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

Floréal Morandat[1], Roland Ducournau[1], Jean Privat[2]

[1] LIRMM – CNRS et Université Montpellier II
161 rue Ada Montpellier – 34392 Cedex 5 France
{morandat,ducour}@lirmm.fr,
[2] Université du Québec à Montréal
privat.jean@uqam.ca

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. Les schémas d'optimisation globale démontrent un gain significatif par rapport à la coloration qui fait fonction de référence, tandis que le chargement dynamique rend réel le surcoût de l'héritage multiple. Enfin, ces tests confirment l'intérêt du hachage parfait pour les interfaces de Java.

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.*, 2001a; 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 méthode 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, 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.

Cet article étend les premiers résultats de [Morandat *et al.*, 2009] avec les nouvelles expérimentations ainsi qu'une partie des résultats présentés dans [Ducournau *et al.*, 2009]. Nous commencerons par décrire 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. Nous présentons ensuite une série de tests réalisés sur PRM$_\mathcal{C}$, afin de mesurer l'impact réel des différents schémas et techniques sur de vrais programmes — en l'occurrence 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,] 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.

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.

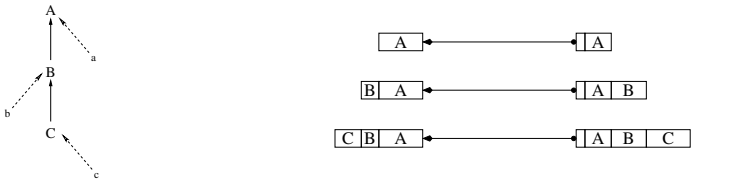
Sous-objets (SO) Dans le cas de l'héritage multiple, ces invariants sont trop forts car deux classes incomparables, peuvent avoir une sous-classe commune (par exemple B et C dans la figure 1-b). La technique de base ne peut donc plus être utilisée.

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

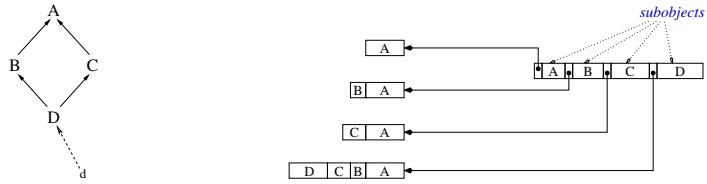
pointeur. Cet ajustement est dépendant du type dynamique de l'instance pointée, donc la table des méthodes doit contenir aussi les décalages à utiliser. Dans le pire des cas, la taille totale des tables est ainsi *cubique* dans le nombre de classes. Cette implémentation reste compatible avec le chargement 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 l'héritage multiple et le chargement dynamique, mais c'est au prix d'une réutilisabilité limitée (CWA).

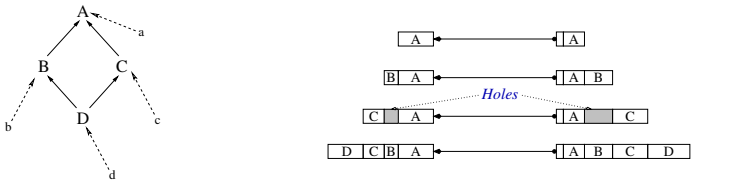
Coloration (MC/AC) La coloration a été initialement proposée par Dixon *et al.* [1989], Pugh et Weddell [1990] et Vitek *et al.* [1997] pour chacun des trois mécanismes de base. Ducournau [2006] montre l'identité des trois techniques et fait la synthèse de l'approche. L'idée principale est de maintenir 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).



