

Dans le contexte de Java, ces trois mécanismes cruciaux sont implémentés selon différentes techniques qui sont perfectibles. En typage statique ces mécanismes sont généralement difficiles à implémenter, et l'efficacité obtenue est inférieure à celle d'un langage fonctionnant en monde fermé. Le problème est également complexifié par l'héritage multiple. En typage statique, 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.

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.

L'objectif est d'avoir un mécanisme 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. De plus, le temps constant garantit un aspect prédictible dans l'implémentation. L'espace linéaire n'est pas vraiment possible au sens strict du terme étant donné que certaines données des super-classes sont recopiées dans les sous-classes. L'objectif est d'avoir un espace linéaire dans la taille de la relation de spécialisation. 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.

La problématique du test de sous-typage est surtout à explorer en héritage multiple. En sous-typage simple, le test de Cohen [Coh91] est l'implémentation usuelle. Il permet d'avoir un test efficace en temps et espace constant. Par contre, il ne fonctionne pas en héritage multiple et donc en sous-typage multiple. Il s'agit d'une méthode plus efficace qu'une recherche linéaire dans le tableau. Chaque classe se voit attribuer une position fixe (sa profondeur dans la hiérarchie depuis la racine) dans un tableau de super-types. Lors d'un test de sous-typage il suffit de tester si la case à l'indice indiqué contient l'identifiant de la classe pour connaître le résultat du test. Néanmoins ce test n'est pas compatible avec l'héritage multiple.

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 la nécessité de l'héritage multiple 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. Ada ne fonctionnait qu'en héritage simple, les interfaces ont été rajoutées dans sa version de 2005.

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 ce qui est extrêmement coûteux.

Enfin, l'*inlining* est une optimisation très largement utilisée aujourd'hui. Le test de sous-typage car il est très souvent utilisé, doit contenir peu d'instructions pour pouvoir être inliné. Dans cette section seront comparées différentes implémentations du test de sous-typage et de la sélection de méthodes.

### 2.3.1 Test de sous-typage dans Hotspot

Le mécanisme de sous-typage de Hotspot est décrit dans [CR02]. Le test utilisé reprend le test le test de Cohen avec l'utilisation de caches et propose une adaptation au sous-typage muliple. Ce test est implémenté dans la machine virtuelle de référence en Java, il est souvent utilisé dans les comparatifs bien qu'il ne soit pas parfait. Dans l'article, 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. Un tableau de type *Object* pourra donc recevoir n'importe quel élément à l'intérieur car *Object* est le sommet de la hiérarchie de classes. L'ajout d'un élément dans un tableau entraîne donc toujours un test de sous-typage. Il faut s'assurer que l'élément ajouté est bien un sous-type du type de déclaration. Ce mécanisme entraîne énormément de tests de sous-typage, celui-ci est donc crucial pour les performances.

**Ancien test de sous-typage**

Avant cette implémentation le test dans Hotspot était implémenté par un système de caches contenant deux éléments. Lors d'un test de sous-typage : S $\prec$ T ?. Le système cherche à déterminer si T est présent dans le cache de S. S'il y est, le test est positif. S'il n'y est pas, un appel au cœur de la VM est effectué pour connaître le résultat qui est ensuite positionné dans le cache. Ainsi, un test qui échoue fera toujours appel à la VM pour avoir un résultat. De plus, les programmes testant souvent des types par rapport à trois autres types par exemples seront particulièrement lents à cause du cache à seulement deux entrées.

**Nouveau test proposé par [CR02]**

Ce test propose un traitement différent pour un test de sous-typage par rapport à une cible qui est une interface ou une classe.

Ces deux catégories pour le test sont définies dans l'article :

1. **primary type** : Classe propre, tableau, type primitif ou tableau de types primitifs
2. **secondary type** : Interface ou tableau d'interfaces

Ainsi, chaque type est soit primaire, soit secondaire avec cette définition. Pour la première catégorie, le test de sous-typage utilisé est un test de Cohen. Un tableau est alloué contenant tous les super-types primaires d'une classe, il est trié du type le plus élevé dans la hiérarchie jusqu'à la classe courante. Ce tableau de super-types est appelé *display*.

Ainsi, si on veut tester si S $\prec$ T. La position de T dans $S.display$ dépend de la profondeur dans la hiérarchie de types de T (appelée *depth*). Le résultat est immédiat, si T n'est pas à la place prévue le test renvoie faux.

Mais, d'après les auteurs cette première version du test implique de faire un test des bornes du tableau, ce qui est assez coûteux. Ils introduisent donc un *display* de taille fixe. La taille est choisie en se basant sur les benchmarks specJdb 98. Ils observent que le *display* moyen ne dépasse pas 5. Ils choisissent donc arbitrairement la taille 8 pour avoir un système moins périssable. Bien entendu, une telle limite est fortement discutable. La taille de la hiérarchie des classes aujourd'hui dépasse souvent ce chiffre et donc ralentit le système. La solution serait donc de changer la limite jusqu'à qu'elle soit à nouveau obsolète.

Avec ces limites de test de bornes du tableau, ils introduisent deux autres définitions :

1. **restricted primary type** : type primaire si la taille de son *display* n'excède pas 8
2. **restricted secondary type** : type secondaire (interface), ou type primaire avec une profondeur dans la hiérarchie supérieure à 8

Voici le schéma de ce test de sous-typage :

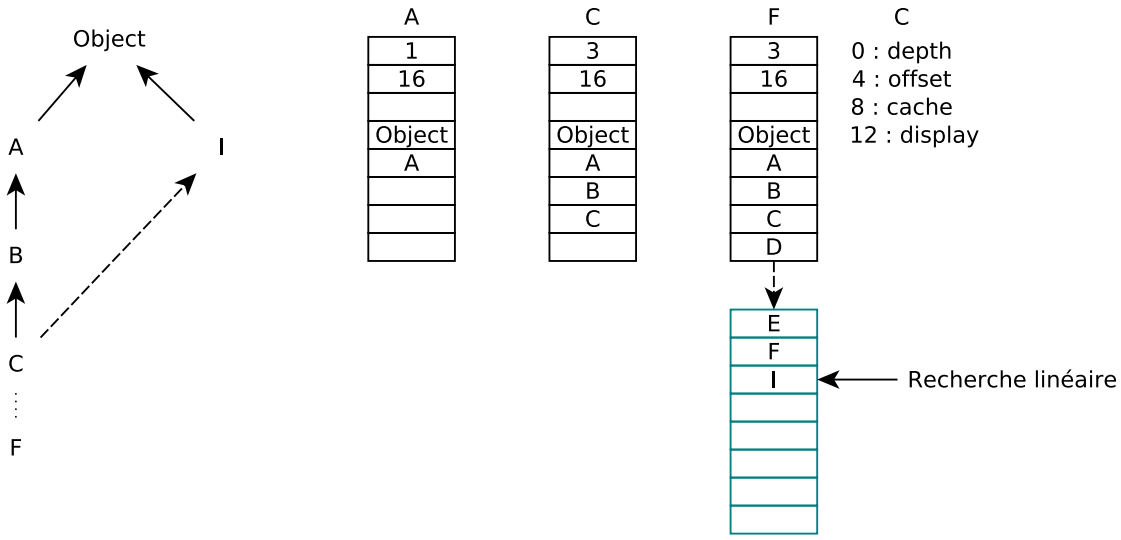


FIGURE 6 – Schéma du test de sous-typage dans Hotspot.

Un test de sous-typage par rapport à la première catégorie sera donc un test de Cohen. Cette technique est en temps constant et le résultat est très rapide. Par contre, pour un type cible de la secondaire catégorie, une recherche linéaire sera effectuée dans un tableau secondaire.

Ce tableau secondaire est donc constitué des interfaces implémentées par la classe ainsi que des classes dont la profondeur est supérieure à 8. Pour les grandes hiérarchies de types le test de sous-typage effectuera toujours une recherche linéaire.

Ce mécanisme n'est donc plus en temps constant pour ce cas là. Le test en général n'est pas en temps constant à cause de ce test secondaire. Pour limiter le coût de la recherche linéaire, un cache d'un élément qui contiendra un test de sous-typage positif dans la liste secondaire est utilisé. L'enjeu est donc de déterminer très rapidement si le type cible est un type primaire ou secondaire. Pour cela, les auteurs utilisent un champs $offset$ dans la table de méthodes. Ce champs contient soit $display[T.depth]$ pour un type primaire ou alors l'offset du cache pour un type secondaire.

Pour optimiser le test de sous-typage, le compilateur tente de déterminer à la compilation les types pour supprimer le test. D'autres optimisations sont aussi réalisées pour éviter d'utiliser la recherche linéaire. Le test est donc coûteux dans le pire des cas mais c'est relativement rare grâce à plusieurs petites optimisations.

Ce test est également compatible avec l'*inlining* d'après l'article. Cette affirmation est cependant à discuter, car les auteurs ne précisent pas s'ils choisissent d'inliner tout le test (y-compris la fonction de recherche linéaire). En effet, inliner la totalité de la recherche linéaire, ainsi que le reste du test pourrait provoquer un surcoût de la taille du programme qui serait non négligeable. Il est aussi compatible avec le chargement dynamique. Par contre, il n'est pas en temps constant. L'espace semble être linéaire puisqu'il augmente proportionnellement au nombre de classes.
Le test est également compatible avec le sous-typage multiple de Java. Voici le code du test de Hotspot, quelques optimisations sont appliquées mais l'idée générale est la suivante :

```
S.is_subtype_of(T) := {
    int off = T.offset;
    if (T == S[off]) return true;
    if (off != &cache) return false;
    if (S == T) return true;
    if( S.scan_s_s_array(T) )
    {
        S.cache = T;
        return true;
    }
    return false;
}
```

Ce tableau compare la résolution d'un test de sous-typage par rapport à une cible type primaire (classe, type primitif ou tableau de types primaires) et secondaire (interface, tableau d'interfaces).

| Taille de la hiérarchie des types | Type primaire | Type secondaire |
|---|---|---|
| <8 | Test de Cohen | Cache + recherche linéaire |
| >8 | Cache + recherche linéaire | Cache + recherche linéaire |

FIGURE 7 – Tableau récapitulatif du test de sous-typage de Hotspot

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 et des tests par rapport à des interfaces. Ils disent avoir choisi d'optimiser ces cas défavorables pour l'espace mémoire plutôt que le temps car ils sont rares. Une condition non remplie est donc le temps constant.

