Exploration et cartographie des passes de LLVM

Rapport de stage de recherche de M1

Sébastien Michelland, sous la supervision de Sébastien Mosser (UQÀM),
cosupervisé par Laure Gonnord et Matthieu Moy (Université Lyon 1)

1 Introduction

1.1 L'équipe ACE et les problèmes de composition

J'ai été accueilli dans le groupe ACE[1] du département d'informatique à l'Université du Québec à Montréal pendant 11 semaines, pour y étudier les optimisations de LLVM [1]. L'équipe travaille dans le domaine du génie logiciel, qui regroupe les techniques et méthodes utiles à l'élaboration de logiciels. Mon sujet en particulier se concentre sur la notion de *composition*.

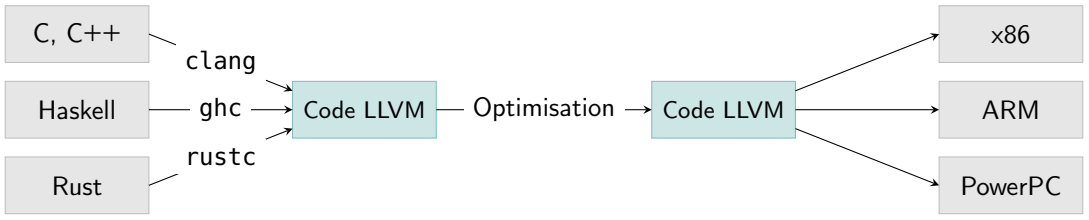
L'idée de la composition provient d'un principe de génie logiciel dit de *séparation des préoccupations* et attribué à Dijkstra, qui consiste à rendre indépendantes les parties d'un logiciel qui accomplissent des objectifs différents. Cela permet de travailler sainement sur chacune d'entre elles sans risquer d'affecter les autres. Par exemple, on peut décider de séparer une application en différents services, ou d'implémenter chaque nouvelle fonctionnalité d'un logiciel dans sa propre branche d'un gestionnaire de version.

Ce principe fondamental laisse toutefois au programmeur la tâche de *composer* les parties maintenant indépendantes pour résoudre son problème initial. Ce travail est moins trivial qu'il n'y paraît et peut se présenter sous différentes formes ; on voudra par exemple définir un protocole de communication entre les services séparés, ou fusionner les branches associées aux nouvelles fonctionnalités développées.

1.2 Passes d'optimisation dans LLVM

LLVM est un ensemble d'outils de compilation originellement développé en 2004 pour étudier des techniques de compilation dynamique, et largement diffusé dans le monde logiciel et la communauté scientifique depuis. C'est un compilateur moderne et riche en fonctionnalités, utilisé pour compiler plusieurs dizaines de langages et le standard *de facto* en recherche.

La force de LLVM repose sur sa modularité. Les outils de compilation sont construits autour d'une *représentation intermédiaire* (que j'appellerai aussi *code LLVM*) indépendante des langages source et cible. Dans un schéma simplifié du processus de compilation, c'est une étape intermédiaire où diverses optimisations ont lieu. Pour compiler, on génère d'abord du code LLVM à partir du langage source, on l'optimise de différentes façons, puis on génère le code cible, typiquement de l'assembleur.



Front-end *Middle-end* *Back-end*

```
int add(int x, int y)
{
    return x + y;
}
```

```
define i32 @add(i32, i32) {
  %3 = add i32 %1, %0
  ret i32 %3
}
```

```
add:
  leal (%rdi,%rsi), %eax
  retq
```

Figure 1 – Schéma du processus de compilation d'un programme avec LLVM.

L'existence de la représentation intermédiaire permet de brancher n'importe quel *front-end* (langage source) avec n'importe quel *back-end* (langage cible) et ainsi créer une grande variété de compilateurs.

1. https://ace-design.github.io/

Le *middle-end* peut de plus être réutilisé dans toutes les combinaisons : c'est le cœur du projet LLVM et la partie qui nous intéresse ici.

En effet, le procédé d'optimisation cache un problème de composition. Pour optimiser un programme, LLVM exécute une série de *passes*, qui l'analysent et le transforment. Ces passes sont indépendantes en termes de code, mais doivent être composées pour optimiser fortement. Environ 80 passes sont utilisées couramment, ce qui laisse plus de 80! façons de les choisir et de les ordonner.