-a Implémentation en héritage simple



-b Implémentation par sous-objets



-c Implémentation par coloration

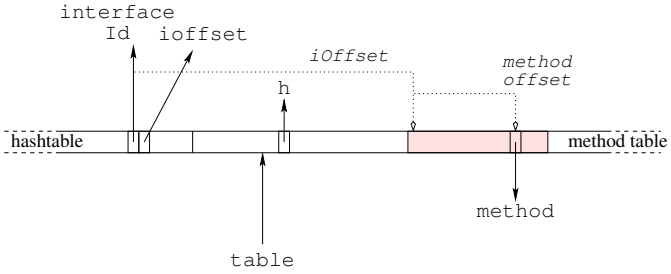
A chaque exemple de hiérarchie de classes $(A, B, C, \ldots)$ est associée l'implémentation correspondante d'instances $(a, b, c, \ldots)$. Les objets sont à droite, les tables de méthodes à gauche, inversées pour éviter les croisements. Dans les tables, les étiquettes de classe désignent le groupe de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

Fig. 1. Différentes techniques d'implémentation

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, MC désigne l'application de la coloration aux méthodes et au test de sous-typage et AC l'application à la représentation des instances. D'autres techniques de compression de tables existent, par exemple *row displacement* [Driesen, 2001], mais elles ne s'appliquent pas à la représentation des objets.

Simulation des accesseurs (AS) La simulation des accesseurs, proposée par Myers [1995] et Ducournau [], 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].

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 BTD_i un BTD de profondeur i. Les BTD_0 correspondent aux appels statiques.



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

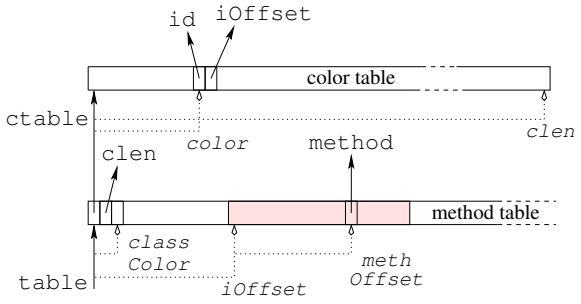
Hachage parfait (PH) Le hachage parfait est une optimisation en temps constant des tables de hachage pour des ensembles immutables [Czech *et al.*, 1997]. Ducournau [2008] propose de l'utiliser pour le test de sous-typage et l'envoi de message, dans le cas particulier des interfaces en JAVA (Figure 2). Cette technique nécessite une indirection supplémentaire avant l'appel de la méthode. 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.

Le hachage parfait peut être utilisé pour l'accès aux attributs grâce à la simulation des accesseurs, pour cela la table de hachage doit contenir des entrés triples[3].

Coloration incrémentale (IC) Une version incrémentale de la coloration a été proposée par Palacz et Vitek [2003] pour implémenter le test de sous-typage en JAVA. Pour les besoins de la comparaison, une application à l'invocation de méthode basée sur les mêmes principes que l'appel de méthode du hachage parfait a été proposée dans [Ducournau, 2008].

La coloration nécessitant l'hypothèse du monde clos (CWA), la coloration incrémentale peut entraîner certains recalculs durant le chargement. De ce fait, la structure de donnée utilisée ajoute une indirection dans une zone mémoire différente, ce qui a pour conséquence d'augmenter le risque de défaut de cache (Figure 3). Le lecteur intéressé est renvoyé à [Ducournau, 2008] pour les détails de l'implémentation.



L'implémentation ressemble à celle du hachage parfait sauf que la position d'une interface dépend d'un `load` au lieu d'être le résultat d'une fonction de hachage. La position d'une couleur peut être stockés dans la table des méthodes de la classe cible ou dans une table séparée. Dans tous les cas, le `load` nécessite un accès à une région mémoire différente et peut conduire à trois défauts de cache supplémentaire
De plus, comme la table des couleurs doit pouvoir être recalculée, elle doit nécessairement être disjointe de la table des méthodes et nécessite donc une indirection supplémentaire ainsi qu'un test de borne sur la valeur `clen` — et donc un `load` supplémentaire.

Fig. 3. Coloration incrémentale pour les interfaces de JAVA

Caches et recherche (CA) Une technique d'implémentation souvent appliquée dans les langages à typage dynamique ou aux interfaces de JAVA consiste à coupler une technique d'implémentation (qui est supposée, ici, être plutôt naïve et inefficace) à un cache qui mémoïse le résultat de la dernière recherche [Alpern *et al.*, 2001a; Alpern *et al.*, 2001b; Click et Rose, 2002].