### 2.3.2 Test de sous-typage basé sur des trits

Dans [ACG01] est présenté le test de sous-typage implémenté dans Jikes RVM. Plusieurs compilateurs sont présents dans Jikes, avec des objectifs différents pour chacun d'entre eux. Pour le test de type dynamique le compilateur de base appellera une méthode tandis que le compilateur optimisé effectuera un *inlining* systématique du test de sous-typage.

Tous les objets dans Jikes ont un pointeur vers leur TIB (*Type Information Block*). Ce TIB contient :

- Une référence vers l'IMT
- Une référence vers la table de méthodes
- Une référence vers le *VM_Type*

Le *VM_Type* a trois sous-classes finales : *VM_Primitive* (type primitif), *VM_Class* (classe propre ou interface, un champs dans cette classe permet de différencier) et *VM_Array* (tableau). Le test de sous-typage sera différent selon son *VM_Type*.

Le compilateur optimisé effectuera des analyses statiques pour essayer de supprimer le plus de tests possibles lors de la compilation. Par exemple les tests de sous-typage avec une cible qui est une classe finale [2] ne peuvent être positifs que si la source est égale à la cible.

Le fonctionnement du test de sous-typage par rapport aux interfaces repose sur un système de *trit* (pour *Tertiary digit*). Un trit peut avoir trois valeurs : *yes*, *no* et *maybe*. Chaque classe possède un tableau indexé par les identifiants des interfaces et contenant des trits. Chaque case du tableau contient le résultat du test de sous-typage par rapport à l'interface. Lors de la création d'une classe, son vecteur de trits pointe vers un tableau global rempli de valeurs *maybe*. Lors du premier test de sous-typage, le système détecte que la classe référence ce tableau global. Le système alloue un nouveau vecteur de trits (de taille fixe) et le remplit de valeurs *maybe*.

Ensuite, le test de sous-typage demandé est exécuté par un appel opaque au coeur de la machine virtuelle. Le résultat du test (*yes* ou *no*) est stocké à l'indice correspondant à l'identifiant de l'interface.

Si un test de sous-typage est provoqué par rapport à une interface dont l'identifiant est plus grand que les bornes de ce tableau de taille fixe, un nouveau tableau est alloué. Il est ensuite rempli de valeurs *maybe*, avant de stocker le résultat du test de sous-typage.

Pour économiser de la mémoire et optimiser le système, les interfaces les plus utilisés (*Cloneable*, *Serializable* par exemple) se voient affecter des identifiants de valeurs proches de 0. Une autre optimisation est de partager un même tableau de trits entre deux classes ayant les mêmes interfaces implémentées.

Le système est généralement efficace par le jeu des numérotations favorables aux interfaces souvent utilisées, et aux allocations de tableaux supplémentaires de manière paresseuse. Néanmoins, l'efficacité de cette implémentation pour un programme avec beaucoup d'interfaces est sans doute assez peu efficace.

2. Une classe finale est une classe qui n'a pas de sous-classes, également appelé *frozen* ou *sealed* selon les langages.

### 2.3.3 Implémentation des interfaces par IMT

La technique basée sur les IMT (*Interface Method Table*) est utilisée pour traiter la sélection d'une méthode d'une interface dans JikesRVM. Cette implémentation est décrite dans [ACFG01] (à l'époque Jikes RVM s'appelait encore Jalapeño). L'implémentation utilise une table de hachage de taille fixe. À l'intérieur, sont hachées les identifiants des méthodes introduites par des interfaces. La résolution des conflits de hachage est faite par *separate chaining* c'est à dire une liste chaînée dans un indice du tableau où apparaissent des conflits. Il est précisé que ces identifiants sont attribués incrémentalement dans l'ordre de découverte des méthodes.

Cette table de hachage est appelée *Interface Method Table*. Leur objectif est d'avoir l'efficacité des tables à accès direct mais sans utiliser autant de mémoire. Ils effectuent donc une coloration dans la table de hachage pour optimiser l'espace. Le problème est que Java fonctionne en monde ouvert, la coloration est plutôt adaptée pour le monde fermé. Il a donc fallu mettre en place une gestion des collisions pour pouvoir incrémentalement rajouter des éléments dans les IMT. Pour résoudre un conflit, un morceau de code est généré à la volée et la case contenant le conflit pointe dessus. Lors d'une sélection de méthode il est donc possible d'obtenir directement l'adresse de la méthode ou alors d'effectuer une résolution d'un conflit de hachage. Cette implémentation est relativement efficace tant qu'il n'y a pas trop de conflits ni trop d'interfaces. Néanmoins, un grand nombre d'interfaces ou une grande hiérarchie de classes avec plusieurs interfaces entraîneront beaucoup de conflits et donc une très forte dégradation des performances. De plus, les tables de hachage sont remplies de manière très inégales : certaines seront presque vides tandis que d'autres auront énormément de collisions.

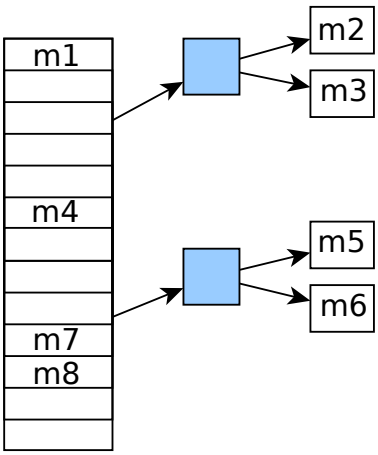


FIGURE 8 – Implémentation de la sélection d'une méthode des interfaces dans JikesRVM.

Dans la figure 2.3.3 ci-dessus, les identifiants des méthodes sont hachés et pointent directement vers la méthode. Dans le cas d'un conflit de hachage, les cases bleues représentent la

portion de code compilé permettant de résoudre les conflits entre les méthodes. Plus précisément, la case de la table de hachage contient une arbre binaire plutôt qu'une simple liste chaînée. Cette implémentation est relativement efficace lorsque il y a peu de conflits.

### 2.3.4 Table à accès direct

Une table à accès direct est utilisée dans SableVM notamment. Cette table sert à effectuer la sélection des méthodes introduites par des interfaces (instruction *bytecode invokeinterface*). Le postulat de base est d'effectuer une sélection de méthode définie dans une interface aussi rapidement que toute autre méthode. Le coût final est donc analogue à celui d'une instruction *invokevirtual*.

La table à accès direct représente une grande matrice avec les interfaces et les méthodes. Chaque méthode définie dans une interface se voit attribuer un identifiant unique. Cette table sera indexée par ces identifiants de méthodes. Chaque classe se voit associée sa propre table à accès direct. Elle contiendra autant d'interfaces que la classe en implémente. Cette table sera en grande majorité vide puisque elle ne contiendra que les adresses des méthodes définies dans les interfaces.

Pour effectuer une sélection de méthode, il suffit de récupérer son identifiant et puis d'aller directement à l'indice donné pour appeler la méthode. Cette grande matrice sera en très grande partie vide étant donnée que sa taille correspond à l'identifiant de méthode le plus élevé. Cette implémentation est très efficace en termes de temps mais au détriment d'une consommation mémoire démesurée (nombre de classes $\times$ nombre de méthodes des interfaces).

### 2.3.5 Hachage parfait

Le hachage parfait ( [Duc08] pour son utilisation pour le test de sous-typage) est une technique s'appuyant sur une table de hachage. Il s'agit d'une variante d'un hachage classique pour éliminer les collisions. Les définitions suivantes sont tirées de l'article [DM11].

**Définition 5.** *Hachage parfait*
Pour $I$ un ensemble non vide d'entiers. $N \times N \rightarrow$ N est une fonction telle que $hash(x, y) < y$ et $hash(x, y) \leq x$ pour tous $x, y \in N$. *Le paramètre de hachage parfait* de $I$ est le plus petit $H \in N$ telle que la fonction $h$ qui pour $x$ fait correspondre $h(x) = hash(x, H)$ est injective sur $I$.

C'est à dire que pour tout $x, y \in I$, $h(x) = h(y)$ implique $x = y$. Cette définition est étendue aux ensembles vides en considérant que $H = 1$ quand $I = \emptyset$.

**Définition 6.** *Hachage parfait de classe*
Soit $(X, \preceq)$ une hiérarchie de classe possédant des identifiants de classes injectifs $id : X \Rightarrow N$. Le hachage parfait s'applique pour chaque classe $c$ dans $X$ en considérant l'ensemble $I_c = id_d | c \preceq D$. Le paramètre $H_c$ résultant est la taille de la table de hachage (représentée par un tableau) de la classe $c$. Et pour chaque super-classe $d$, la table contient $id_d$ à la position $h_c(id_d)$. Toutes les autres positions $j$ contiennent un entier quelconque $l$ tel que $h_c(l) \neq j$.

$hash(x, y) \leq x$ est une contrainte qui n'est pas strictement nécessaire, mais elle a été vérifiée pour les fonctions de hachage testées dans [DM11]. Ceci implique que $h_c(0) = 0$ pour

tout $c$. Dans les hiérarchies de classes avec une racine 0 est un identifiant convenable pour la racine et c'est aussi une valeur pour les entrées vides de la table aux positions $j > 0$. Pour les hiérarchies de classes sans racine, n'importe quel entier non-nul peut représenter une entrée vide à la position 0 de la table. Mais cet entier ne doit pas être utilisé pour numéroter une classe.

