Étude de machines virtuelles Java existantes et adaptation au hachage parfait

Julien Pagès

UM2 - LIRMM

9 juillet 2013

Sous la direction de Roland DUCOURNAU

Spécialité : AIGLE

Plan

Contexte du stage

Sujet : étude de machines virtuelles Java existantes et adaptation au hachage parfait.

Contexte

- Langages de programmation à objets
- Systèmes d'exécution modernes : les machines virtuelles
- Le langage Java

Contexte du stage

Sujet : étude de machines virtuelles Java existantes et adaptation au hachage parfait.

Contexte

- Langages de programmation à objets
- Systèmes d'exécution modernes : les machines virtuelles
- Le langage Java

Caractéristiques principales d'une machine virtuelle

- Chargement dynamique
- Compilation à la volée
- Gestion automatique de la mémoire
- Synchronisation et multi-tâches

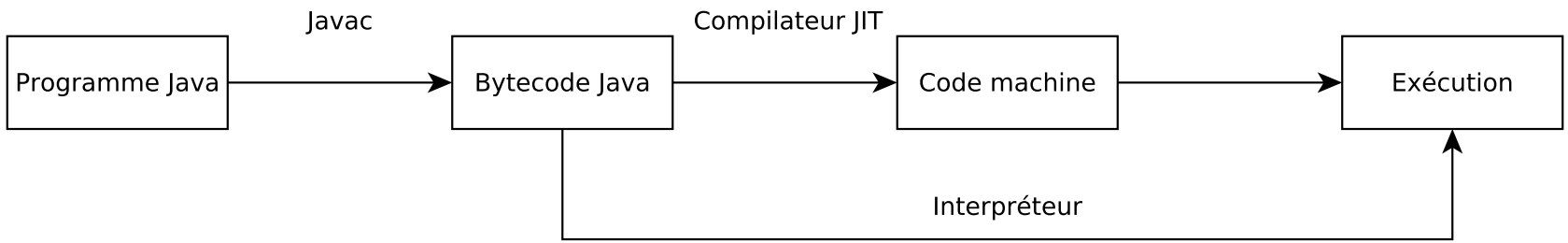
Java et sa machine virtuelle

Java a été créé en 1995 par James Gosling. La machine virtuelle Java (JVM) est à l'origine une spécification.

Caractéristiques

- Langages à objets en typage statique
- Langage à héritage simple mais avec des interfaces (déclarant uniquement des signatures de méthodes) en sous-typage multiple
- Un programme Java fonctionne sur la machine virtuelle Java (JVM)

Fonctionnement général de Java :



Objectifs de l'étude bibliographique

Objectifs

- État de l'art des JVM de recherche
- Étude des techniques employées pour réaliser le test de sous-typage et l'appel de méthode
- Choix d'une machine virtuelle pour le stage

Axes de travail

- Étude du fonctionnement de la JVM en général
- Étude de plusieurs implémentations représentatives des grandes familles d'implémentation
- État de l'art des techniques de sous-typage et d'appel de méthodes dans ces JVM
- Étude du hachage parfait