Par exemple, pour les interfaces de JAVA, en utilisant le hachage parfait (PH) ou la coloration incrémentale (IC) comme technique sous-jacente, la table des méthodes pourrait *cacher* l'identifiant de l'interface ainsi que l'offset du groupe de méthode du dernier accès réussi. La séquence de code pour l'invocation de méthode et le test de sous-typage est relativement plus longue que l'implémentation sous-jacente. De plus, le pire des cas est plutôt inefficace car il rajoute la gestion du cache au temps nécessaire pour l'appel sous-jacent. Quant au meilleur des cas, il est légèrement plus efficace que PH-`and` pour l'invocation de méthode et identique au test de Cohen [1991] pour le test de sous-typage. Nous pourrions attendre du cache qu'il dégrade PH-`and` mais qu'il améliore PH-`mod`.

Ce cache peut être utilisé avec toutes les techniques d'implémentation basées sur des tables et pour chacun des trois mécanismes, au prix de *cacher* les trois données qui sont : l'identifiant de

[3] Pour respecter l'alignement, il vaut mieux faire des entrés quadruples.

la classe, l'offset du groupe de méthodes ainsi que celui du groupe d'attributs introduits par cette classe. De plus, le cache peut être commun aux trois mécanismes ou dédié à chacun d'eux. A l'instar de la coloration incrémentale et à l'inverse de toutes les autres techniques, le cache nécessite que les tables de méthodes soient accessibles en écriture — donc allouées dans un segment mémoire de données.

L'amélioration proposée par le cache est une question de statistiques. Celles présentées dans [Palacz et Vitek, 2003] montrent qu'en fonction des benchmarks les défauts de cache varient de 0.1% à plus de 50%. Le taux de réussite du cache peut être amélioré en utilisant plusieurs caches, ce qui augmente encore la taille du code par rapport à l'implémentation sous-jacente. Les classes (ou interfaces) sont statiquement partitionnées en n ensembles — par exemple en hachant leur nom — et la structure du cache est répliquée n fois dans la table des méthodes. Quand n augmente, le taux de défaut de cache doit asymptotiquement tendre vers 0 — néanmoins, le meilleur et le pire des cas restent inchangés. De plus, avec cette approche, les tables de la structure sous-jacente peuvent se restreindre à contenir les identifiants de classe (ou d'interface) pour lesquelles il demeure une collision dans leur propre cache.

Dans nos tests, nous considérerons aussi le hachage parfait lorsqu'il est couplé à un cache.

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 (superclasses 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) dans un cadre de chargement dynamique (cf. 2.2) 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.

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'héritage 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.

Compilation globale (G) Ce schéma de compilation est utilisé par Eiffel[4] et constitue le cadre de toute la littérature sur l'optimisation des programmes objet autour des langages Self,

[4] 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.

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.

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.

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

Le tableau 1 résume la compatibilité des schémas de compilation avec les techniques d'implémentation ainsi que l'ensemble des combinaisons testés.

Technique d'implémentation		Schéma de compilation				
		D dynamique	S séparé	O optimisé	H hybride	G global
sous-objets	SO	◇	◇	∗	∗	∗
hachage parfait	PH	○	●	∗	∗	∗
coloration incrémentale	IC	○	●	∗	∗	∗
caching	CA	○	●	∗	∗	∗
method coloring	MC	⋆	●	●	◇	●
binary tree dispatch	BTD	⋆	⋆	●	◇	●
attribute coloring	AC	⋆	●	●	◇	●
accessor simulation	AS	○	●	●	◇	●

● : Testé, ○ : Extrapolé, ◇ : Compatible mais non testé, ∗ : inintéressant, ⋆ : Incompatible

Tab. 1. Compatibilité avec les schémas et efficacité des techniques d'implémentation

Le tableau 2 décrit l'efficacité a priori des techniques en se basant sur les études antérieures [Ducournau,]. 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. Un des objectifs des expérimentations qui suivent est de vérifier empiriquement les évaluation théoriques.

