# Evaluation de l'efficacité des implémentations de l'héritage multiple en typage statique

Floréal Morandat*, Roland Ducournau*, Jean Privat**

*LIRMM – CNRS et Université Montpellier II
161 rue Ada Montpellier – 34392 Cedex 5 France
{morandat,ducour}@lirmm.fr,
http://www.lirmm.fr/~{morandat,ducour}/
**Université du Québec à Montréal
privat.jean@uqam.ca
http://www.info2.uqam.ca/~privat/

**Résumé.** La programmation par objets présente une apparente incompatibilité entre trois termes : l'héritage multiple, l'efficacité et l'hypothèse du monde ouvert — en particulier, le chargement dynamique. Cet article présente des résultats d'expérimentations exhaustives comparant l'efficacité de différentes *techniques d'implémentation* (coloration, BTD, hachage parfait, ...) dans le contexte de différents *schémas de compilation* (de la compilation séparée avec chargement dynamique à la compilation purement globale). Les tests sont effectués avec et sur le compilateur du langage Prm. Ils confirment pour l'essentiel les résultats théoriques antérieurs tout en montrant une sur-additivité marquée des surcoûts. Les schémas d'optimisation globale démontrent un gain significatif par rapport à la coloration qui fait fonction de référence. Des techniques comme la simulation des accesseurs ou le hachage parfait entraînent un surcoût limité, mais la combinaison des deux double le surcoût total.

## 1 Introduction

*L'hypothèse du monde ouvert* (OWA pour *Open World Assumption*) représente certainement le contexte le plus favorable pour obtenir la *réutilisabilité* prônée par le génie logiciel. Une classe doit pouvoir être conçue, programmée, compilée et implémentée indépendamment de ses usages futurs et en particulier de ses sous-classes. C'est ce qui permet d'assurer la compilation séparée et le chargement dynamique.

Cependant l'héritage multiple — ou ses variantes comme le sous-typage multiple d'interfaces à la Java— s'est révélé difficile à implémenter sous l'hypothèse du monde ouvert. En typage statique, seule *l'hypothèse du monde clos* (CWA) permet d'obtenir la même efficacité qu'en héritage simple, c'est-à-dire avec une implémentation en temps constant nécessitant un espace linéaire dans la taille de la relation de spécialisation. Les deux langages les plus utilisés actuellement illustrent bien ce point. C++, avec

le mot-clef `virtual` pour l'héritage, procure une implémentation pleinement réutilisable, en temps constant — bien que pleine d'ajustements de pointeurs — mais qui nécessite un espace cubique dans le nombre de classes (dans le pire des cas). En Java, où l'héritage multiple est restreint aux interfaces, il n'existe pas à notre connaissance d'implémentation des interfaces en temps constant (Alpern et al., 2001; Ducournau, 2008). En typage dynamique, même l'héritage simple pose des problèmes et nous nous restreindrons ici au typage statique.

Dans cet article, nous distinguons l'implémentation qui concerne la *représentation* des objets et la compilation qui calcule cette représentation. Au total, l'exécution des programmes à objets met en jeu une ou plusieurs *techniques d'implémentation* dans le contexte de différents *schémas de compilation*. L'implémentation est concernée par les trois mécanismes de base des langages objets — accès aux attributs, invocation de méthodes et test de sous-typage. Le schéma de compilation constitue la chaîne de production de l'exécutable : génération de code, édition de liens, chargement.

L'objectif de ce travail est d'évaluer de manière réaliste et objective l'impact effectif des schémas de compilation et des techniques d'implémentation sur l'efficacité d'un programme objet significatif en comparant deux à deux les techniques ou les schémas, *toutes choses égales par ailleurs*. Ces expérimentations sont réalisées sur le compilateur de Prm, un langage objet qui supporte l'héritage multiple et qui propose une notion de module et de raffinement de classe (Privat et Ducournau, 2005b). Grâce à cela, $\text{Prm}_\mathcal{C}$ son compilateur autogène (écrit en Prm) est un programme modulaire et remplacer une technique par une autre peut se faire à moindre coût. Les premières expérimentations décrites ici présentent le *temps d'exécution* de la compilation d'une version de $\text{Prm}_\mathcal{C}$ par différentes versions du même compilateur obtenues en faisant varier techniques d'implémentation et schémas de compilation.

Cet article décrit d'abord des techniques d'implémentation des mécanismes objet, puis des schémas de compilation — de la compilation séparée qui permet de compiler le code modulairement à la compilation globale qui permet d'effectuer de nombreuses optimisations, ainsi que certains schémas intermédiaires qui ont été peu étudiés. Une première série de comparaisons et d'évaluation *a priori* est présentée. Nous présentons ensuite une série de tests réalisés sur $\text{Prm}_\mathcal{C}$, afin de mesurer l'impact réel des différents schémas et techniques sur de vrais programmes — en l'occurrence $\text{Prm}_\mathcal{C}$ lui-même. Enfin nous commenterons ces résultats avant de parler des travaux connexes, puis nous conclurons en détaillant les perspectives de ce travail.

# 2 Implémentation et compilation

Cette section présente les différentes techniques d'implémentation et les schémas de compilation que nous considérons dans cet article. Le lecteur intéressé est renvoyé à (Ducournau, 2002) pour une synthèse plus générale.

## 2.1 Techniques d'implémentation

Nous présentons d'abord la technique de référence de l'héritage simple puis les différentes techniques d'implémentation de l'héritage multiple que nous considérons

ici. Nous parlerons principalement de l'appel de méthode, l'implémentation des deux autres mécanismes pouvant généralement se déduire de celui-ci.

### 2.1.1 Technique de base : héritage simple et typage statique

Dans le cas de l'héritage simple, le graphe de spécialisation d'un programme est une arborescence. Il n'existe donc qu'un chemin reliant une classe (sommet dans le graphe) à la classe racine. En typage statique, cette propriété donne lieu à une implémentation simple et efficace de l'héritage simple. Il suffit pour cela de concaténer les méthodes introduites par chacune des classes dans l'ordre de spécialisation pour construire les tables de méthodes. L'opération similaire appliquée aux attributs permet de représenter les instances (Figure 1-a). Le test de sous-typage de Cohen (1991) s'intègre dans la table de méthodes suivant le même principe.