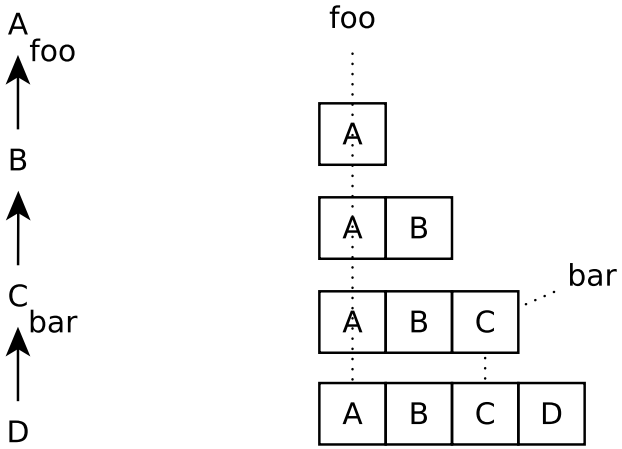
Problématique de l'appel de méthodes

Appels de méthodes dans la JVM

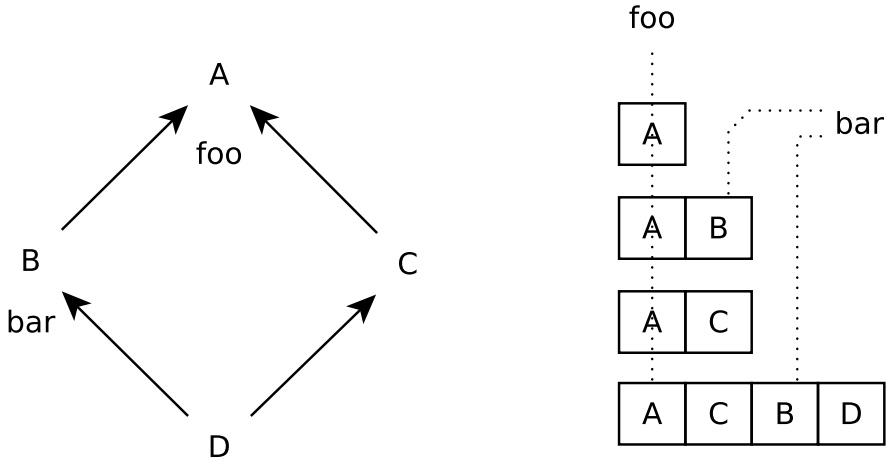
- *invokestatic* : Méthode statique d'une classe
- *invokespecial* : Méthode virtuelle invoquée statiquement
- *invokevirtual* : Méthode virtuelle d'une classe
- *invokeinterface* : Méthode introduite par une interface

Appel de méthode dans les langages à objets :

Sous-typage simple :



Sous-typage multiple :



Étude des techniques utilisées

Les langages sont très dépendants de l'implémentation efficace de trois mécanismes :

- Appel de méthode
- Test de sous-typage
- Accès aux attributs

Dans le cas de Java, seules les interfaces sont en sous-typage multiples

Les deux mécanismes avec des implémentations non-consensuelles sont donc :

- Appel de méthode d'une interface (opération *invokeinterface*)
- Test de sous-typage par rapport à une interface

Test de sous-typage

Différentes implémentations avec des objectifs différents :

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
Maxine	×	hachage parfait avec l'opération *modulo*

Appel de méthode d'une interface

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 espaces vides

Différentes techniques cherchant une optimisation en temps ou en espace.

Bilan sur les techniques utilisées

Bilan

- Beaucoup de techniques différentes utilisées
- Peu de techniques connues d'un point de vue algorithmique

Aucune ne remplit les conditions suivantes:

- Temps constant
- Espace linéaire dans la taille de la relation de spécialisation
- Compatible avec l'héritage multiple
- Compatible avec le chargement dynamique
- Compatible avec l'inlining (séquence de code courte)

Étude des machines virtuelles Java

- Lire les articles de références
- Schéma de présentation commun à chacune d'entre elle
- Les comparer afin d'en choisir une pour le stage

Choix d'un ensemble de JVM

- JVM en mode interprétée (SableVM)
- JVM de référence (Hotspot)
- JVM écrites en Java et orientées recherche (Jikes, Maxine)
- Supporte plusieurs langages : Open Runtime Platform
- Basée sur un framework : J3
- Cacao est souvent citée dans les JVM de recherche

Comparatif des JVM

Tableau comparatif des JVM :

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

Choix de la JVM

J3 du projet VMKIT (INRIA, LIP6) a été choisie pour :

- Projet de recherche
- Simplicité, peu de lignes de codes, maintenance

VMKIT est un framework pour développer des machines virtuelles. Il apporte :