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 la restriction suivante :

– les spécifications de Prm, en particulier le raffinement de classes, le rendent absolument incompatible avec le chargement dynamique ; le hachage parfait et la coloration incrémentale ont donc été implémentés en compilation séparée avec édition de liens globale (S) mais le code généré est exactement celui qui aurait été généré avec chargement dynamique (D). Seuls les défauts de cache et les recalculs liés à la coloration incrémentale sont sous-estimés.

3.1 Expérimentations

Le dispositif d'expérimentation (Figure 4) 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 $\text{Prm}_{\mathcal{C}}$. 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

	Espace			Temps		
	Code	Statique	Dyn.	Exécution	Compilation	Chargement/Édt. de liens
SO	−	− −	− −	−	++	++
IC	−	+	+++	−	++	− −
PH-`and`	−	−	+++	−	++	+
PH-`mod`	−	+	+++	− −	++	+
PH-`and` +AC	− −	− −	+++	− −	++	+
PH-`mod` +AC	− −	−	+++	−	++	+
MC	++	++	+++	++	+	−
$\text{BTD}_{i<2}$	+++	+++	+++	+++	++	− −
$\text{BTD}_{i>4}$	− − −	+++	+++	− − −	−	− −
AC	++	++	+	++	+	−
AS	+	+	+++	−	+	+

+++ : optimal, ++ :très bon, + : bon, − : mauvais, − − : très mauvais, − − − : déraisonnable

Tab. 2. Efficacité attendue

des processeurs multi-cœur). Les mesures du temps et les statistiques 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 le meilleur temps pris sur au moins dix exécutions. Un compilateur P_i est généré pour chaque exécution pour tenir compte 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.

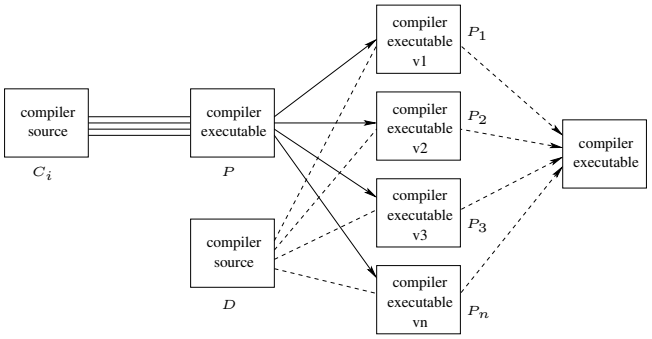


Fig. 4. Protocole d'expérimentation

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.

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_{ref} est le temps de la version de référence, c'est-à-dire le schéma S avec coloration (MC-AC). 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.

nombres de		statique	dynamique
classes	introductions	532	—
	instantiations	6651	35 M
	tests de sous-typage	194	87 M
méthodes	introductions	2599	—
	définitions	4446	—
	appels	15873	1707 M
BTD	0	124875	1134 M
	1	848	61 M
	2	600	180 M
	3	704	26 M
	4..7	1044	306 M
	8	32	228 K
attributs	introductions	614	—
	accès	4438	2560 M

La colonne "statique" représente le nombre d'éléments du programme (classes, méthodes, attributs) ainsi que le nombre de sites d'appel pour chaque mécanisme. La colonne "dynamique" rapporte le nombre d'invocations de ces sites durant l'exécution (en milliers ou millions). Les appels de méthodes sont comptés séparément en fonction de leur degré de polymorphisme.

Tab. 3. Statistiques sur le programme de test P [Ducournau *et al.*, 2009]

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 I-5, 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. 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 *ramasse-miettes* conservatif de Boehm, qui n'est pas aussi efficace que le serait un *ramasse-miettes* dédié au modèle objet original de PRM. La gestion mémoire n'utilisant pas de mécanismes objet, son surcoût ne pèse que sur le dénominateur.