Cela donne lieu à un problème combinatoire appelé *ordonnancement des phases* (*phase ordering*), qui est étudié depuis plus de 30 ans [2]. Parmi les solutions proposées, on note des modèles d'apprentissage machine qui prédisent le bon ordre de passes pour optimiser chaque programme individuellement [3]. Ces modèles, implémentés sur LLVM comme sur GCC, dépassent souvent de 15% ou 20% les performances du compilateur [4, 5]. Cependant toutes les solutions sont encore difficiles à implémenter ou à intégrer, si bien que les compilateurs modernes reposent toujours sur des séquences de passes choisies par expérience du métier.

1.3 Objectifs de ce stage

Le problème d'ordonnancement des phases témoigne de la complexité du système d'optimisation des compilateurs modernes. L'objectif de ce stage a été de comprendre les problématiques de recherche et de développement liées à ce système, et de proposer des outils théoriques et pratiques pour les faciliter.

Concrètement, je me suis intéressé à **expliquer le comportement des passes et leurs interactions**, dans la perspective de construire des outils permettant de mieux appréhender le système d'optimisation : par exemple localiser les passes les plus utilisées, estimer l'impact d'une analyse sur la qualité de l'optimisation, ou comprendre comment une nouvelle passe interagit avec celles qui existent.

Le résultat principal est la construction d'une **carte des passes de LLVM**, un graphe interactif offrant une vue d'ensemble sur les passes et leurs propriétés, et annoté de dépendances, conflits et d'autres interactions. Avec bien entendu les outils permettant d'obtenir ces informations.

Ce stage fait suite au Projet d'Orientation de Master de Julian Bruyat à l'Université Lyon 1, dont le rapport est disonible en ligne[2]. Julian a construit des outils permettant de compiler les programmes de test de LLVM avec différentes options d'optimisation et de tester leurs performances. J'ai réutilisé ce travail pour construire mes méthodes de mesure dans la section 3.

1.4 Plan

La section 2 présente la construction de la carte non annotée à partir d'une analyse du code source des passes. La section 3 montre une méthode d'analyse de performance de programmes pour évaluer l'impact de chaque passe. La section 4 exploite une notion de différence de programmes dans le langage LLVM pour étudier les conflits d'édition entre les passes.

La section 5 montre comment les outils construits peuvent être utilisés pour prouver l'impact d'une passe d'analyse sur la séquence d'optimisation. Enfin, la section 6 présente une autre notion de différence de programmes, non implémentée mais qui permettrait de mieux capturer et comprendre les transformations faites par les passes sur les programmes.

Le rôle et l'utilisation des outils de ce projet sont décrits dans l'annexe A à la fin du document.

2. Notamment une copie sur ma page personnelle :
https://perso.ens-lyon.fr/sebastien.michelland/llvm/RapportJulian/rapport_POM_JB.pdf

2 Construction de la carte avec les métadonnées

Pour comprendre et étudier les passes de LLVM, la première étape est de les lister et classifier selon des critères permettant de comprendre leur comportement ; et il faut le faire automatiquement car le *middle-end* contient environ 280 passes.

Dans cette section, je construis une première **carte des passes** sous la forme d'une base de données requêtable et extensible, munie d'une vue interactive. Les données présentes sont des métadonnées des passes, extraites automatiquement du code source de LLVM. La figure 2 montre des métadonnées typiques pour la passe `loops` ; les différentes propriétés sont décrites dans les sections 2.1 et 2.2.

```
LoopInfoWrapperPass:
  analysis: true
  arg: loops
  file: Analysis/LoopInfo.cpp
  parent: FunctionPass
  preserves: [(all)]
  requires: [DominatorTreeWrapperPass]
```

Figure 2 – Exemple de métadonnées pour la passe `loops`.