On choisit la fonction $ET$ binaire comme fonction de hachage car c'est la plus efficace d'après l'article. Il ne s'agit pas de refaire des expérimentations pour le démontrer. Dans ce cas les données représentent des identifiants d'interfaces. Plus précisément, ce sont les identifiants des super-interfaces. L'ensemble de données à hacher est donc immutable, il est possible de calculer un masque de hachage pour éliminer les collisions. Ce masque est calculé par une suite d'opérations binaires sur ces identifiants pour s'assurer que la fonction de hachage est injective sur la table de hachage. Le hachage parfait calcule donc un masque de hachage parfait à partir duquel on déduit la taille de la table de hachage ($masque + 1$).

Les interfaces sont donc numérotées dans l'ordre de leur chargement. Ces identifiants sont ensuite rangés à l'indice donné dans la table de hachage. Il est donc possible de les retrouver rapidement. Le hachage parfait garantit qu'il n'y a pas de collisions dans la table de hachage. Ceci signifie que l'accès est réellement en temps constant. De plus, l'espace occupé reste faible : il est linéaire dans la taille de la relation de spécialisation des interfaces.

Le hachage parfait est utilisé pour le test de sous-typage par rapport à des interfaces dans le cas de Java. Si on veut tester si la classe B est sous-type d'une interface I, il s'agit de trouver (ou non) l'identifiant de I dans la table de hachage de B. Si cet identifiant est présent dans la table le test sera positif, sinon négatif. Le hachage parfait remplit les cinq exigences pour le test de sous-typage énoncées dans [Duc08] :

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

Dans le cadre du **test de sous-typage**, le hachage parfait reste néanmoins deux fois plus lent que le test de Cohen (technique classique d'implémentation d'un sous-typage en héritage simple). Si le type cible du test est une classe, ce dernier sera donc préféré. Par contre, pour les cibles qui sont des interfaces, donc dans un contexte d'héritage multiple le hachage parfait s'applique. Pour implémenter des tests de sous-typage en héritage ou sous-typage multiple il n'y a pas de consensus sur la manière de les implémenter.

Pour **l'appel de méthode** le hachage parfait est aussi utilisé (uniquement pour les méthodes 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.

Le fonctionnement est très similaire au test de sous-typage par rapport à l'interface ayant introduit la méthode. Les tables de hachage contiennent des doubles entrées correspondant à l'identifiant ainsi qu'un pointeur vers le groupe de méthodes introduites par l'interface.

Une fois le pointeur récupéré dans la table de hachage, un appel classique est effectué à une position fixe dans ce groupe de méthodes. Il est donc possible d'utiliser le hachage parfait pour le test de sous-typage et pour l'appel de méthode (de manière presque identique).

# 3 État de l'art des implémentations

## 3.1 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 benchmark 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 lancée 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, CacaoVM ou encore Hotspot.
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 d'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 profilage (*profiling*) pendant l'exécution, par exemple en comptant les appels de méthodes et en optimisant de manière poussée celles qui sont souvent appelées.

Ces deux approches sont souvent utilisées en parallèle. Il est par exemple possible d'optimiser les appels de certaines méthodes lors de la compilation. Mais il est aussi possible d'optimiser une méthode qui est très souvent utilisée. Pour ce faire, il faut pouvoir compter à l'exécution le nombre d'appels.

**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.

## 3.2 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.

### 3.2.1 Cacao VM

**Présentation**

CacaoVM est à l'origine un projet de recherche développé par l'université de Vienne. Cette machine virtuelle est décrite dans [KG97]. 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 (cette statégie est appelée *Static single assignment*, SSA)
- 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.

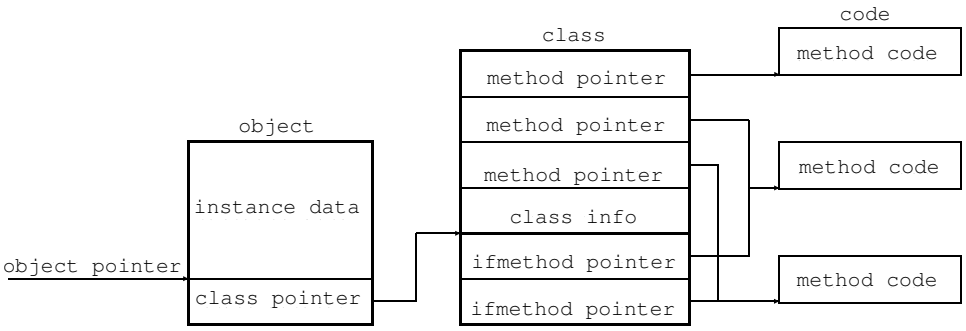


FIGURE 9 – Représentation mémoire des objets dans Cacao d'après [KG97]

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**

Dans Cacao est utilisée une représentation bidirectionnelle. L'objet *Class* contient la table de méthodes, et dans les indices négatifs il contient la table des méthodes introduites par des interfaces. La sélection d'une méthode introduite par une interface (opération *invokeinterface*) est réalisée grâce à une table à accès direct. Pour optimiser l'espace, un algorithme de coloration est utilisé. Le coût d'une instruction *invokeinterface* dans Cacao est donc à peine

plus élevé que celui de *invokevirtual*. Bien sûr, cette implémentation est faite au détriment de la consommation mémoire.

L'article [VHK97] présente l'implémentation du test de sous-typage utilisé dans Cacao. À l'époque de la parution de l'article la seule technique de test de sous-typage en temps constant était une une matrice $classe \times interface$ contenant les relations de sous-typage.

Les auteurs retiennent plusieurs critères pour qualifier un bon test de sous-typage. Il s'agit de l'efficacité en temps du test, en mémoire (test inlinable) et la compatibilité avec le chargement dynamique. C'est pourquoi ils choisissent cette matrice pour implémenter le test de sous-typage. Pour économiser de l'espace, ils présentent des manières d'encoder la relation de sous-typage pour économiser de l'espace. Pour le test de sous-typage par rapport à des classes ils utilisent le test de Cohen.

### 3.2.2 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 de démarrage, elle est donc plus efficace.

**Version client**

La machine virtuelle fait deux types d'optimisation : de l'analyse statique avant la compilation à la volée et du *profiling* à l'exécution. Ces deux approches combinées permettent une amélioration des 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*[3]. 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

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

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 :

- 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 dans la section 2.3.1. L'implémentation de la sélection de méthodes n'est pas décrite explicitement.

### 3.2.3 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. 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 10. 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.

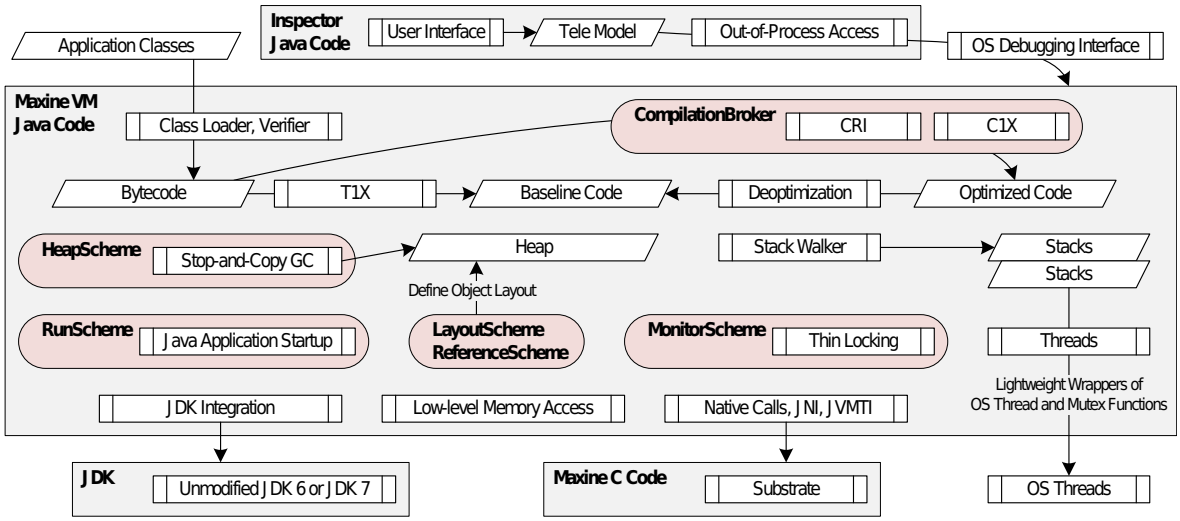


FIGURE 10 – Architecture interne de Maxine d'après [WD12].

**Test de sous-typage et sélection de méthode**

D'après le code de Maxine la sélection des méthodes introduites par des interfaces est réalisée avec des *itable*. Il faut donc à l'exécution trouver la bonne *itable* puis la méthode appelée. C'est donc une double sélection qui est réalisée pendant l'exécution.

La sélection de la bonne *itable* est réalisée grâce à une fonction de hachage :

1. Calcul de la taille de la table minimale pour qu'il n'y ait pas de collision
2. Hachage à l'exécution basé sur l'opération modulo

Les *itables* sont positionnées à la fin de la table de méthodes, il y a donc un hachage puis un décalage à la fin de la table de méthodes. Il s'agit donc d'un hachage en temps constant (sans collision) qui correspond au hachage parfait. Les identifiants sont néanmoins alloués sans utiliser la numérotation parfaite. Le code utilisé dans Maxine pour réaliser la sélection de la *itable* est le suivant :

```
public final int getMTableIndex(int id) {
        return (id % mTableLength) + mTableStartIndex;
    }
```

FIGURE 11 – Code de sélection d'une *itable* dans Maxine

Le test de sous-typage utilise aussi le hachage parfait. Pour un sous-typage par rapport à une interface, l'identifiant de l'interface est inliné dans la *itable*.