Nous considérons cette implémentation comme celle de référence car elle rajoute le minimum d'information nécessaire à l'exécution d'un programme dans les diverses tables. De plus, elle vérifie deux invariants : *de position*, chaque méthode (resp. attribut) a une position dans la table des méthodes (resp. l'objet) invariante par spécialisation, donc indépendante du type dynamique du receveur ; *de référence*, la référence à un objet est indépendante du type statique de la référence.

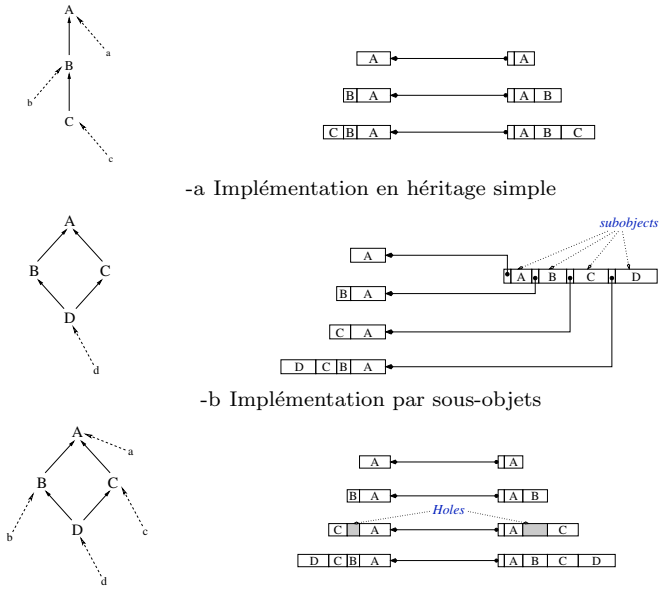
En héritage simple, il suffit de connaître la super-classe pour calculer l'implémentation d'une classe (OWA). La taille totale des tables est *linéaire* dans le cardinal de la relation de spécialisation, donc, dans le pire des cas, *quadratique* dans le nombre de classes.

### 2.1.2 Sous-objets (SO)

Dans le cas de l'héritage multiple, ces invariants sont trop forts car deux classes incomparables, peuvent avoir une sous-classe commune (par exemple B et C dans la figure 1-b). La technique de base ne peut donc plus être utilisée.

L'implémentation de C++ détaillée dans Ellis et Stroustrup (1990) repose entièrement sur les types statiques des références. La représentation d'un objet est faite par concaténation des *sous-objets* de toutes les super-classes de sa classe (elle-même comprise). Pour une classe donnée, un sous-objet est un pointeur sur sa table de méthodes suivi des attributs *introduits* par cette classe. Chaque table de méthodes contient l'ensemble des méthodes *connues* par le type statique du sous-objet. Une référence à un objet pointe sur le sous-objet correspondant au type statique de la référence et toutes les opérations polymorphes sur un objet nécessitent un ajustement de pointeur. Cet ajustement est dépendant du type dynamique de l'instance pointée, donc la table des méthodes doit contenir aussi les décalages à utiliser. Dans le pire des cas, la taille totale des tables est ainsi *cubique* dans le nombre de classes. Cette implémentation reste compatible avec le chargement dynamique (OWA).

En C++, cette implémentation suppose que le mot-clef `virtual` annote chaque super-classe. L'absence de `virtual` dans les clauses d'héritage se traduit par une implémentation simplifiée qui entraîne un risque d'héritage répété. La majorité des programmes utilisent peu ce mot-clef et ils ne souffrent donc pas du surcoût entraîné par




A chaque exemple de hiérarchie de classes $(A, B, C, ..)$ est associée l'implémentation correspondante d'instances $(a, b, c, ..)$. Les objets sont à droite, les tables de méthodes à gauche, inversées pour éviter les croisements. Dans les tables, les étiquettes de classe désignent le groupe de méthodes ou d'attributs *introduits* par la classe.

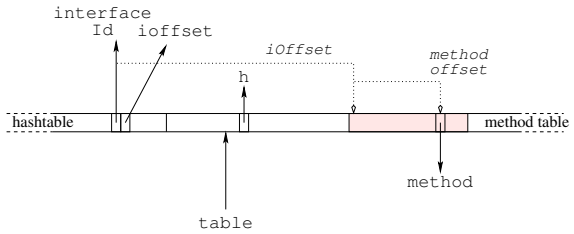
FIG. 1 – *Différentes techniques d'implémentation*

l'héritage multiple et le chargement dynamique, mais c'est au prix d'une réutilisabilité limitée (CWA).

### 2.1.3 Coloration (CM/CA)

La coloration a été initialement proposée par Dixon et al. (1989), Pugh et Weddell (1990) et Vitek et al. (1997) pour chacun des trois mécanismes de base. Ducournau (2006) montre l'identité des trois techniques et fait la synthèse de l'approche. L'idée principale est de maintenir en héritage multiple les invariants de l'héritage simple, mais sous l'hypothèse du monde clos. Comme les éléments doivent avoir une position invariante par spécialisation, il faut laisser certains trous dans les tables, par exemple dans celle de C (Figure 1-c). Pour obtenir un résultat efficace, il faut minimiser la taille des tables (ou le nombre de trous).

La technique de la coloration peut être utilisée pour les trois mécanismes objet et le code généré pour chacun de ces mécanismes ne dépend que de la couleur de l'entité accédé. Seul le calcul des couleurs nécessite de connaître la hiérarchie de classes (CWA). Dans la suite, CM désigne l'application de la coloration aux méthodes et au test de sous-typage et CA l'application à la représentation des instances. D'autres techniques




L'objet pointe (par `table`) à l'indice 0 de la table de méthodes. Les indices positifs contiennent les adresses des méthodes dans l'implémentation de l'héritage simple, où les méthodes sont groupées par interface d'introduction. Les indices négatifs contiennent la table de hachage dont le paramètre `h` est aussi dans la table. Une entrée de la table est formée de l'identifiant de l'interface (`interfaceId`) pour le test de sous-typage et de la position relative (`iOffset`) du groupe de méthodes introduites par l'interface.

FIG. 2 – *Hachage parfait en* JAVA

de compression de tables existent, par exemple *row displacement* (Driesen, 2001), mais elles ne s'appliquent pas à la représentation des objets.