3.3 Discussion

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. Le fort taux de sites d'appels monomorphes explique ce résultat, puisque la seule différence entre MC-S et MC-BTD$_0$-G est la *mise en ligne* des appels monomorphes.

En revanche, le schéma O ne semble être qu'une légère amélioration du schéma S. Il s'agit cependant d'un résultat 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 en *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 par exemple 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 (AS) est réel mais il est suffisamment faible avec la coloration de

processeur		S-1	UltraSPARC III		I-2	Xeon Prestonia		I-4	Xeon Irwindale		I-5	Core T2400	
fréquence			1.2 GHz			1.8 GHz			2.8 GHz			2.8 GHz	
L2 cache			8192 K			512 K			2048 K			2048 K	
année		123.2s	2001		87.4s	2001		43.3s	2006		34.8s	2006	
technique	scheme	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
MC-BTD$_\infty$	RTA G	-22.6	-11.5	14.4	-10.9	-2.2	9.7	-13.3	-1.0	14.1	-8.9	2.9	13.0
MC-BTD$_2$	RTA G	-22.2	-10.9	14.6	-11.7	-3.8	10.6	-13.7	-1.3	14.4	-10.2	-0.8	10.5
MC-BTD$_\infty$	CHA O	***	***	***	-2.9	1.4	4.5	-3.0	4.9	8.2	2.7	19.4	16.2
MC-BTD$_2$	CHA O	***	***	***	-5.4	-2.8	2.8	-5.9	2.3	8.7	-2.7	14.2	17.3
MC	S	0	9.8	9.8	0	5.7	5.7	0	7.9	7.9	0	11.1	11.1
IC	D	13.7	34.1	17.9	5.3	14.5	8.7	4.3	16.6	11.8	7.7	28.5	19.2
PH-`and`	D	13.4	35.4	19.5	2.5	14.5	11.6	4.2	19.0	14.2	6.2	31.4	23.8
PH-`mod`	D	81.1	226.0	80.0	28.6	104.3	58.8	55.2	172.0	75.2	24.4	106.3	65.8
PH-`mod+CA`$_4$	D	33.1	121.0	66.1	21.1	81.2	49.6	28.1	98.2	54.7	21.4	82.7	50.5
processeur		A-6	Athlon 64		A-7	Opteron Venus		I-8	Core2 T7200		I-9	Core2 E8500	
fréquence			2.2 GHz			2.4 GHz			2.0 GHz			3.16 GHz	
L2 cache			1024 K			1024 K			4096 K			6144 K	
année		34.0s	2003		32.8s	2005		30.4s	2006		18.5s	2008	
technique	scheme	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
MC-BTD$_\infty$	RTA G	-12.5	2.9	17.6	-13.7	9.9	27.3	-9.4	7.5	18.6	-10.3	0.6	12.1
MC-BTD$_2$	RTA G	-12.5	1.1	15.5	-13.1	10.7	27.3	-9.7	7.0	18.5	-9.7	0.6	11.5
MC-BTD$_\infty$	CHA O	5.8	27.8	20.8	4.0	23.3	18.6	1.1	12.1	10.8	1.8	15.0	12.9
MC-BTD$_2$	CHA O	3.4	27.3	23.1	2.0	22.8	20.4	-1.8	10.1	12.2	-2.5	13.0	16.0
MC	S	0	15.8	15.8	0	15.2	15.2	0	11.3	11.3	0	10.3	10.3
IC	D	14.9	40.1	21.9	12.6	38.9	23.3	7.7	29.7	20.5	8.0	29.7	20.1
PH-`and`	D	15.2	52.4	32.2	16.8	57.3	34.7	5.1	30.0	23.7	6.2	30.1	22.6
PH-`mod`	D	80.1	235.4	86.2	73.4	221.5	85.4	19.5	108.7	74.7	17.6	85.7	57.9
PH-`mod+CA`$_4$	D	40.4	141.2	71.8	38.3	129.3	65.8	19.6	81.3	51.6	20.8	74.6	44.5

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 (MC-AC) dans le schéma séparé (S). Les deux premières colonnes représentent le surcoût vis-à-vis de la référence, respectivement avec la coloration d'attributs (AC) ou la simulation des accesseurs (AS). La troisième colonne donne le surcoût de AS par rapport à AC.