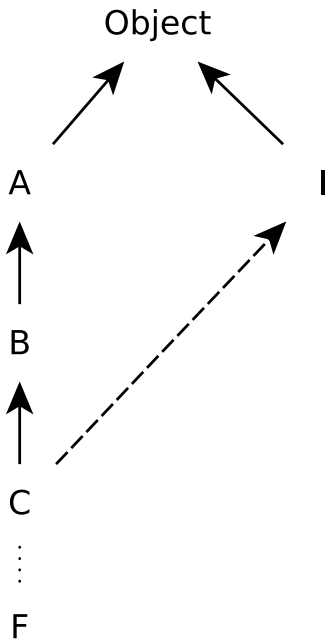
- Un gestionnaire de mémoire : MMTK
- Un gestionnaire de threads : POSIX
- Un compilateur JIT : LLVM

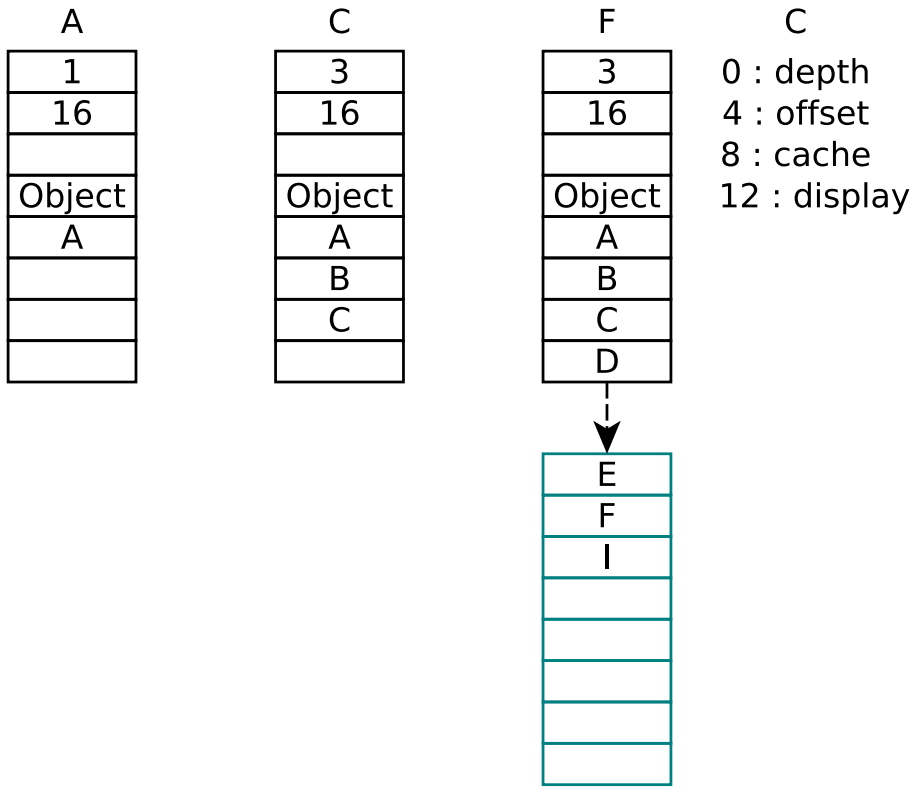
LLVM est une suite d'outils de compilation contenant entre autres :

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

Techniques actuelles dans J3

Sous-typage dans J3 avec la technique de Hotspot :