### 2.1.4 Simulation des accesseurs (SA)

La simulation des accesseurs, proposée par Myers (1995) et Ducournau (2002), permet de réduire l'accès aux attributs à l'envoi de message, en rajoutant une indirection par la table des méthodes. Les attributs sont groupés par classe d'introduction et la position de chaque groupe est incluse dans la table des méthodes comme si cela en était une. La simulation des accesseurs s'applique à n'importe quelle technique d'implémentation des méthodes — elle suppose juste qu'un attribut soit introduit par une classe unique, comme en typage statique. Nous la considérons ici couplée à la coloration de méthodes comme alternative à la coloration d'attributs qui peut engendrer un nombre de trous trop important dans certaines classes, voir (Ducournau, 2006).

### 2.1.5 Arbres binaires d'envoi de messages (BTD)

Toutes les techniques présentées jusqu'à présent utilisent des tables pour invoquer les méthodes. L'idée des *binary tree dispatch* (BTD), proposée par Zendra et al. (1997), est de remplacer les tables par une série de tests d'égalité de types qui déterminent la méthode à appeler. L'appel de méthode lui-même est implémenté comme un arbre binaire équilibré de comparaison de types dont les feuilles sont les appels statiques de méthodes. En pratique, grâce au cache d'instructions et aux prédictions de branchements des processeurs modernes, si la profondeur de l'arbre n'excède pas trois niveaux — donc avec huit feuilles au maximum — le nombre moyen de cycles pris pour ces comparaisons est inférieur au temps d'accès par l'indirection d'une table de méthodes. On notera par la suite $\text{BTD}_i$ un BTD de profondeur $i$. Les $\text{BTD}_0$ correspondent aux appels statiques.

### 2.1.6 Hachage parfait (PH)

Le hachage parfait est une optimisation en temps constant des tables de hachage pour des ensembles immutables (Czech et al., 1997). Ducournau (2008) propose de l'utiliser pour le test de sous-typage et l'envoi de message, dans le cas particulier des interfaces en JAVA (Figure 2). Cette technique nécessite une indirection supplémentaire avant l'appel de la méthode. Elle peut être aussi appliquée aux attributs avec la simulation des accesseurs. A notre connaissance, cette technique est la seule alternative à l'implémentation par sous-objets de C++ qui soit en temps constant et incrémentale (OWA). Cependant sa constante de temps pour l'appel de méthode est à peu près le double de celle de l'héritage simple. Deux fonctions de hachage ont été étudiées : modulo, le reste de la division entière (notée `mod`) et la conjonction bit-à-bit (notée `and`). Les deux ne nécessitent qu'une instruction machine mais, alors que `and` prend un seul cycle d'horloge, la latence de la division entière peut attendre 20 ou 25 cycles, en particulier sur les processeurs comme le Pentium où la division entière est prise en charge par l'unité de calcul flottant. Par ailleurs, les expérimentations effectuées sur PH-`and` et PH-`mod` montre que le premier nécessite des tables beaucoup plus grandes que le second. Le hachage parfait semble donc conduire à un choix espace-temps difficile.

## 2.2 Schémas de compilation

Les schémas de compilation constituent la chaîne de production d'un exécutable. Elle est composée de plusieurs phases distinctes qui incluent de façon non limitative la compilation et le chargement des unités de code ainsi que l'édition de liens. La compilation nécessite au minimum le code source de l'unité considérée et des informations sur toutes les classes qu'elle utilise (super-classes par exemple). Ces informations sont contenues dans le *modèle externe*, qui est l'équivalent des fichiers d'en-têtes (.h) en C et peut être extrait automatiquement comme en JAVA. La distinction entre tous ces schémas n'est pas toujours très tranchée. En particulier les compilateurs adaptatifs (Arnold et al., 2005) font appel à des techniques globales (cf. 2.2.2) dans un cadre de chargement dynamique (cf. 2.2.1) au prix d'une recompilation dynamique quand les hypothèses initiales se trouvent infirmées. Dans cet article nous nous intéressons seulement aux schémas sans recompilation à l'exécution.

### 2.2.1 Compilation séparée et chargement dynamique (D)

La compilation séparée, associée au chargement dynamique, est un schéma de compilation classique illustré par les langages LISP, SMALLTALK, C++, JAVA. Chaque unité de code est compilée séparément, donc indépendamment des autres unités, puis l'édition de liens rassemble les morceaux de façon incrémentale au cours du chargement (OWA). Ce schéma permet de distribuer des modules déjà compilés sous forme de boites noires. De plus, comme les unités sont traitées de manière indépendante, la recompilation des programmes peut se restreindre aux unités modifiées directement ou indirectement. Malheureusement, si l'on exclut des recompilations dynamiques, ce schéma ne permet aucune optimisation des mécanismes objet sur le code produit — librairies, programmes — et ne permet d'utiliser que peu d'implémentations pour l'hé-

ritage multiple — seuls les sous-objets de C++ et le hachage parfait sont compatibles. La coloration a été essayée, dans le cas particulier du test de sous-typage (Palacz et Vitek, 2003), mais le chargement dynamique impose un recalcul et des indirections qui sont coûteuses, aussi bien au chargement qu'à l'exécution.

### 2.2.2 Compilation globale (G)

Ce schéma de compilation est utilisé par EIFFEL[1] et constitue le cadre de toute la littérature sur l'optimisation des programmes objet autour des langages SELF, CECIL, EIFFEL (Zendra et al., 1997; Collin et al., 1997). Il suppose que toutes les unités de code soient connues à la compilation (CWA) y compris le point d'entrée du programme. Il est donc possible d'effectuer une *analyse de types* (Grove et Chambers, 2001), de déterminer l'ensemble des *classes vivantes* (effectivement instanciées par le programme). Grâce à cela, on peut réduire la taille des exécutables en éliminant le *code mort* et compiler les envois de messages par des BTD — en effet le nombre de types à examiner pour chaque appel devient raisonnable — voire des appels statiques. La coloration peut alors s'utiliser en complément, pour les sites d'appels dont le degré de polymorphisme reste grand. Bien que ce schéma permette de générer le code le plus efficace il présente de nombreux inconvénients. L'ensemble des sources doit être disponible, ce qui empêche la distribution de *modules* pré-compilés. De plus, chaque modification sur le code, même mineure, entraîne une recompilation globale. Inversement, si la modification porte sur le code mort, elle peut ne même pas être vérifiée.