### 3.2.4 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 [Alp05]. 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 fut le premier développé. 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. JikesRVM a permis de développer des approches intéressantes. En particulier, un gros travail a été réalisé sur des systèmes d'optimisations adaptatifs dans cette JVM. Il s'agit de systèmes complexes permettant d'adapter certaines optimisations en fonction de l'état de l'exécution (comptage du nombre d'appels de méthodes par exemple). En 2001, Jalapeño devient JikesRVM.

JikesRVM possède un système complexe d'optimisations adaptatives. Le système est notamment décrit dans [AFG+00]. Jikes compile toutes les méthodes, mais tous les compilateurs n'ont pas le même niveau d'optimisation. C'est pour cette raison que le système d'optimisation adaptatif a été mis en place.

Il se base sur plusieurs composants importants :

- *Controller* : gère les autres composants du système, il coordonne le profilage et le système de recompilations en faisant le lien entre les deux. C'est également lui qui prend des décisions d'optimisations en fonction des données qu'il possède et du rapport bénéfice/coût de l'optimisation.
- *Recompilation subsystem* : fait appel aux compilateurs d'après les données reçues en entrée
- *AOS Database* : base de données des décisions prises et des données de profilage et d'analyses statiques.

Ces différents composants communiquent entre eux pour prendre des décisions d'optimisations. Par exemple le système permet de détecter qu'un lien méthode appelante vers méthode appelée est souvent utilisé. Dans ce cas, une décision d'optimisation est prise et celle-ci est enregistrée. Ces systèmes adaptatifs semblent efficaces mais au prix d'une très grande complexité.

**Test de sous-typage et sélection de méthode**

L'implémentation de la sélection des méthodes des interfaces est décrite dans [ACFG01], il s'agit d'une implémentation par **IMT** (cf section 2.3.3).

Selon les auteurs, 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

L'article présente le système de sélection d'une méthode introduite par une interface. Il s'agit donc de leur implémentation de l'opération bytecode *invokeinterface*. 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].

L'implémentation usuelle utilise des *interface table*, elles sont analogues aux tables de méthodes. Elles contiennent les méthodes implémentées par une classe qui sont introduites par une interface. Le système doit usuellement déterminer la bonne paire classe/interface et trouver la bonne méthode ensuite. Pour accélérer le processus, Jalapeño effectue quelques optimisations avec de l'analyse statique et du profilage. Si le compilateur est capable de dé-

tecter la classe qui implémente la méthode en question, il peut effectuer une dévirtualisation. Il s'agit de transformer un appel de type *invokeinterface* en un *invokevirtual*. Ce procédé est très similaire à un appel statique. Il s'agit de transformer une sélection de méthodes dans un contexte de sous-typage multiple en un contexte de sous-typage simple. Par exemple, avec une classe B qui implémente une interface I, si on a la portion de code suivante :

```
if(I instanceof B)
    i.foo();
```

Ici, il est possible de savoir que la méthode est implémentée dans B. Le compilateur peut effectuer un appel statique ou alors un inlining. L'appel de méthode en question est donc dans un contexte d'héritage simple plutôt que de sous-typage multiple.

L'implémentation de la sélection de méthodes dans Jikes RVM est une sorte de compromis entre des tables à accès direct comme dans Sable avec une solution moins gourmande en espace basée sur le hachage des méthodes. L'efficacité est bien sûr moindre car il faut gérer les collisions mais en contrepartie la consommation mémoire est moins démesurée.

Le test de sous-typage de JikesRVM est basé sur des **trits**, c'est à dire des entrées ternaires contenant le résultat du test pour chaque combinaison de types. Cette technique est décrite en section 2.3.2.

### 3.2.5 J3

**Présentation**

J3 [GTL+10] est la machine virtuelle Java développée avec 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. J3 est maintenant intégrée au projet VMKit.

**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 [4]. Elle est initialement basée sur GNU [5] Classpath et non pas sur openJDK (implémentation des librairies Java par Sun). J3 supporte maintenant partiellement OpenJDK (version 6 actuellement). D'après les tests de performances effectués dans l'article de présentation, 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.

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

5. GNU Classpath est une implémentation libre des bibliothèques standards de Java.

**Test de sous-typage et sélection de méthode**

J3 utilise le test de sous-typage de Hotspot [CR02] décrit précédemment dans la section 2.3.1. Elle utilise les IMT de JikesRVM décrite dans [ACFG01] et en section 2.3.3 pour l'implémentation des interfaces.

### 3.2.6 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.

**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). D'après les informations fournies dans l'article, le support de deux langages de bytecode est fait grâce à quelques tests dans le coeur de la VM. Certains mécanismes diffèrent légèrement, mais la plupart des techniques sont factorisables entre Java et C#.

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 envisagent de 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. Des *inlining* et appels statiques sont également possibles grâce à CHA. Toutes ces optimisations ont pour but de se rapprocher des performances des produits commerciaux (Hotspot). Bien sûr, les performances sont moins bonnes, mais l'objectif visait la compatibilité du système de composants 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.

### 3.2.7 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 également la synchronisation et les threads.

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 présentes 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. Java considèrera que ces deux méthodes sont identiques puisque il ne tient pas compte de l'interface dans laquelle a été introduite la 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 hasardeuse à la fois), pour éviter de gaspiller trop d'espace est d'allouer des objets dans ces trous.

## 3.3 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.

### 3.3.1 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 12 – Tableau comparatif des JVM

### 3.3.2 Performances

La performance des JVM n'est pas forcément essentielle pour une visée de recherche puisque l'intérêt est de tester et de comparer les différentes techniques d'implémentations. Cela permet toutefois d'avoir une idée. En particulier, la différence entre les machines virtuelles de recherches et les commerciales apparaît clairement.

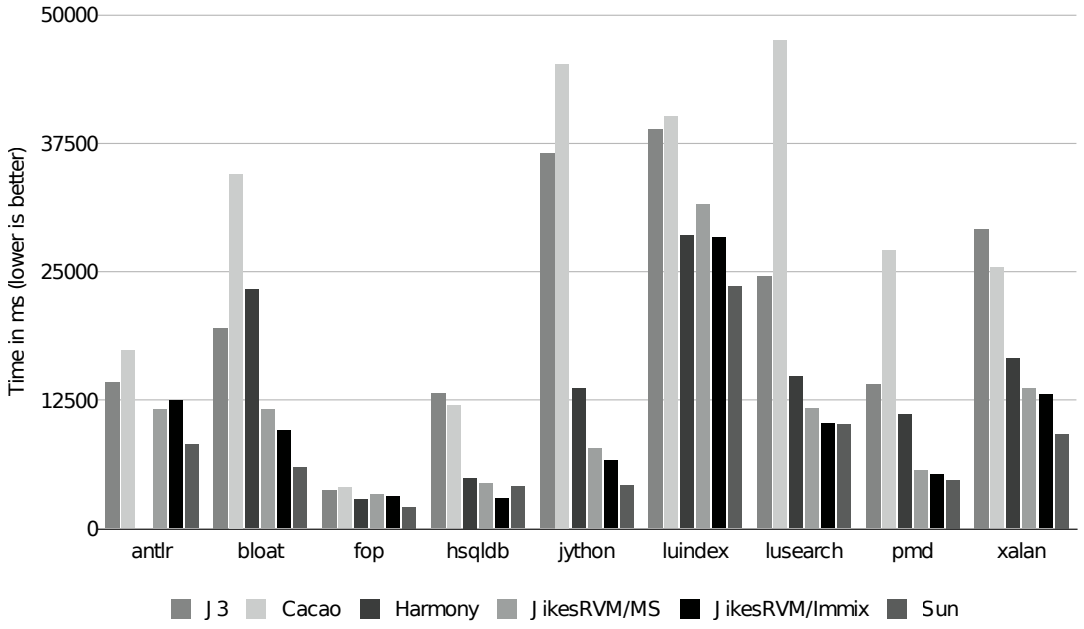


FIGURE 13 – Performances de quelques JVM d'après [GTL+10].

### 3.3.3 Test de sous-typage

Ce tableau présente le comparatif des techniques utilisées pour le test de sous-typage et la sélection de méthodes. Le test de sous-typage est parfois implémenté différemment selon que le type cible soit une classe ou une interface. C'est aussi le cas pour la sélection d'une méthode introduite par une classe ou une interface.

| JVM | La cible est une classe | La cible est une interface |
|---|---|---|
| Cacao | Test de Cohen | Matrice *classe* × *classe* compressée |
| Hotspot | Test de Cohen | Caches + recherche linéaire dans un tableau |
| J3 | Test de Cohen | Caches + recherche linéaire dans un tableau |
| JikesRVM | × | *trits* : valeurs ternaires indiquant le résultat du test |
| Maxine | × | *itables* et hachage parfait avec l'opération *modulo* |
| ORP | × | × |
| SableVM | × | × |

FIGURE 14 – Tableau comparatif des techniques d'implémentations du test de sous-typage

J3 utilise la même technique que celle de Hotspot.