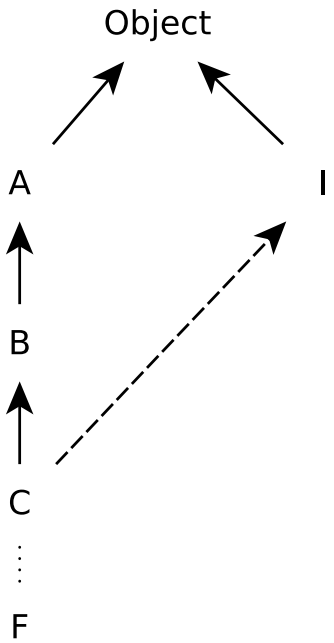


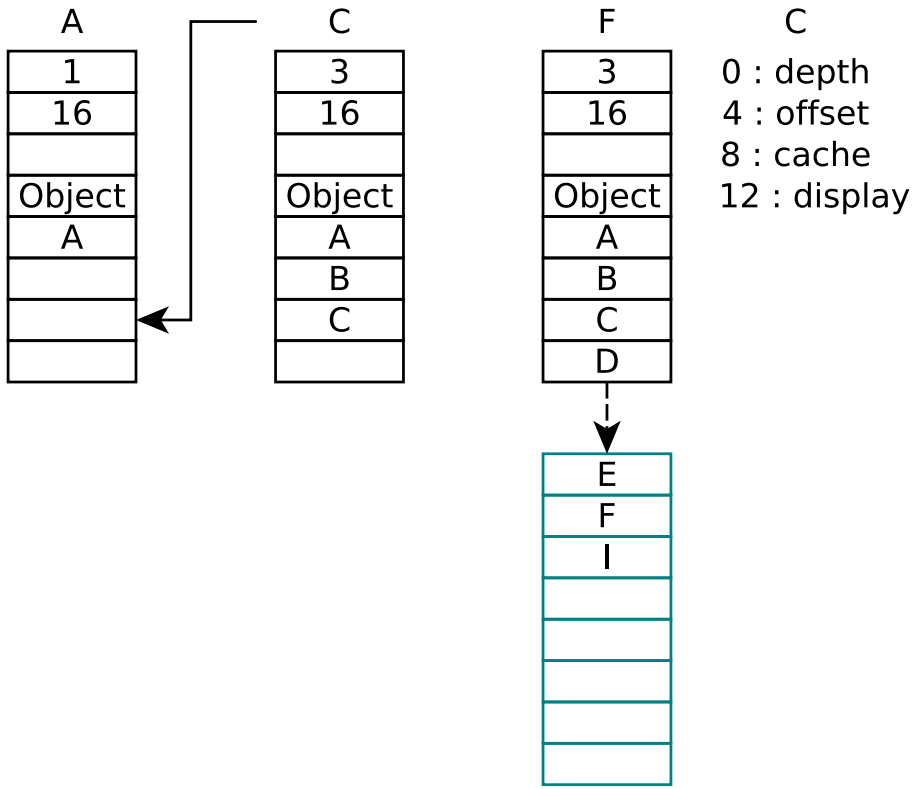


Techniques actuelles dans J3

Sous-typage dans J3 avec la technique de Hotspot :

```
A x = new A();
x instanceof C
```