### 2.2.3 Compilation hybride

Il est possible de combiner la compilation séparée (OWA) avec une édition de liens globale (CWA). C'est ainsi que la majorité des programmes utilisent C++. Dans le cas de C++, l'éditeur de liens n'a besoin d'aucune spécificité, mais les approches suivantes nécessitent un calcul spécifique avant l'édition de liens proprement dite. Une alternative que nous ne considérons pas plus serait que la compilation génère le code idoine qui effectuerait ce calcul lors du lancement du programme.

**Compilation séparée et édition de liens globale (S).** Pugh et Weddell (1990) proposaient initialement la coloration dans un cadre de compilation séparée, où le calcul des couleurs serait fait à l'édition de liens : seuls les modèles externes de toute la hiérarchie sont nécessaires. Comme la coloration est une technique intrinsèquement globale, une fois l'édition de liens effectuée, aucune nouvelle classe ni aucune nouvelle méthode ne peuvent être ajoutées sans recalculer tout ou partie de la coloration. Cette version de la compilation séparée est donc incompatible avec le chargement dynamique mais elle permet d'en conserver de nombreuses qualités — distribution du code compilé, recompilation partielle. En revanche, ce schéma ne s'applique pas aux BTD, car le code du BTD doit être généré à la compilation en connaissant la totalité des classes.

[1] Bien que les spécifications des langages de programmation soient en principe indépendantes de leur implémentation, de nombreux langages sont en fait indissociables de celle-ci. Par exemple, la covariance non contrainte d'EIFFEL n'est pas implémentable efficacement sans compilation globale et la règle des *catcalls* n'est même pas vérifiable en compilation séparée (Meyer, 1997). Il en va de même pour C++ et son implémentation par sous-objets ainsi que pour JAVA et le chargement dynamique.

**Compilation séparée avec optimisations globales (O).** Privat et Ducournau (2004, 2005a) proposent un schéma de compilation séparée, permettant l'utilisation d'optimisations globales à l'édition de liens. C'est une généralisation du schéma précédent, qui nécessite certains artifices supplémentaires. Lorsqu'une unité de code est compilée, le compilateur produit, en plus du code compilé et du *modèle externe*, un *modèle interne* qui contient l'information relative à la circulation des types dans l'unité compilée. La phase d'édition de liens nécessite l'ensemble des unités compilées, y compris le point d'entrée du programme, ainsi que l'ensemble des modèles internes et externes. Grâce à ces modèles, on peut procéder à des analyses de types en se servant du flux de types contenu dans les modèles internes. La majorité des optimisations proposées dans un cadre de compilation globale (analyses de types, BTD, ...) deviennent possibles dans ce schéma hybride. A la compilation, chaque site d'appel est lui-même compilé comme un appel à un symbole unique. Lors de l'édition de liens, le code correspondant est généré suivant les spécificités du site d'appel : appels statiques (site monomorphe), BTD (site oligomorphe), coloration (site mégamorphe) — on appelle cela un *thunk* comme dans l'implémentation de C++ (Ellis et Stroustrup, 1990). Mais la phase d'édition de liens travaille toujours suivant l'hypothèse du monde clos, et le chargement dynamique reste exclu. Un schéma voisin avait été proposé par Boucher (2000) dans un cadre de programmation fonctionnelle.

**Compilation séparée des bibliothèques et globale du programme (H).** Une dernière approche consiste à compiler les bibliothèques de manière séparée comme dans le schéma précédent. Le programme lui-même est compilé de manière globale en effectuant toutes les optimisations possibles : schéma global (G) sur le programme et schéma séparé avec optimisations globales (O) sur les bibliothèques.

## 2.3 Comparaisons et compatibilités

La tableau 1 décrit la compatibilité des schémas de compilation avec les techniques d'implémentation (colonnes de gauche). Le reste du tableau décrit l'efficacité a priori des techniques en se basant sur les études antérieures (Ducournau, 2002). L'espace est considéré suivant trois aspects : le code, les tables statiques, la mémoire dynamique (objets). Le temps est considéré à l'exécution, à la compilation et au chargement. La notation va de « − − » à « +++ », « ++ » représentant l'efficacité de la technique de référence en héritage simple. Le cas des BTD est particulier : pour la taille du code et le temps d'exécution, la technique va du meilleur (appel statique) au pire (arbre de taille 100 ou 1000) et est notée « ∗∗∗ ». L'objectif des expérimentations qui suivent est de vérifier empiriquement les évaluation théoriques.

Le tableau 2 classe les schémas de compilation en fonction des critères imposés par le génie logiciel. La capacité à réutiliser et assembler des unités de code conçues indépendamment est présentée dans la ligne *réutilisation*. La *distribution* représente la capacité à distribuer les unités de code indépendamment les unes des autres tout en protégeant le contenu de l'unité considérée (boîte noire). La *vérification modulaire* permet de tester les unités de code de manière atomique, elle permet de limiter le nombre de *bouchons d'intégration* (Traon et Baudry, 2006) lors des tests unitaires.

TAB. 1 – *Compatibilité avec les schémas et efficacité des techniques d'implémentation*

| | Schéma | | | | | Espace | | | Temps | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | D | S | O | H | G | Code | Sta. | Dyn. | Exé. | Compil. | Chargement |
| SO | ∗ | ∗ | ∗ | ∗ | ∗ | – | – – | – | – | ++ | ++ |
| PH | • | ∗ | ∗ | ∗ | ∗ | – | – | ++ | – | ++ | + |
| CM | × | • | • | • | • | ++ | ++ | | ++ | + | |
| CA | × | • | • | • | • | ++ | | – | ++ | + | |
| SA | ∗ | ∗ | • | ∗ | ∗ | + | + | ++ | – | + | |
| BTD | × | × | • | • | • | ∗ ∗ ∗ | +++ | ++ | ∗ ∗ ∗ | ++ | |