Tab. 4. Temps de compilation suivant les techniques, schémas et processeurs [Ducournau *et al.*, 2009]

méthodes (MC) pour que ce ne soit pas un handicap si les trous de la coloration des attributs en sont un.

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.

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 simulation des accesseurs implique une séquence plus longue ce 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 processeurs considérés ici. Si l'on compare les processeurs, qui sont, dans les tables 4, classés de gauche à droite dans l'ordre chronologique, il est difficile d'en tirer des conclusions régulières. 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.

Esquisse de prescription. Certes les conclusions de ces expériences ne nous permettent pas de conclure définitivement pour tous les processeurs et pour tous les programmes, le choix d'une implémentation particulière restant dépendant des besoins fonctionnels du langage, par exemple le chargement dynamique. Cet article n'a pas la prétention de livrer une prescription sur comment implémenter les langages objets. Cependant, en partant du compromis entre modularité, efficacité et expressivité, ces premiers résultats peuvent être formulés de manière plus prescriptive.

- Si le point essentiel est l'efficacité, par exemple pour les petits systèmes embarqués, la compilation globale (G) avec un mélange de BTD — éventuellement restreint à BTD_1 si le processeur n'est pas équipé de prédiction de branchement — et de coloration est définitivement la solution. En effet, dans ce cadre le surcoût de l'héritage multiple est insignifiant.
- Si l'essentiel est l'expressivité et que le chargement dynamique n'est pas requis, la coloration en compilation séparée (S) représente certainement le meilleur compromis entre la modularité et l'efficacité — il peut être amélioré avec des optimisations globales lors de l'édition de liens (O). Le schéma hybride (H) qui combine la compilation séparée des librairies et la compilation globale du programme permet un bon compromis entre la flexibilité — recompilations rapides — et l'efficacité du code produit. Dans ce cadre, l'utilisation conjointe des BTD et de la coloration assure le code le plus compact et le plus efficace.
- Si l'essentiel est la modularité, C++ est une solution éprouvée dans le cadre de l'héritage multiple et du chargement dynamique. De plus, par rapport à Java, le surcoût des sous-objets est en *pratique* contre-balancé par l'implémentation hétérogène des génériques, par le fait que les types primitifs ne sont pas intégrés au système de type objet et que les programmeurs utilisent beaucoup moins les fonctionnalités objet. Cependant, le passage à l'échelle de cette implémentation est douteux, dans le pire des cas, la taille des tables est cubique dans le nombre de classes et le nombre d'entrées ajoutées par le compilateur dans les diverses tables peut être élevé. De ce fait, ces conclusions ne s'appliquent sûrement qu'à des programmes de taille moyenne, comme le compilateur Prm. En revanche, le sous-typage multiple — à la Java ou C#— est un compromis intéressant entre l'expressivité et l'efficacité. Actuellement cette efficacité vient principalement du compilateur JIT, cependant PH-`and` devrait être considéré par les concepteurs de machines virtuelles comme une implémentation sous-jacente pour les interfaces — dans le cas où le compilateur JIT n'arrive pas à la court-circuiter.

4 Travaux connexes, conclusion et perspectives

Dans nos travaux précédents, nous avons réalisé des évaluations abstraites [Ducournau, ; Ducournau, 2006; Ducournau, 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 $\text{Prm}_{\mathcal{C}}$ 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.*, 2001a]. 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 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$_C$), 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].

Du côté des techniques d'implémentation, deux conclusions se dégagent : le surcoût de la simulation des accesseurs (AS) est faible quand elle est basée sur la coloration de méthodes (MC). En revanche, sa combinaison avec des techniques plus coûteuses augmente le surcoût. 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) 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 *ramasse-miettes* 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 *et al.*, 2001a] B. Alpern, A. Cocchi, S. Fink, et D. Grove. Efficient implementation of Java interfaces : Invokeinterface considered harmless. In *Proc. OOPSLA'01*, SIGPLAN Notices, 36(10), pages 108–124. ACM Press, 2001.