### 3.3.4 Sélection de méthodes

Comparatif des implémentations de la sélection de méthodes introduites par des interfaces.

| JVM | Technique |
|---|---|
| Cacao | Table à accès direct et coloration |
| Hotspot | |
| J3 | *Interface Method Table* |
| JikesRVM | *Interface Method Table* |
| Maxine | *itables* et hachage parfait |
| ORP | |
| SableVM | Table à accès direct et allocation dans les trous |

FIGURE 15 – Tableau comparatif des techniques d'implémentations de la sélection de méthodes dans quelques machines virtuelles

**Définition 7. Une "miranda method"** est un *invokevirtual* référençant un *invokeinterface*.

Le terme *miranda method* est utilisé pour qualifier certaines méthodes dans Java. Il revient souvent et n'est que rarement expliqué par les personnes l'utilisant. D'après les explications fournies dans les sources de Maxine et Jikes, ce problème est posé par exemple depuis le code suivant :

```
interface I {
    void foo();
}

abstract class C implements I { }

class D extends C {
    public void foo() { }
}

void m() {
    C c = new D();
    c.foo(); // invokevirtual C.foo
}
```

Dans cette portion de code, un appel virtuel est utilisé alors qu'il s'agit d'une méthode d'une interface car non présente dans la classe abstraite *C*. Dans ce cas, le traitement consiste généralement à lever une erreur lors d'un *invokevirtual* sur foo. Ceci en rajoutant une méthode *foo()* sans code dans la classe abstraite *C*. Les *Miranda methods* proviennent d'un *bytecode* étrange si ce n'est bogué qui oblige les concepteurs de machines virtuelles à traiter ce problème.

D'après les concepteurs de Maxine ce terme de *Miranda* provient de la loi américaine dite "Miranda". Si l'accusé ne dispose pas d'un avocat, il aura droit à un avocat commis d'office. Le même traitement est donc appliqué pour les méthodes *Miranda* en ajoutant la méthode abstraite dans la classe *C*.

# 4 Intégration du hachage parfait aux JVM

## 4.1 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 pré-requis dans la représentation des tables de méthodes introduites par des interfaces dans 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 pré-requis mais cela entraîne des indirections supplémentaires et donc un coût plus élevé.

Chaque classe contiendra les identifiants des interfaces qu'elle implémente dans ses indices négatifs de la table de méthodes. Chaque entrée dans les indices négatifs contient l'identifiant de l'interface et un pointeur vers les méthodes qu'elle a introduit. Le schéma général de l'implémentation voulue est le suivant :

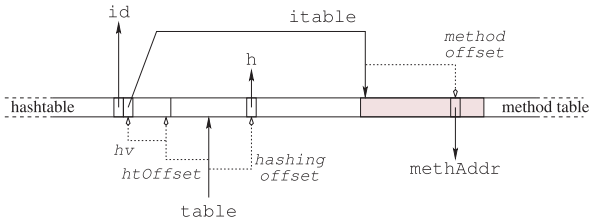


FIGURE 16 – Hachage parfait des interfaces pour Java d'après [DM11].

## 4.2 Choix de la JVM

Pour la partie pratique, il faut travailler sur une machine respectant ces pré-requis 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 imposé 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.

# 5 Stage

## 5.1 Introduction

Les principaux objectifs du stage visaient à poursuivre l'étude des machines virtuelles Java et d'implémenter le hachage parfait. Un autre travail plus distant de ces objectifs consistait à étudier l'adaptation du langage PRM sur une machine virtuelle Java, c'est à dire d'identifier les caractéristiques qui ne seraient pas compatibles avec les JVM en l'état.

Les parties suivantes présentent les différentes tâches à réaliser, puis leur implémentation avec des détails techniques ainsi que les résultats. Nous présenterons ensuite l'étude de PRM pour une machine virtuelle Java.

**Implémentation du hachage parfait**

Le hachage parfait peut être utilisé pour faire un test de sous-typage par rapport à une interface. Si le test de sous-typage est effectué par rapport à un type inconnu, il faut pouvoir déterminer si c'est une classe ou une interface. Pour cela, il est possible d'appeler une fonction qui effectuera le test à l'exécution.

Le hachage parfait est donc utilisé pour le test de sous-typage, mais aussi de manière presque identique pour l'appel de méthodes. En effet, dans la table de hachage sera présent une paire constituée de l'identifiant de l'interface et d'un pointeur vers le groupe de méthodes introduites par l'interface. Il faudra donc grouper les méthodes introduites par une interface dans la table de méthodes de la classe pour pouvoir pointer directement dessus.

**Numérotation parfaite**

Le hachage parfait permet d'avoir une fonction de hachage optimale qui garantisse qu'il n'y aura pas de collisions dans la table de hachage. En soi, le hachage parfait permet d'avoir un accès en temps constant. Mais pour limiter la taille d'une table de hachage en cours de construction il est possible d'avoir une approche inverse. Plutôt que de calculer un masque optimal pour des valeurs données, calculer les valeurs à hacher en fonction du masque pré-calculé.

C'est ce que fait la numérotation parfaite, en prenant une table de hachage partiellement remplie dans laquelle on doit rajouter des éléments. La numérotation parfaite permet de calculer des identifiants compatibles avec un masque donné, et une table de hachage donné. Ainsi, lors de la numérotation des interfaces on pourra affecter le bon identifiant qui sera compatible.

Les interfaces déjà découvertes seront déjà numérotées, il suffira d'appliquer le hachage parfait et de mettre en place la table de hachage.

Lors du chargement d'une interface inconnue jusqu'alors, il faudra lui affecter un identifiant calculé via la numérotation parfaite et ensuite de mettre en place la table de hachage.

**Amélioration du test de Cohen**

Pour l'implémentation en Java, le test de Cohen pour le test de sous-typage est toujours plus rapide que le hachage parfait.

Dans les implémentations usuelles la hiérarchie des classes est présente dans un tableau propre. Ceci implique de tester les bornes du tableau ou alors d'avoir un tableau de taille fixe (comme dans le test de Hotspot).

Il est possible d'éviter de tester les bornes du tableau en fusionnant les identifiants des classes dans les tables de méthodes. La table de méthodes contiendra donc les adresses des méthodes ainsi que les identifiants des classes. Il faut donc faire attention à ne pas confondre les identifiants avec des adresses. Les identifiants des classes sont donc maintenant à position invariante dans les tables de méthodes.

La solution est d'allouer les tables de méthodes de manière contigüe et dans une zone dédiée toutes les tables de méthodes. J3 ne remplit pas ces conditions à l'heure actuelle, et faire ces modifications serait sans doute trop long pour cette étude.

## 5.2 Analyse de la JVM choisie

Pour mener à bien ce projet il a fallu découvrir l'architecture technique de la JVM J3. J3 fonctionne grâce à VMKIT qui se base lui-même sur LLVM. LLVM est une suite d'outil de compilation contenant entre autres :

- Un compilateur JIT
- Sa propre représentation intermédiaire
- Des outils pour la génération de code

Ainsi, dans J3 le code Java tiré de fichiers *.class* est transformé à la volée en représentation intermédiaire de LLVM. C'est d'ailleurs pour cela que VMKIT se base sur LLVM, pour pouvoir utiliser leur compilateur à la volée. Le projet VMKIT est une suite d'outils qui est constituée à partir de briques diverses :

- Un compilateur JIT : LLVM
- Un *Garbage collector* : *mmtk* (utilisé dans Jikes RVM)
- Un gestionnaire de tâches : threads *POSIX*
- Architecture de base d'une machine virtuelle

Ces différents éléments ont été assemblés pour former VMKIT. De par ses outils offerts ainsi que l'aspect paramétrage de VMKIT, il peut être qualifié de framework de machines virtuelles. On peut par exemple retrouver des méta-classes permettant de représenter un objet ou une table de méthodes.

VMKIT possède également sa propre machine virtuelle java appelée J3.

Cette dernière est complètement compilée en représentation intermédiaire LLVM. Elle possède également des structures LLVM déclarées dans le code de J3 pour introduire un niveau méta. Celui-ci est nécessaire pour pouvoir appeler certaines méthodes de J3 à l'exécution. J3 possède donc un compilateur permettant de traduire J3 en structures LLVM. Elle possède également un compilateur de *bytecode* Java vers la représentation LLVM.

LLVM utilisera son propre compilateur à la volée pour faire fonctionner J3 ainsi que les programmes Java compilés.

Le bootstrap de J3 est réalisé en chargeant toutes les classes Java des librairies de base et en les compilant. Cette approche est plus simple que celle basée sur des images mémoire comme pour Maxine ou JikesRVM, mais en contrepartie le temps de démarrage de J3 est très élevé.

## 5.3 Déroulement de l'implémentation du hachage parfait

Dans cette section est présentée le déroulement de l'implémentation du hachage parfait dans J3. La première étape était de coder le hachage parfait en C. Ensuite attribuer à chaque interface un identifiant via la numérotation parfaite.

L'étape suivante était de mettre en place les structures pour le hachage parfait et ensuite de réaliser un test de sous-typage. La dernière étape était de grouper les méthodes des interfaces et ensuite de réaliser un *invokeinterface* grâce au hachage parfait. Une partie résultats et difficultés présente ensuite un bilan, des difficultés et perspectives.

### 5.3.1 Hachage parfait

Les algorithmes sont présents en pseudo-code Lisp dans l'article [DM11]. Le premier travail a été de développer le hachage parfait en C dans une première version en C pour effectuer quelques tests. Il s'agit de quelques fonctions permettant de calculer le masque de hachage en fonction de certaines valeurs en entrée (qui représentent des identifiants d'interfaces).