• : Compatible, ∗ : Compatible mais non testé, × : Incompatible

TAB. 2 – *Qualités et défauts des schémas de compilation*

| | D | S | O | H | G |
|---|---|---|---|---|---|
| Réutilisation | ++ | + | + | + | – – |
| Distribution | ++ | ++ | ++ | ++ | – – |
| Vérification modulaire | ++ | ++ | ++ | ++ | – – |
| Implémentation efficace | – – | + | + | ++ | +++ |

# 3 Expérimentations et discussion

Les expérimentations ont été réalisées avec le langage PRM, un langage expérimental modulaire (Privat et Ducournau, 2005b) qui a été conçu en grande partie dans ce but. Les divers schémas et techniques présentés plus haut ont été testés avec les restrictions suivantes :

- les spécifications de PRM, en particulier le raffinement de classes, le rendent absolument incompatible avec le chargement dynamique ; le hachage parfait a donc été implémenté en compilation séparée avec édition de liens globale (S) mais le code généré est exactement celui qui aurait été généré avec chargement dynamique (D).
- pour les BTD, l'idée est de baser les tests sur l'adresse des tables de méthodes. La disposition en mémoire des tables doit donc être connue lors de la génération des BTD. Comme il se trouve que `gcc` prend des libertés en réordonnant les tables, seules les $\text{BTD}_0$ (appels monomorphes) ont été testées.

## 3.1 Expérimentations

Le dispositif d'expérimentation (Figure 3) est constitué d'un programme de test $P$, qui est compilé par une variété de compilateurs $C_i, i \in [1..k]$ (ou par le même compilateur avec une variété d'options). Le temps d'exécution de chacun des exécutables $P_i$ obtenus est ensuite mesuré sur une donnée commune $D$. Les $C_i$ représentent différentes versions de $\text{PRM}_C$. Comme le compilateur est le seul programme significatif disponible en PRM, le programme de test $P$ est lui aussi le compilateur, tout comme la donnée $D$.

Les mesures temporelles sont effectuées avec la fonction UNIX `times(2)` qui comptabilise le temps utilisé par un processus indépendamment de l'ordonnanceur système (obligatoire à cause des processeurs multi-cœur). Les mesures du temps et les statis-

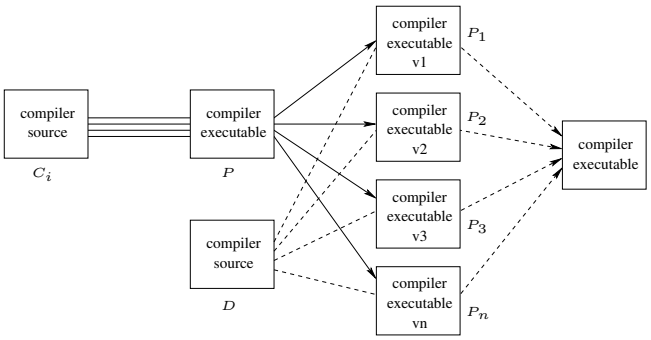



FIG. 3 – *Protocole d'expérimentation*

tiques dynamiques sont effectuées dans des passes différentes afin de ne pas les influencer réciproquement.

Ces tests ont été réalisés sous Ubuntu 8.4 et `gcc` 4.2.4 sur des processeurs de la famille des Pentium d'Intel. Chaque test est la moyenne de dix exécutions au moins. Un compilateur $P_i$ est généré pour chaque exécution pour tenir des éventuelles variations dans l'implantation mémoire. Malgré cela, les variations observées ne dépassent pas 1 à 2%. Un test complet représente donc plusieurs heures de calcul. Les tests ont été faits sur différents processeurs de même architecture pour confirmer la régularité du comportement observé.

La table 3 donne des statistiques sur le programme de test ($P$), d'un point de vue statique (nombre d'éléments du programme) et dynamique (nombre d'accès), lorsqu'il est appliqué à la donnée $D$.

TAB. 3 – *Statistiques sur le programme de test P*

| | Statique | Dynamique (sur $D$) |
|---|---|---|
| Modules | 75 | |
| Classes | 511 | |
| Méthodes | 2518 | |
| Méthodes définies | 4262 | |
| Attributs | 609 | |
| Appels de méthodes | 14551 | 2600 M |
| Appels monomorphes | 3969 | 1657 M |
| Appels megamorphes | 1264 | 940 M |
| Lecture d'attributs | 3114 | 287 M |
| Affectation d'attributs | 1192 | 13 M |
| Instanciations | 6091 | 34 M |

## 3.2 Résultats

La table 4 présente les temps d'exécution de différents couples technique-schéma, sous la forme du rapport $(t_T - t_{\text{ref}})/t_{\text{ref}}$, où $t_T$ est le temps de la version $T$ considérée, et $t_{\text{ref}}$ est le temps de la version de référence, c'est-à-dire le schéma S avec coloration (CM-

| processeur | | Pentium Xeon | | | Pentium Duo | | | Pentium Xeon | | |
|---|---|---|---|---|---|---|---|---|---|---|
| fréquence | | 1.8 GHz | | | 2.8 GHz | | | 2.8 GHz | | |
| cache | | 512 K | | | 1024 K | | | 2048 K | | |
| temps de référence | | 109.1s | | | 55.7s | | | 52.8s | | |
| technique | schéma | CA | SA | SA/CA | CA | SA | SA/CA | CA | SA | SA/CA |
| CM-BTD$_0$ | G | -9 | -1.8 | 7.9 | -14.1 | -7.4 | 7.7 | -15.4 | -1 | 17 |
| CM-BTD$_0$ | O | -2.9 | 2.3 | 5.4 | -3.1 | 2.6 | 5.9 | -3.4 | 3.6 | 7.3 |
| CM | S | 0 | 5 | 5 | 0 | 6.1 | 6.1 | 0 | 10 | 10 |
| PH-`and` | D | 7.3 | 21.4 | 13.2 | 4.7 | 25.1 | 19.5 | 5.8 | 30.5 | 23.3 |
| PH-`mod` | D | 46.2 | 194.8 | 101.6 | 63.2 | 226.8 | 100.3 | 70.9 | 246.3 | 102.6 |

Chaque sous-table détaille les résultats pour un processeur particulier, en donnant d'abord ses caractéristiques et le temps de compilation de référence (en secondes). Tous les autres chiffres sont des pourcentages. Chaque ligne décrit une technique d'invocation de méthode et de test de sous-typage dans un schéma particulier. La référence est la coloration de méthodes et d'attributs (CM-CA) dans le schéma séparé (S). Les deux premières colonnes représentent le surcoût vis-à-vis de la référence, respectivement avec la coloration d'attributs (CA) ou la simulation des accesseurs (SA). La troisième colonne donne le surcoût de SA par rapport à CA.