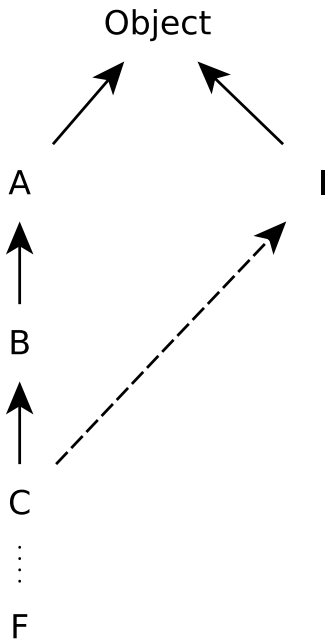


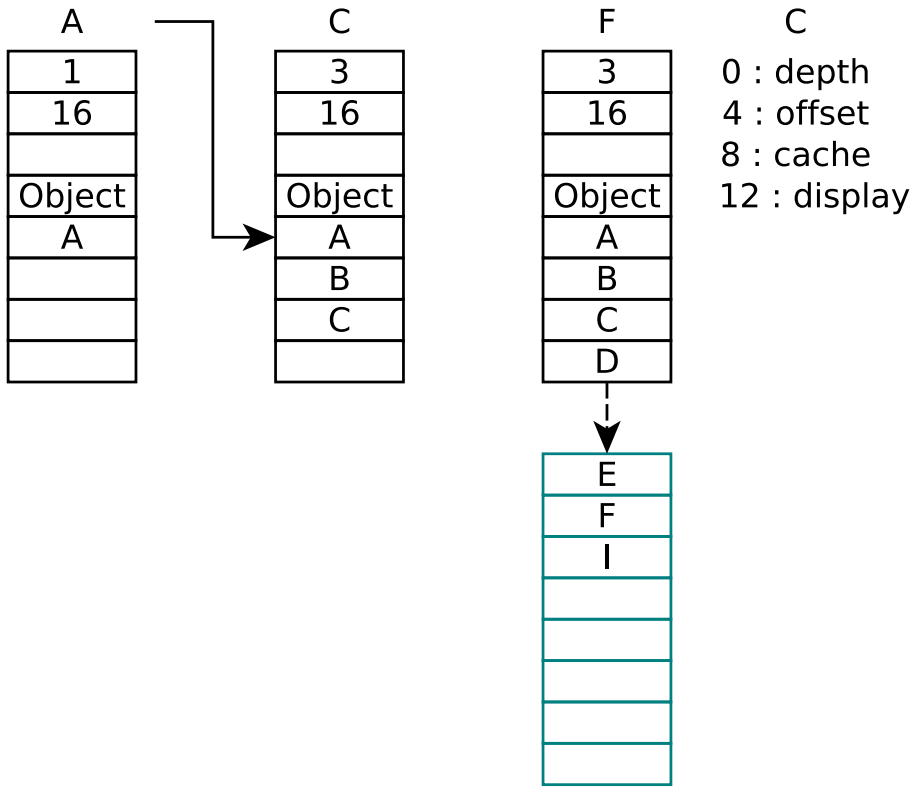


Techniques actuelles dans J3

Sous-typage dans J3 avec la technique de Hotspot :

```
C x = new C();
x instanceof A
```