À partir de cette version en C a été adaptée une version en C++ en vue de l'intégrer à J3. Le hachage parfait est présent dans une classe dédiée *PerfectHashing* avec plusieurs méthodes pour numéroter ainsi que les structures de données qui gèrent la numérotation.

```
/*
        Perfect hashing with AND
        Compute a mask composed by least discriminant bits
        and returns the hashtable size
        ids : identifiers to hash
        sizeIds : size of ids' array
        return mask + 1 : size of the hashtable to create
*/
int phand(int *ids, int sizeIds)
{
        if(sizeIds == 0)
                return 1;

        int ior = ids[0];
        int and = ids[0];

        // Computes mask
        for(int i=1; i<sizeIds; i++)
        {
                ior = ior | ids[i];
                and = and & ids[i];
        }

        // mask contains all discriminant bits
        int mask = ior ^ and;

        // Fills hashTable with null values
        for(int i=0; i<mask+1; i++)
                hashTable[i] = -1;

        int new = 0;

        for(int i=highestBit(mask); i>=0; i--)
        {
                if(extractBit(mask, i) == 1)
                {
                        // New = mask with switched bits
                        new = mask ^ (1 << i);

                        if(phandp(ids, new, size) == 1)
                                mask = new;
                }
        }

        return 1 + mask;
}
```

```
/*
        Perfect numbering
        ids : identifiers to hash
        n : identifier to hash
        size : size of ids table
        hc : size of hashTable to create
*/
void pnand(int *ids, int n, int size, int *idc, int *hc)
{
        int mask = phand(ids, size) - 1;

        for(int b=0; (n+size) > (1 << logcount(mask)); b++)
        {
                // When there are not enough 1-bits
                if(extractBit(mask, b) == 0)
                        mask = mask ^ (1 << b);
        }

        phandp(ids, mask, size);

        computeLeastFreeIds(n, mask, idc);

        *hc = mask + 1;
}


/*
        Checks if the mask is a perfect hashing function for the identifiers
        ids : Table of identifiers
        mask : Actual mask for PH
        size : Size of ids' table
*/
int phandp(int *ids, int mask, int size)
{
        for(int i=0; i<size; i++)
        {
                int hv = ids[i] & mask;
                if(hashTable[hv] == mask)
                        return 0;
                else
                        hashTable[hv] = mask;
        }
        return 1;
}
```

```
/*
        Extract bit number pos in byte
*/
int extractBit(int byte, int pos)
{
        int op = pow(2, pos);

        if((byte & op) == 0)
                return 0;
        else
                return 1;
}

/*
        Return number of 1-bit in n
*/
int logcount(int n)
{
        // Size of int in bytes
        long int bound = pow(2, sizeof(int)*8);
        int count = 0;

        // For each bit
        for(long int i=bound; i>0; i/=2)
        {
                if(n & i)
                        count++;
        }

        return count;
}

/*
        Return the position of highest bit set to 1 from mask
*/
int highestBit(int mask)
{
        long int msb = pow(2, sizeof(int)*8);
        int pos = sizeof(int)*8;

        while(msb > 0 && !(mask & msb))
        {
                msb /= 2;
                pos--;
        }

        return pos+1;
}
```

*computeLeastFreeIds* est une fonction qui retourne le premier identifiant correspondant au masque et aux données en entrée. C'est une structure d'union d'intervalles précédemment décrite.

### 5.3.2 Numérotation des interfaces

Pour utiliser le hachage parfait, les interfaces doivent posséder un identifiant unique qui servira pour le sous-typage. Cette numérotation est incrémentale dans l'ordre de découverte (chargement) des interfaces du programme.

Une première version consiste en une numérotation purement linéaire. Mais la numérotation parfaite permet d'optimiser l'espace des tables de hachage. Plutôt que d'optimiser l'espace en fonctions des données il s'agit d'optimiser les données pour qu'elles ne rentrent pas en collision avec des identifiants déjà présents. Ainsi lors de la numérotation d'une classe formant une feuille de la hiérarchie, il est possible de lui attribuer un identifiant libre correspondant mieux aux identifiants de ses super-classes. L'attribution des identifiants libres pour numéroter les interfaces est réalisée avec une structure d'union d'intervalles :

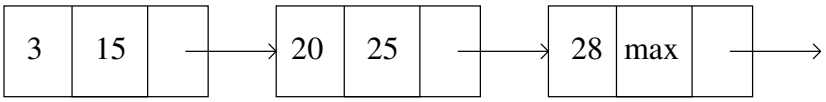


FIGURE 17 – Structure d'union d'intervalles pour la numérotation

Lors de la numérotation d'une nouvelle interface, cette structure sera parcourue pour trouver l'identifiant qui peut correspondre à une position inoccupée de la table de hachage. S'il n'y en a pas le premier identifiant disponible sera attribué. Si un identifiant est sélectionné au milieu d'un bloc de la liste chaînée, ce bloc est partagé en deux. Ces ensembles de blocs ont une occupation mémoire relativement faible.

Ainsi l'attribution des identifiants d'interfaces sont fait en optimisant l'espace des tables de hachage. Il s'agit cependant d'une optimisation locale, qui sera faite à chaque appel à la numérotation parfaite. Il est sans doute possible d'obtenir des identifiants plus performants pour numéroter une hiérarchie de classes pas encore chargée. Seulement, le gain est assez minime d'après [DM11], un identifiant retourné par appel à la numérotation est donc acceptable.

Dans J3, les informations de types sont situées dans la table de méthodes (pour un accès plus rapide), la méta-classe contenant assez peu d'informations. L'attribution d'un identifiant pour une interface est donc fait au moment de la création de sa table de méthodes (la méta-classe représentant la table de méthodes).

### 5.3.3 Sous-typage

Changer la manière de réaliser un test de sous-typage quand la cible est une interface implique de rajouter des structures de données. Dans J3, les informations servant à réaliser un test de sous-typage sont présentes dans la classe *JavaVirtualTable*.

La table de hachage parfaite contiendra donc les interfaces présentes dans la liste des types secondaires (technique utilisée initialement). Cette table de hachage parfaite sera créée lors de l'instanciation d'une classe *JavaVirtualTable*, qui représentera donc une table de méthodes d'une classe.

Le déroulement de la création de la table est le suivant :

1. Récupération des identifiants des interfaces parentes
2. Appel au hachage parfait pour calculer le masque de hachage
3. Allocation et remplissage de la table

Ces tables de hachage parfaites servent ensuite à réaliser un test de sous-typage. Lors d'un test avec une cible qui est une interface, on peut rapidement savoir le résultat du sous-typage. Là où la technique originelle va effectuer une recherche linéaire dans un tableau, le hachage parfait donnera un résultat en temps constant.

Il suffit de récupérer l'identifiant de l'interface cible, le masque de la classe source, et d'appliquer la fonction de hachage pour s'assurer que l'identifiant est bien présent à sa position supposée. S'il est présent le test est positif, sinon il est négatif. Ainsi, en matière de nombres de cycles utilisés la technique de Hotspot est plus coûteuse que le hachage parfait.

Dans J3 modifiée, le hachage parfait s'applique si la cible est une interface. Le test de sous-typage modifié utilise la technique de Hotspot pour les classes et le hachage parfait pour des cibles de type interface.

Le test de sous-typage final est le suivant :

```
bool JavaVirtualTable::isSubtypeOf(JavaVirtualTable* otherVT)
{
        assert(this);
        assert(otherVT);
        if (otherVT == ((JavaVirtualTable**) this)[otherVT->offset])
                return true;
        else if (otherVT->offset != getCacheIndex())
                return false;
        else if (this == otherVT)
                return true;
        else{
                // If the target is an interface, use PH
                if (otherVT->cl->isInterface()){
                        int otherID = otherVT->id;
                        int hv = mask & otherID;
                        return (hashTable[hv].id == otherID);
                }

                for (uint32 i = 0; i < nbSecondaryTypes; ++i){
                        if (secondaryTypes[i] == otherVT){
                                cache = otherVT;
                                return true;
                        }
                }

                if (cl->isArray() && otherVT->cl->isArray())
                        return baseClassVT->isSubtypeOf(otherVT->baseClassVT);
        }
        return false;
}
```

FIGURE 18 – Test de sous-typage utilisant le hachage parfait dans J3

### 5.3.4 Appel de méthodes

L'appel de méthodes grâce au hachage parfait est très similaire à un test de sous-typage. Une entrée de la table de hachage est une paire *identifiant, pointeur*. L'identifiant est celui de l'interface (utile pour le sous-typage) et le pointeur représente un bloc dans la table de méthodes de la classe en question. Chaque bloc de méthode est appelé *itable*, il s'agit des implémentations des méthodes introduites dans l'interface.

Pour économiser de l'espace, il est possible de se servir d'un *offset* (décallage) dans la table de méthodes plutôt qu'un pointeur (un entier 32 bits plutôt qu'une adresse en 64 bits). L'espace utilisé est donc de 64 bits par entrée dans la table de hachage. C'est la solution retenue lors de l'implémentation.

Pour implémenter l'appel de méthode avec le hachage parfait il est nécessaire de faire un travail sur les tables de méthodes. Il faut en effet que pour chaque interface d'une classe, l'im-

plémentation de ses méthodes regroupées dans la table de méthodes. De plus, il faut calculer toutes les interfaces qu'implémente une classe. Il s'agit déjà des super-interfaces directes, et ensuite des interfaces implémentées par héritage mais qui ne sont pas déjà implémentées par les super-classes. Les méthodes des classes doivent être distinguées des méthodes introduites par des interfaces et sont groupées au début des tables de méthodes.

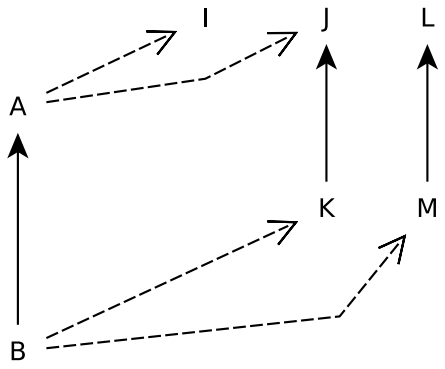


FIGURE 19 – Super-interfaces à implémenter

Dans le schéma ci-dessus, la classe $B$ a deux super-interfaces directes. Mais elle doit également implémenter les méthodes de l'interface $L$. L'interface $J$ en l'occurrence est déjà implémentée dans la super-classe, il n'y a pas besoin de le faire dans la classe $B$.