TAB. 4 – *Temps de compilation suivant les techniques, schémas et processeurs.*

CA). La signification de ces résultats dépend étroitement de la signification de ce temps de référence, qui pourrait être grossièrement surévalué, réduisant ainsi artificiellement les différences observées.

Il faut donc évaluer le niveau général d'efficacité de PRM$_\mathcal{C}$, ce qui réclame une référence externe. Nous avons pris SMART EIFFEL : les deux langages ont des fonctionnalités équivalentes et SMART EIFFEL est considéré comme très efficace (BTD/G). Sur le Pentium Duo, le temps d'une compilation de SMART EIFFEL vers C sur la machine de test est d'environ 1 minute, soit le même ordre de grandeur que PRM$_\mathcal{C}$ — une comparaison plus fine ne serait pas significative. Pour les deux compilateurs, qui compilent vers C, il faut rajouter le temps de `gcc`, y compris l'édition de liens : 76s pour PRM$_\mathcal{C}$ et 140s pour SMART EIFFEL. Comme les deux compilateurs ne font pas appel au même niveau d'optimisation, ces résultats ne sont pas non plus strictement comparables et la principale conclusion à en tirer est que, au total, les ordres de grandeur sont similaires. On peut donc affirmer que PRM$_\mathcal{C}$ est suffisamment efficace pour que les résultats présentés ici soient considérés comme significatifs : dans le rapport différence sur référence que nous analysons ci-dessous, le dénominateur n'est pas exagérément surévalué. Il l'est cependant un peu, par exemple par l'utilisation de *garbage collector* conservatif de Boehm, qui n'est pas aussi efficace que le serait un *garbage collector* dédié au modèle objet simple de PRM. La gestion mémoire n'utilisant pas de mécanismes objet, son surcoût ne pèse que sur le dénominateur.

## 3.3 Discussion

On considérera d'abord les résultats du Pentium Duo. La première conclusion à tirer de ces résultats concerne les schémas de compilation. Malgré la limitation des optimisations globales effectivement implémentées dans ces tests, le schéma G produit un code nettement plus efficace — ce n'est pas une surprise. En revanche, le schéma O ne semble être qu'une légère amélioration du schéma S. Il s'agit cependant d'un résul-

tat très positif : même avec une faible optimisation, les sauts rajoutés par les *thunks* sont compensés par les gains sur les appels monomorphes. C'est une confirmation de l'analyse abstraite des processeurs modernes dont l'architecture de *pipe-line* est effectivement censée annuler le coût des sauts statiques, modulo les défauts de cache bien entendu (Driesen, 2001). Dans les deux cas, la différence devrait se creuser avec des optimisations plus importantes comme les $\text{BTD}_i$, avec $i \geq 1$, et une analyse de types plus sophistiquée que la simple CHA (Dean et al., 1995) utilisée pour ces tests.

Du point de vue des techniques d'implémentation, les conclusions sont multiples. Le surcoût de la simulation des accesseurs (SA) est réel mais il est suffisamment faible avec la coloration de méthodes (CM) pour que ce ne soit pas un handicap si les trous de la coloration des attributs en sont un. Une analyse a priori de la simulation des accesseurs conclut à un surcoût d'une dizaine de cycles par accès (Ducournau, 2002), soit un total de 2,5 G-cycles d'après les chiffres de la table 3. La différence constatée entre CM-CA et CM-SA est d'un ordre de grandeur légèrement supérieur mais cela s'explique certainement par l'accroissement des défauts de cache et la réduction du parallélisme.

Les résultats des tests sur le hachage parfait sont très intéressants par leur caractère marqué. PH-`and` apparaît très efficace, dépassant presque nos espérances, quand il est couplé avec la coloration d'attributs. Il faut donc vraiment l'envisager pour implémenter les interfaces de Java, d'autant qu'il serait utilisé beaucoup moins intensément dans ce cadre. En revanche, la simulation des accesseurs le dégrade de façon non additive : le surcoût PH-SA/CM-CA est largement supérieur à la somme des surcoûts PH-CA/CM-CA et CM-SA/CM-CA. De son côté, PH-`mod` entraîne un surcoût qui le met vraisemblablement hors jeu sur ce type de processeur et sa combinaison avec la simulation des accesseurs est absolument inefficace. La non additivité marquée qui apparaît avec la simulation des accesseurs vient sans doute du fait que, avec la coloration d'attribut, l'accès se fait par une unique instruction qui est aisément parallélisable. La simulation des accesseurs implique une séquence plus longue qui réduit le parallélisme. PH-`mod` aggrave la situation car le passage à l'unité de calcul flottant présente en plus l'inconvénient de vider le *pipeline*.

Ces conclusions s'appliquent peu ou prou aux trois processeurs considérés ici. Si l'on compare les processeurs, qui sont, dans la table 4, classés de gauche à droite dans l'ordre chronologique, on constate que l'avantage du schéma global s'accroit avec le temps, alors que la simulation des accesseurs et PH-`mod` au contraire se dégradent. L'absence de points de comparaison "toutes choses égales par ailleurs" ne permet pas de juger de l'effet de l'augmentation parallèle de la taille du cache.