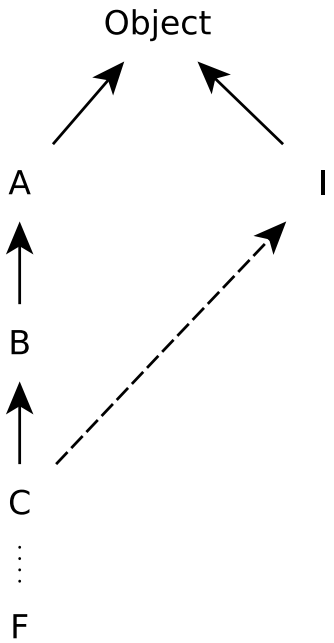


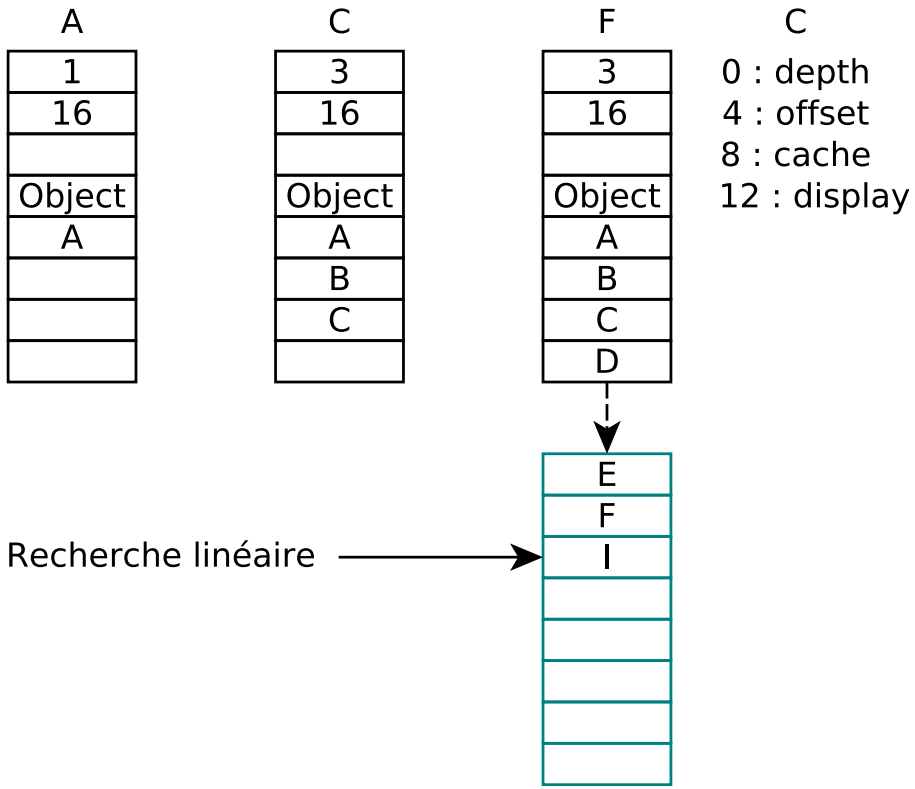


Techniques actuelles dans J3

Sous-typage dans J3 avec la technique de Hotspot :

```
I x = new C();
x instanceof I
```





Techniques actuelles dans J3

Appel d'une méthode d'une interface dans J3 :

```
I x = new A();
x.foo();
```

L'appel de méthode est réalisé grâce à une table de hachage contenant les méthodes des interfaces :

- Les signatures des méthodes sont hachées
- Résolution des conflits de hachage par **separate chaining**
- La gestion des collisions est directement compilée

Le hachage parfait

Hachage parfait

- Le hachage parfait est un hachage sans collision d'un ensemble d'entiers
- Le masque de hachage est calculé pour former une fonction injective entre l'ensemble d'identifiants et la fonction binaire *AND*

Le hachage parfait

Hachage parfait

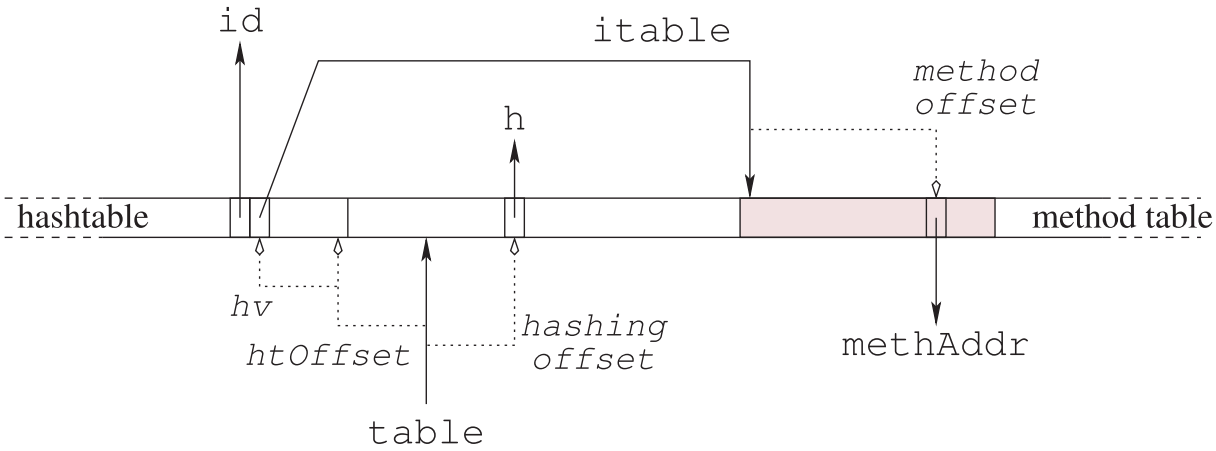
- Le hachage parfait est un hachage sans collision d'un ensemble d'entiers
- Le masque de hachage est calculé pour former une fonction injective entre l'ensemble d'identifiants et la fonction binaire *AND*