[Alpern *et al.*, 2001b] B. Alpern, A. Cocchi, et D. Grove. Dynamic type checking in Jalapeño. In *Proc. USENIX JVM'01*, 2001.

[Arnold *et al.*, 2005] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, et Peter F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proc. of the IEEE, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation*, 2005.

[Boucher, 2000] Dominique Boucher. GOld : a link-time optimizer for Scheme. In *Proc. Workshop on Scheme and Functional Programming. Rice Technical Report 00-368*, éditeur M. Felleisen, pages 1–12, 2000.

[Click et Rose, 2002] C. Click et J. Rose. Fast subtype checking in the Hotspot JVM. In *Proc. ACM-ISCOPE conference on Java Grande (JGI'02)*, pages 96–107, 2002.

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

[Collin *et al.*, 1997] S. Collin, D. Colnet, et O. Zendra. Type inference for late binding. the SmallEiffel compiler. In *Proc. Joint Modular Languages Conference*, LNCS 1204, pages 67–81. Springer, 1997.

[Czech *et al.*, 1997] Zbigniew J. Czech, George Havas, et Bohdan S. Majewski. Perfect hashing. *Theor. Comput. Sci.*, 182(1-2) :1–143, 1997.

[Dean *et al.*, 1995] J. Dean, D. Grove, et C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. ECOOP'95*, éditeur W. Olthoff, LNCS 952, pages 77–101. Springer, 1995.

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

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

[Driesen, 2001] K. Driesen. *Efficient Polymorphic Calls*. Kluwer Academic Publisher, 2001.

[Ducournau,] R. Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comp. Surv.*, 42. (to appear).

[Ducournau *et al.*, 2009] R. Ducournau, F. Morandat, et J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In *Proc. OOPSLA'09*, éditeur Gary T. Leavens, SIGPLAN Notices, 44(10), pages 41–60. ACM Press, 2009.

[Ducournau et Morandat, 2009] R. Ducournau et F. Morandat. More results on perfect hashing for implementing object-oriented languages. Rapport Technique 09-001, LIRMM, Université Montpellier 2, 2009.

[Ducournau, 2006] R. Ducournau. Coloring, a versatile technique for implementing object-oriented languages. Rapport Technique LIRMM-06001, Université Montpellier 2, 2006.

[Ducournau, 2008] R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6) :1–56, 2008.

[Ellis et Stroustrup, 1990] M.A. Ellis et B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, Reading, MA, US, 1990.

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

[Meyer, 1997] B. Meyer. *Eiffel - The language*. Prentice-Hall, 1997.

[Morandat *et al.*, 2009] F. Morandat, R. Ducournau, et J. Privat. Evaluation de l'efficacité des implémentations de l'héritage multiple en typage statique. In *Actes LMO'2009*, éditeurs B. Carré et O. Zendra, pages 17–32. Cépaduès, 2009.

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

[Nystrom *et al.*, 2006] Nathaniel Nystrom, Xin Qi, et Andrew C. Myers. $\mathcal{JE}$: Nested intersection for scalable software composition. In *Proc. OOPSLA'06*, éditeurs Peri L. Tarr et William R. Cook, SIGPLAN Notices, 41(10), pages 21–35. ACM Press, 2006.

[Palacz et Vitek, 2003] Krzysztof Palacz et Jan Vitek. Java subtype tests in real-time. In *Proc. ECOOP'2003*, éditeur L. Cardelli, LNCS 2743, pages 378–404. Springer, 2003.

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

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

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

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

[Vitek *et al.*, 1997] J. Vitek, R.N. Horspool, et A. Krall. Efficient type inclusion tests. In *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 142–157. ACM Press, 1997.

[Zendra *et al.*, 1997] O. Zendra, D. Colnet, et S. Collin. Efficient dynamic dispatch without virtual function tables : The SmallEiffel compiler. In *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 125–141. ACM Press, 1997.