# 4 Travaux connexes, conclusion et perspectives

Dans nos travaux précédents, nous avons réalisé des évaluations abstraites (Ducournau, 2002, 2006, 2008) ou des évaluations concrètes reposant sur des programmes artificiels (Privat et Ducournau, 2005a). Cet article présente les premiers résultats d'expérimentation permettant de comparer des techniques d'implémentation et des schémas de compilation, de façon à la fois systématique et aussi équitable que possible. Le protocole d'expérimentation, basé sur le *bootstrap* du compilateur n'est pas nouveau — il

avait été utilisé, entre autres, pour Smart Eiffel. Cependant, Smart Eiffel imposait pour l'essentiel un schéma de compilation (G) et une technique d'implémentation (BTD) et ne les comparait qu'avec un compilateur Eiffel existant. Des travaux analogues ont aussi été menés autour des langages Self et Cecil (Dean et al., 1996), mais ils concernent à la fois la compilation globale (G) et le typage dynamique, dans un cadre adaptatif. En typage statique, le compilateur Polyglot (Nystrom et al., 2006) possède une architecture modulaire analogue à celle de Prm$_\mathcal{C}$ mais nous ne connaissons pas d'expérimentations comparatives réalisées avec ce compilateur dédié a Java. Dans le cadre de Java, toute une littérature s'intéresse à l'invocation de message et au test de sous-typage lorsqu'ils s'appliquent à des interfaces (Alpern et al., 2001). Cependant, le schéma de compilation de Java (D) autorise peu d'implémentations en temps constant — les sous-objets (SO) sont vraisemblablement incompatibles avec les machines virtuelles et le hachage parfait (PH) n'a pas encore été expérimenté. Les autres expérimentations que nous connaissons comparent des techniques en temps non-constant, à base de cache et de recherche plus ou moins naïve, à des techniques, soit non incrémentales — la coloration pour Palacz et Vitek (2003) qui pose de gros problèmes de recalcul au chargement — soit guère plus sophistiquées.

Les expérimentations présentées ici sont donc, à notre connaissance, les premières à comparer différentes techniques d'implémentation ou schémas de compilation *toutes choses égales par ailleurs*. La plate-forme de compilation de Prm produit un code globalement efficace — si l'on compare les temps de compilation avec ceux de Smart Eiffel— même si l'optimum est loin d'être atteint. Les différences que l'on observe devraient donc rester significatives dans des versions plus évoluées. On peut tirer deux conclusions assez robustes de ces expérimentations : (i) même avec une optimisation globale limitée, le schéma global (G) reste nettement meilleur que le schéma séparé (S) ; (ii) le schéma optimisé (O) est prometteur : le surcoût des *thunks* est bien compensé par les optimisations. S'il ne sera vraisemblablement jamais aussi bon que le global, des optimisations plus poussées devraient en faire un bon intermédiaire entre S et G. Cependant, des expérimentations complémentaires sont nécessaires pour valider complètement le schéma O : pour des raisons techniques (l'inefficacité de l'analyseur syntaxique de Prm$_\mathcal{C}$), nous n'avons utilisé ici qu'une analyse de types très primitive, CHA, qui ne nécessite pas d'employer des *modèles internes*. Une analyse plus sophistiquée pourrait rendre la compilation trop lente : notez que nous n'avons pas mesuré ici le temps de compilation des différents compilateurs $C_i$, mais uniquement celui des différentes versions $P_i$ du même compilateur. On peut enfin ajouter à ces conclusions techniques particulières une conclusion méthodologique générale : ces expérimentations confirment pour l'essentiel les analyses abstraites qui ont été menées depuis une dizaine d'années autour d'un modèle de processeur simplifié (Driesen, 2001), même si la sur-additivité des surcoûts excède nos prévisions.

Du côté des techniques d'implémentation, deux conclusions se dégagent : le surcoût de la simulation des accesseurs (SA) est faible quand elle est basée sur la coloration de méthodes (CM). En revanche, sa combinaison avec des techniques plus coûteuses augmente le surcoût de façon non additive. Par ailleurs, cette première implémentation du hachage parfait confirme les analyses abstraites antérieures en séparant nettement PH-`and` et PH-`mod`. Cela confirme l'intérêt de PH-`and` pour les interfaces de Java,

d'autant que des résultats récents démontrent que son coût spatial n'est pas si élevé que cela (Ducournau et Morandat, 2009).

Ces tests présentent une limitation évidente. Un seul programme a été mesuré, dans des conditions de compilation différentes. Cette limitation est d'abord inhérente à l'expérimentation elle-même — bien que ces techniques de compilation soient applicables à tout langage (modulo les spécificités discutées par ailleurs à propos de C++ et Eiffel), le langage Prm a été conçu d'abord pour cette expérimentation. C'est ce qui la rend possible mais explique le fait que le compilateur Prm soit le seul programme Prm significatif. Cela dit, le compilateur Prm est un programme objet représentatif, par son nombre de classes et le nombre d'appels de mécanisme objet. On retrouve aussi un taux d'appels monomorphes comparable à ceux qui sont cités dans la littérature. En revanche, d'autres programmes pourraient différer sur le taux d'accès aux attributs par rapport aux appels de méthodes, ce qui pourrait changer les conclusions vis-à-vis de la simulation des accesseurs.

Les perspectives de ce travail sont de deux ordres. Les comparaisons systématiques ne sont pas encore complètes — il nous reste par exemple à intégrer l'implémentation de C++ (SO), les BTD de profondeur quelconque et à analyser le temps de compilation (par $C_i$), les défauts de cache et l'espace mémoire consommé (par $P_i$). Des expérimentations sur d'autres familles ou architectures de processeurs sont aussi indispensables. Nos expériences sur des Pentiums de diverses générations donnent des résultats assez similaires même si les différences peuvent varier de façon marquée. Il est très possible que des architectures vraiment différentes, RISC par exemple, changent les conclusions. Par exemple, PH-`mod` pourrait très bien revenir dans la course sur un processeur doté d'une division entière native. Le schéma de compilation original de Prm doit être encore amélioré. Il reste des difficultés, par exemple l'élimination du code mort dans un module vivant. Le schéma hybride de compilation séparée des bibliothèques et globale du programme (H), qui représente sans doute un compromis pratique pour le programmeur, reste aussi à tester. Certaines techniques peuvent aussi être améliorées : ainsi la simulation des accesseurs nécessiterait un traitement particulier pour les vrais accesseurs, afin qu'ils ne soient pas doublement pénalisés. Enfin, l'adoption d'un *garbage collector* plus adapté à Prm pourrait augmenter l'efficacité globale et la part des mécanismes objet dans la mesure totale, donc les différences relatives.