Certains problèmes se posent néanmoins pour des raisons historiques avec Java. Par exemple l'interface *Cloneable* permet de cloner un objet Java. Elle ne possède aucune méthode déclarée. Par contre, la méthode *clone()* est déclarée dans *Object*. Des raisons historiques justifient sans doute de tels mécanismes, ils ont été gardés pour conserver la compatibilité. Cependant, l'implémentation du hachage parfait est complexifiée car cette méthode n'apparait pas dans l'interface *Cloneable* mais bien dans *Object*. La classe *Object* est donc une sorte d'interface et de classe. Elle contient des méthodes considérées comme des méthodes d'interfaces et des méthodes de classe.

Dans l'implémentation, il faut la considérer comme à la fois une interface et une classe. Cette implémentation récupère l'adresse de la méthode appelée dans le groupe de méthodes introduite par l'interface. L'adresse renvoyée correspondant à une position dans la table de méthodes de la classe. Cette adresse peut être soit le code de la méthode ou alors une portion de code faisant appel au compilateur. Le compilateur paresseux compilera si besoin et remplacera l'adresse du *trampoline* de compilation par le code compilé de la méthode. L'implémentation de l'opération *invokeinterface* est la suivante en utilisant le hachage parfait :

```
extern "C" void* j3ResolveInterface(JavaObject* obj, JavaMethod* meth, uint32_t index)
{
        llvm_gcroot(obj, 0);
        word_t result = 0;

        // Perfect hashing for method dispatch
        int id = meth->classDef->virtualVT->id;
        int mask = JavaObject::getClass(obj)->virtualVT->mask;
        JavaVirtualTable* vtable = JavaObject::getClass(obj)->virtualVT;

        uint32 itable = vtable->hashTable[mask & id].itable;
        uint32 offset = itable + meth->itableOffset;

        result = ((word_t*)(vtable))[offset];
        return (void*)result;
}
```

FIGURE 20 – Appel de méthode d'une interface dans J3 modifiée

## 5.4 Résultats et difficultés rencontrées

La version de J3 modifiée possède donc les outils pour construire des tables de hachages parfaites. Le test de sous-typage par rapport à une interface utilise le hachage parfait en lieu et place de l'ancienne technique utilisée par Hotspot.

L'appel de méthodes avec le hachage parfait est une implémentation incomplète. Certains cas précis posent encore problème. Par exemple l'architecture de J3 fait qu'il est difficile de traiter correctement la fusion de deux méthodes de mêmes noms dans deux interfaces différentes. Dans ce cas précis le hachage parfait doit outrepasser ce comportement pour rester correct. De plus certaines méthodes dans Java ont un fonctionnement particulier. C'est par exemple le cas des interfaces *Cloneable* et *Serializable*. Ces deux interfaces n'introduisent pas de méthodes, mais par convention implémenter ces interfaces indique une certaine sémantique et indique au programmeur de définir des méthodes privées avec une signature bien précise.

Pour ces raisons l'implémentation proposée est incomplète et n'est pas capable d'exécuter de benchmarks standards.

### Difficultés

La principale difficulté a été de comprendre le fonctionnement de J3 et par extension VMKIT. J3 ne possède absolument aucune documentation et la seule approche possible est la lecture du code source. De plus, son fonctionnement intrinsèque basé sur LLVM rajoute une étape de traduction qui complexifie le code de la machine virtuelle et oblige à faire certaines choses à plusieurs endroits différents (rajouter un attribut dans une classe par exemple). J3 s'est également révélée ne pas être prévue pour des modifications ultérieures. Il s'agissait plutôt d'une preuve de faisabilité pour VMKIT et pas d'une JVM réellement destinée à la recherche.

Le projet VMKIT est toutefois très intéressant car unique dans son genre et permet

de factoriser des points importants des machines virtuelles. Le fonctionnement de Java est également complexe à appréhender, en particulier pour certains points qui sont mal traités tel que la fusion de méthodes.

**Bilan**

Néanmoins, l'implémentation proposé est plus simple à développer et à comprendre et possède de nombreuses qualités déjà énoncées précédemment. L'importance d'un méta-modèle correct permettant de connaître la classe d'introduction d'une méthode est mise en valeur. Celui-ci permettrait de regrouper plus facilement les méthodes des interfaces ainsi que de nombreuses optimisations (méthodes monomorphes par exemple). Le hachage parfait permet de simplifier de nombreux points en plus d'être une technique transposable pour l'héritage multiple. Il est donc possible de l'utiliser pour des JVM bien qu'un méta-modèle adapté simplifierait grandement l'implémentation.

Pour une implémentation plus correcte et efficace l'apport du méta-modèle devrait être important. De plus, les techniques existantes présentées au cours de cette étude sont clairement très différentes et souvent perfectibles. Peu de techniques pour l'héritage multiple sont des concepts connus et utilisés en algorithmique, contrairement au hachage parfait.

# 6 Étude de l'adaptation de PRM sur une machine virtuelle Java

PRM est un langage développé initialement dans le cadre de la thèse de Jean Privat [Pri06]. C'est un langage à typage statique et à héritage multiple. PRM utilise aussi des modules et du raffinement de modules (analogue à l'héritage et à la spécialisation).

L'adaptation de PRM sur une machine virtuelle permettrait de lui apporter des propriétés intéressantes telles que l'introspection, le chargement dynamique ou encore, plus de compatibilité. En particulier, PRM pourrait gagner des bénéfices à être adapté sur une machine virtuelle Java. Il est par exemple possible de profiter des nombreuses bibliothèques Java pour les utiliser avec PRM. Ceci permettrait d'utiliser un système d'interfaces graphiques en PRM sans tout re-développer. La machine virtuelle qui permettrait d'exécuter du PRM devra donc être compatible avec Java. La question de l'adaptation de PRM pour une machine virtuelle présente donc plusieurs problèmes et enjeux qui restent pour certains ouverts. Une adaptation du bytecode Java ou l'utilisation d'une autre représentation intermédiaire semble inévitable pour conserver une implémentation cohérente. Dans cette future machine virtuelle pour PRM, il faudrait avoir plusieurs modèles qui cohabitent si les modèles objets de Java et PRM s'avèrent incompatibles. Il est possible d'essayer d'adapter PRM sans modifier ni la JVM, ni étendre le bytecode, à la manière de SCALA. Néanmoins, cette approche souffrira probablement d'un manque d'efficacité et d'une sémantique de l'implémentation parfois discutable.

Mais l'adaptation de PRM sur une machine virtuelle compatible avec Java n'est pas triviale. Certaines de ses propriétés semblent incompatibles avec les machines virtuelles Java en l'état :

– Généricité non effacée

– Héritage multiple
– Système de modules

L'héritage multiple entraîne en particulier de nombreuses difficultés. Par exemple, un appel de méthodes dans un contexte d'héritage multiple est bien plus complexe qu'en héritage simple. Il faudrait peut être ajouter une opération bytecode *invokegeneral* qui ferait un appel en héritage multiple et qui serait donc utilisée par défaut pour les classes de PRM.

## 6.1 Héritage multiple

PRM est un langage à héritage multiple qui utilise des modules. Son adaptation sur des machines virtuelles demande donc de conserver ces spécificités. Les spécifications du bytecode Java ne sont en l'état pas adaptées à l'héritage multiple. Par exemple au début d'un fichier *.class* est encodée la super-classe dans un champ précis. Dans cet exemple il faudrait modifier le bytecode. Mais d'autres approches sont envisageables.

La spécialisation de classes et l'héritage sont des notions introduites par SIMULA en 1973. Ces concepts apportent une grande force des objets : la réutilisation. L'héritage simple était l'implémentation usuelle. Des concepteurs se sont penchés sur l'implémentation de l'héritage multiple. Des difficultés apparaissent immédiatement avec l'héritage multiple. Le graphe de spécialisation en héritage simple est un arbre. En héritage multiple, il y a possiblement des diamants ce qui entraîne des conflits. Il n'y a pas de réponse universelle et consensuelle pour résoudre ce problème ainsi, plusieurs fonctionnements cohabitent :

– Héritage simple uniquement : SmallTalk
– Héritage multiple : C++, Eiffel, CLOS, Python
– Mixin : SCALA
– Sous-typage multiple : Java et C#

**Traduction de PRM en bytecode Java**

Pour garder la compatibilité de PRM avec les plateformes Java il faudrait pouvoir traduire du PRM en bytecode Java. Ceci n'est pas sans poser problème notamment pour l'héritage multiple. Le bytecode Java est en effet prévu pour ne contenir qu'une seule super-classe dans un fichier *.class*.

L'article [CCLS04] décrit la modification d'une JVM pour supporter l'héritage multiple. La motivation de départ est de ne pas dupliquer du code entre différentes classes des librairies standards Java. Plusieurs implémentations de collections en Java ont des implémentations semblables et parfois redondantes à cause de l'absence d'héritage multiple.

Les auteurs introduisent le concept d'*implémentation*. C'est une construction définie par l'association du code générique aux méthodes des interfaces. Il s'agit d'une sorte de classe abstraite contenant le code des méthodes.

La compilation est faite en modifiant le compilateur de Jikes et la JVM hotspot. Lors de la compilation une interface est créée, mais elle contient le code des méthodes. La JVM est modifiée pour accepter le code dans les méthodes des interfaces. L'ensemble est fonctionnel grâce aux aspects héritages multiples des interfaces de Java (Appel de méthodes, sous-typage). Cette modification permet donc l'héritage multiple des méthodes mais pas des attributs.

L'approche est néanmoins intéressante puisque les modifications semblent mineures d'après les auteurs. Mais elle n'est pas transposable directement à PRM car ce dernier est en véritable héritage multiple.

Cependant cela illustre un point important dans la conception de Java. Le système oblige à résoudre certains problèmes de l'héritage multiple sans en apporter toute la puissance.