Numérotation parfaite

- Optimisation de l'espace utilisé pour les tables de hachage
- Lors de l'attribution d'un nouvel identifiant : on choisit celui qui convient le mieux aux identifiants déjà présents dans les super-classes

Utilisation du hachage parfait

Utilisation concrète du hachage parfait dans une JVM :



Le hachage parfait est une technique permettant :

- Un test de sous-typage et un appel de méthode rapides et en temps constant
- Une consommation mémoire raisonnable (linéaire dans la taille de la relation de spécialisation).
- Une compatibilité avec l'héritage multiple et le chargement dynamique

Implémentation du hachage parfait

L'implémentation du hachage parfaite a été réalisée en plusieurs étapes :

1. Codage en C du hachage parfait
2. Codage de la numérotation parfaite
3. Numérotation de chaque interface
4. Création des structures appropriées et utilisation du hachage parfait pour le sous-typage
5. Groupement des méthodes des interfaces et utilisation du hachage parfait pour l'appel de méthode

Numérotation des interfaces

Lors de la création d'une méta-interface en C++, plus précisément lors de la création de sa table des méthodes :

Numérotation

- Récupération des identifiants des super-interfaces
- Attribution du premier identifiant disponible via la numérotation parfaite

Construction des tables de hachage

- Allocation des tables à la taille retournée par le hachage parfait
- Remplissage des tables avec uniquement les identifiants des super-interfaces

Test de sous-typage

Le test de sous-typage est possible par rapport à des cibles qui sont interfaces via le hachage parfait.

Si i est un objet dont le type statique est une interface :

```
B b = new C();
I i = new A();

if(b instanceof i)
...
```

```
// 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);
}
```

Appel de méthode

Prérequis

- Grouper les implémentations des méthodes introduites par chaque interface
- Chaque entrée de la table de hachage pointe vers le bloc de méthodes
- Récupérer les pointeurs dans les super-classes pour les implémentations héritées

Opération *invokeinterface* :

```
I i = new A();
i.foo();
```

Appel de méthode

Code final de l'appel de méthode d'une interface

```
// 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];
```

Résultats

Résultats

- État de l'art sur les machines virtuelles Java de recherche
- État de l'art sur plusieurs techniques d'implémentations utilisées dans les JVM (sous-typage, appel de méthode)
- J3 modifiée utilise le hachage parfait pour le sous-typage par rapport à des interfaces
- J3 modifiée propose un début d'implémentation pour l'appel de méthodes avec le hachage parfait (pas complet à cause de l'unification des méthodes en Java)

Difficultés et perspectives

Difficultés

- Documentation de VMKIT/J3 inexistante
- Nécessité de traduire toute la JVM J3 en structures LLVM
- Le hachage parfait impose des contraintes laborieuses à implémenter
- Le langage Java possède des incohérences historiques qui complexifient l'implémentation

Perspectives

- Intégration d'un méta-modèle correct pour surmonter ces difficultés
- Modification de benchmarks pour forcer plus d'appels de méthode des interfaces

Ouverture

Travail plutôt dans une optique préparatoire à la thèse : étude de l'adaptation de PRM/NIT sur une JVM :

- Traitement de la généricité
- Extension du *bytecode* Java
- Traitement de l'héritage multiple

Conclusion

Bilan

- Beaucoup de techniques souvent perfectibles utilisées pour ces deux mécanismes dans les JVM
- Le hachage parfait est une technique efficace et élégante généralement mieux maîtrisée que les mécanismes actuels
- Il est possible de l'utiliser dans les JVM mais un méta-modèle cohérent faciliterait l'implémentation

Merci de votre attention