Depuis le début de cette recherche, notre démarche vise à développer des optimisations globales à l'édition de liens. Dans une perspective à plus long terme, les techniques qui ont été mises au point doivent aussi pouvoir s'appliquer aux compilateurs adaptatifs des machines virtuelles (Arnold et al., 2005). Le *thunk* généré au chargement d'une classe pourrait être recalculé lors du chargement d'une nouvelle classe qui infirme les hypothèses antérieures. La plate-forme Prm peut fournir des premières indications : on pourrait par exemple utiliser des *thunks* avec hachage parfait pour tous les appels polymorphes et des appels statiques pour les appels monomorphes. Cela permettrait une première validation de l'utilisation des *thunks* dans un cadre de compilation adaptative mais seule l'expérimentation dans un cadre de chargement dynamique permettrait de mesurer l'effet des recompilations, en particulier sur la localité des accès mémoire. Dans une moindre mesure, cette limitation de la plate-forme d'expérimentations vaut aussi pour les tests sur le hachage parfait présentés ici.

## Références


Alpern, B., A. Cocchi, S. Fink, et D. Grove (2001). Efficient implementation of Java interfaces : Invokeinterface considered harmless. In *Proc. OOPSLA'01*, SIGPLAN Notices, 36(10), pp. 108–124. ACM Press.

Arnold, M., S. Fink, D. Grove, M. Hind, et P. Sweeney (2005). A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE 93*(2), 449–466.

Boucher, D. (2000). GOld : a link-time optimizer for Scheme. In M. Felleisen (Ed.), *Proc. Workshop on Scheme and Functional Programming. Rice Technical Report 00-368*, pp. 1–12.

Cohen, N. (1991). Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst. 13*(4), 626–629.

Collin, S., D. Colnet, et O. Zendra (1997). Type inference for late binding. the SmallEiffel compiler. In *Proc. Joint Modular Languages Conf.*, LNCS 1204, pp. 67–81. Springer.

Czech, Z. J., G. Havas, et B. S. Majewski (1997). Perfect hashing. *Theor. Comput. Sci. 182*(1-2), 1–143.

Dean, J., G. Defouw, D. Grove, V. Litvinov, et C. Chambers (1996). Vortex : An optimizing compiler for object-oriented languages. In *Proc. OOPSLA'96*, SIGPLAN Notices, 31(10), pp. 83–100. ACM Press.

Dean, J., D. Grove, et C. Chambers (1995). Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff (Ed.), *Proc. ECOOP'95*, LNCS 952, pp. 77–101. Springer.

Dixon, R., T. McKee, P. Schweitzer, et M. Vaughan (1989). A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*, pp. 211–214. ACM Press.

Driesen, K. (2001). *Efficient Polymorphic Calls*. Kluwer Academic Publisher.

Ducournau, R. (2002). Implementing statically typed object-oriented programming languages. Technical Report 02-174, LIRMM, Université Montpellier 2.

Ducournau, R. (2006). Coloring, a versatile technique for implementing object-oriented languages. Technical Report 06-001, LIRMM, Université Montpellier 2.

Ducournau, R. (2008). Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst. 30*(5), 52.

Ducournau, R. et F. Morandat (2009). More results on perfect hashing for implementing object-oriented languages. Rapport de recherche 09-001, LIRMM, Université Montpellier 2, Montpellier.

Ellis, M. et B. Stroustrup (1990). *The annotated C++ reference manual*. Reading, MA, US : Addison-Wesley.

Grove, D. et C. Chambers (2001). A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23*(6), 685–746.

Meyer, B. (1997). *Eiffel - The language*. Prentice-Hall.

Myers, A. (1995). Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pp. 124–139. ACM Press.

Nystrom, N., X. Qi, et A. C. Myers (2006). $\mathcal{JE}$ : Nested intersection for scalable software composition. In P. L. Tarr et W. R. Cook (Eds.), *Proc. OOPSLA'06*, SIGPLAN Notices, 41(10), pp. 21–35. ACM Press.

OOPSLA (1997). *Proc. 12th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*, SIGPLAN Notices, 32(10). ACM Press.

Palacz, K. et J. Vitek (2003). Java subtype tests in real-time. In L. Cardelli (Ed.), *Proc. ECOOP'2003*, LNCS 2743, pp. 378–404. Springer.

Privat, J. et R. Ducournau (2004). Intégration d'optimisations globales en compilation séparée des langages à objets. In J. Euzenat et B. Carré (Eds.), *Actes LMO'2004* in *L'Objet vol. 10*, pp. 61–74. Lavoisier.

Privat, J. et R. Ducournau (2005a). Link-time static analysis for efficient separate compilation of object-oriented languages. In *ACM Workshop on Prog. Anal. Soft. Tools Engin. (PASTE'05)*, pp. 20–27.

Privat, J. et R. Ducournau (2005b). Raffinement de classes dans les langages à objets statiquement typés. In M. Huchard, S. Ducasse, et O. Nierstrasz (Eds.), *Actes LMO'2005* in *L'Objet vol. 11*, pp. 17–32. Lavoisier.

Pugh, W. et G. Weddell (1990). Two-directional record layout for multiple inheritance. In *Proc. PLDI'90*, ACM SIGPLAN Notices, 25(6), pp. 85–91.

Traon, Y. L. et B. Baudry (2006). Test d'intégration d'un système à objets — planification de l'ordre d'intégration. In R. Rousseau, C. Urtado, et S. Vauttier (Eds.), *Actes LMO'06*, pp. 217–230. Hermès Lavoisier.

Vitek, J., R. Horspool, et A. Krall (1997). Efficient type inclusion tests. In (OOPSLA, 1997), pp. 142–157.

Zendra, O., D. Colnet, et S. Collin (1997). Efficient dynamic dispatch without virtual function tables : The SmallEiffel compiler. In (OOPSLA, 1997), pp. 125–141.

## Summary

Object-oriented programming involves a trade-off between three aspects, namely multiple inheritance, efficiency and open world assumption—especially with dynamic loading. This paper presents experiment results comparing the efficiency of different implementation techniques (coloring, BTD, perfect hashing, ...) in the context of different compilation schemes (from separate compilation with dynamic loading to purely global compilation). These tests are performed with the Prm compiler and applied to it. They mostly confirm previous theoretical results, while showing that overheads are not additive. Global optimization schemes markedly improve upon coloring, that serves as a reference. Accessor simulation and perfect hashing, each considered in isolation, yield limited overhead but their combination doubles the total overhead.