## 6.2 Généricité

La généricité est une abstraction de haut niveau dans les langages de programmation. Elle permet de construire des algorithmes et fonctionnements abstraits sans traiter leur implémentation précise. Il est donc possible d'écrire du code qui est indépendant des types concrets. Le type utilisé pour définir une classe générique est appelé type formel. Ainsi, en définissant une classe *List<T>* qui sera utilisée plus tard avec *List<int>* et *List<String>*. Ici *T* sera le type formel et *int* et *String* deux types concrets. Tous les langages de programmation n'implémentent pas le système de la généricité de la même manière. La généricité n'existait d'ailleurs pas en Java jusqu'à la version 1.5. Prenons une classe minimale utilisant la généricité :

```
public class Generic<T>
{
        private T value;

        public Generic()
        {
                value = null;
        }

        public Generic(T val)
        {
                value = val;
        }

        public void setValue(T val)
        {
                value = val;
        }

        public T getValue()
        {
                return value;
        }
}
```

La classe suivante instancie une classe générique :

```
class Main
{
        public static void main(String[] args)
          {
                Generic<String> val = new Generic<String>("toto");

                String s = val.getValue();
        }
}
```

Le compilateur produit ce bytecode :

```
0: new           #2              // class Generic
3: dup
4: ldc           #3              // String toto
6: invokespecial #4              // Method Generic."<init>":(Ljava/lang/Object;)V
9: astore_1
10: aload_1
11: invokevirtual #5              // Method Generic.getValue:()Ljava/lang/Object;
14: checkcast     #6              // class java/lang/String
17: astore_2
18: return
```

FIGURE 21 – Extrait de bytecode utilisant la généricité en Java

À la ligne 11 du *bytecode* généré, la signature de la méthode indique que le type de retour de la méthode est *Object* et pas *String*. Le type paramétrée de la classe générique est *Object*. Il y a donc en Java un effacement du type paramétré. Il n'y a en effet pas de trace du type concret *String* ailleurs que lors des *checkcast*. Java considère que la classe est paramétrée par *Object* et utilise des tests de sous-typage à l'exécution pour s'assurer que les types correspondent. Ce type de fonctionnement est appelé généricité avec effacement de types. Une solution aurait été de conserver le paramètre de la classe générique dans le bytecode. Mais l'introduction tardive de la généricité en Java a sûrement conduit ses concepteurs à utiliser un tel système. En Java, la généricité est donc traitée de la manière suivante :

– Remplacement du type formel par *Object*
– Test de sous-typage dès qu'un type concret est utilisé

Il est possible d'étendre le méta-modèle des propriétés globales et locales à la généricité sans effacement de types. La thèse d'Olivier Sallenave [Sal12] propose une extension du méta-modèle pour cela.

Le langage PRM fonctionne avec une généricité non effacée. Il faudra donc faire cohabiter ces deux systèmes dans la future machine virtuelle pour PRM. Le méta-modèle propose un

traitement de la généricité avec une analogie entre les propriétés globales et locales et les types concrets et formels.

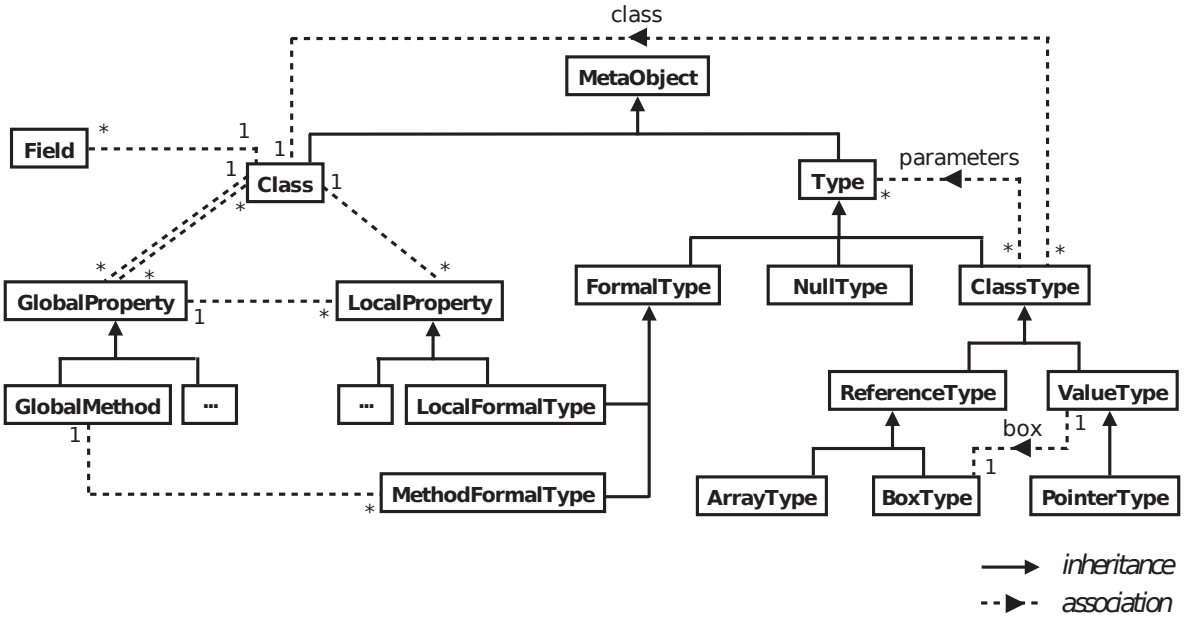


FIGURE 22 – Méta-modèle étendu à la généricité d'après [Sal12]

# 7 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. Mais ces réponses ne sont pas optimales car 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. Nous proposons une implémentation du hachage parfait dans J3 qui est incomplète mais présente toutefois une preuve de faisabilité. En particulier, le manque d'un méta-modèle correct dans la JVM a manqué pour implémenter plus efficacement le hachage parfait. Le hachage parfait est donc une technique élégante, qui permet de traiter les problématiques du test de sous-typage et de l'appel de méthode de manière efficace. C'est également une technique mieux maîtrisée d'un point de vue algorithmique que les autres techniques présentes dans les JVM. Cette technique efficace est compatible avec l'héritage multiple et le chargement dynamique. Ces caractéristiques seront probablement celles de 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.

[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.

[ACG01] Bowen Alpern, Anthony Cocchi, and David et al Grove. Dynamic type checking in jalapeno. In *USENIX Java Virtual Machine Research and Technology Symposium*, pages 41–52, 2001.

[AFG+00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. Adaptive optimization in the jalapeno jvm. In *ACM SIGPLAN Notices*, volume 35, pages 47–65. ACM, 2000.

[Alp05] Bowen et al Alpern. The jikes research virtual machine project : building an open-source research community. *IBM Syst. J.*, 44(2) :399–417, January 2005.

[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.

[CCLS04] Maria Cutumisu, Calvin Chan, Paul Lu, and Duane Szafron. Mci-java : A modified java virtual machine approach to multiple code inheritance. In *Virtual Machine Research and Technology Symposium*, pages 13–28, 2004.

[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.

[DP11] Roland Ducournau and Jean Privat. Metamodeling semantics of multiple inheritance. *Science of Computer Programming*, 76(7) :555–586, 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] David Gregg, M. Anton Ertl, and Andreas 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] Andreas Krall and Reinhard Grafl. Cacao - a 64-bit javavm just-in-time compiler. *Concurrency Practice and Experience*, 9(11) :1017–1030, 1997.

[Kra98] Andreas Krall. Efficient javavm just-in-time compilation. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 205–212, October 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$^{TM}$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.

[Pri06] Jean Privat. *De l'expressivité à l'efficacité une approche modulaire des langages à objets le langage PRM et le compilateur* prmc. PhD thesis, Université Montpellier 2, LIRMM, 2006.

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

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

[Sal12] Olivier Sallenave. *Contribution à l'efficacité des programmes orientés objet pour processeurs embarqués*. PhD thesis, Université Montpellier 2, LIRMM, 2012.

[Sin03] Jeremy 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.

[vdB06] Huib van den Brink. The current and future optimizations performed by the java hotspot compiler, 2006.

[VHK97] Jan Vitek, R Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *ACM SIGPLAN Notices*, volume 32, pages 142–157. ACM, 1997.

[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.

**Résumé**

Cette étude traite des langages de programmation à objets et plus particulièrement des systèmes d'exécution modernes : les machines virtuelles. Nous nous intéressons particulièrement au cas précis du langage Java et à sa machine virtuelle. Nous présentons dans un premier temps l'architecture abstraite et les grands principes des machines virtuelles Java. Puis sont présentées quelques machines virtuelles Java de recherches. L'état de l'art présente les grandes tendances d'implémentation. Nous nous intéressons particulièrement aux technique d'implémentation du test de sous-typage et de la sélection des méthodes des interfaces. Les techniques généralement employées sont très souvent perfectibles et aucune ne fait pour l'instant consensus.

Enfin, nous présentons une implémentation dans une machine virtuelle d'une alternative aux techniques existantes. Celle-ci repose sur le hachage parfait, qui peut être utilisé pour implémenter ces deux mécanismes. L'implémentation est intégrée à la machine virtuelle Java J3. Nous présentons le déroulement de l'implémentation puis les résultats obtenus et les perspectives. Nous étudions également l'adaptation d'un langage à héritage multiple dans une machine virtuelle Java.

**Abstract**

This study focus on object oriented programming langages, especially modern execution systems : virtual machines. We focus particulary on Java langage and his virtual machine. In a first time we present the abstract architecture and big principles of the Java virtual machine. Then we present some research's implementations of the Java virtual machine. Our principal interest is implementation's techniques of subtyping test and interface's method dispatch. Existing techniques are usually not perfect and conceptors can not agreed on one special technique.

Finally we present an implementation in a Java virtual machine of an aternative technique which. This technique uses perfect hashing to implement theses two mecanisms. The implementation is integrate in a Java virtual machine : J3. We present the differents aspects of the implementation, the results et perspectives. We also study of adapting a multiple inheritance langage in a Java virtual machine.