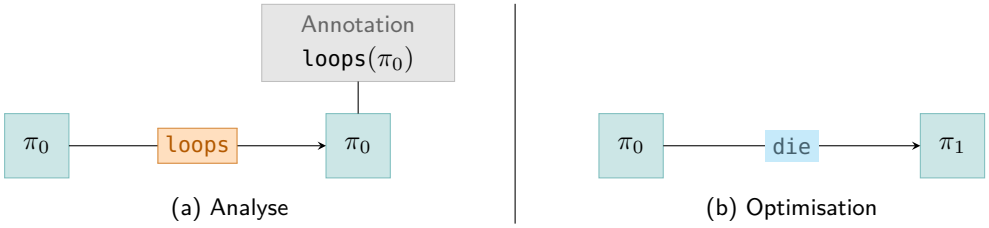
L'enjeu ici est de comprendre les relations fondamentales entre les passes, notamment les *dépendances* qui régissent la façon dont l'information circule, et ce qu'elles nous apprennent qui est utile au développeur de passes. On veut par exemple estimer si améliorer une passe aura des conséquences sur d'autres passes, ou trouver des passes qui ne s'influencent pas l'une l'autre.

2.1 Analyses et optimisations, dépendances

La première information disponible, et la plus importante, est la distinction entre deux genres de passes :

- Les **passes d'analyse**, qui analysent ou prédisent les propriétés du programme sans le modifier. Leur rôle est de collecter des données qui sont annotées sur le programme puis utilisées lors des transformations. Je les note en orange, par exemple `loops`.
- Les **passes d'optimisation**, qui transforment le programme en utilisant les informations collectées par les analyses. L'objectif est typiquement de rendre le programme plus rapide. Je les note en bleu, par exemple `loop-sink`.

Toute passe est d'un de ces deux types exclusivement.



(a) Analyse (b) Optimisation

Figure 3 – Représentation schématique d'une analyse et d'une optimisation.

Une autre notion cruciale est celle de **dépendances entre passes** ; elle permet à chaque passe d'indiquer quels résultats d'analyse elle utilise. C'est cette information qui nous indique qu'une analyse est souvent utilisée. Lorsqu'on veut améliorer une passe, une piste sérieuse est toujours d'améliorer

les analyses dont elle dépend. Et pour une optimisation, on peut chercher à exploiter des analyses supplémentaires pour affiner les décisions.

L'optimiseur de LLVM s'assure que les résultats d'analyse nécessaires pour une optimisation sont disponibles quand l'exécution commence. Cela nécessite de recalculer de temps en temps les analyses, car transformer le programme invalide souvent les annotations. Pour limiter le coût des calculs, les optimisations peuvent déclarer **préserver des analyses**. Par exemple, `die` préserve `loops` ; cela signifie que lorsque `die` transforme un programme π_0 en un programme π_1, l'annotation $\texttt{loops}(\pi_0)$ est identique à l'annotation $\texttt{loops}(\pi_1)$, et la structure de données générée en interne peut être réutilisée. L'optimiseur exploite cet avantage pour éviter des calculs et accélérer le compilateur lui-même.

Grâce à ces informations, l'optimiseur est capable de choisir quand calculer les analyses pour exécuter une séquence d'optimisations. L'ordre exact choisi ne change pas le programme final car les analyses ne modifient pas le programme ; on assimile donc souvent séquence de passes et séquence d'optimisations [3]. En-dehors de la section 5, je m'intéresse uniquement aux optimisations et laisse à LLVM la charge d'ordonnancer les analyses.

2.2 Niveaux de passes et indépendance

La distinction entre les passes d'analyse et d'optimisation nous renseigne sur le rôle de chaque passe, mais pas sur ce qu'elle fait du programme. Pour cela, on peut s'intéresser au niveau du programme auquel elle opère. Dans le code de LLVM, chaque passe hérite d'une classe de passe ; il en existe plusieurs qui ont des niveaux de finesse et contrats différents.

- Les passes les plus générales sont celles sur les modules ; elles opèrent de façon arbitraire sur les fichiers entiers, et sont les seules à pouvoir créer ou supprimer des fonctions.
- On a ensuite des passes plus fines comme celles sur les fonctions, qui sont appelées une fois par fonction dans le fichier et ne peuvent pas effectuer d'optimisations interprocédurales ni créer ou supprimer de fonctions. Hériter de `FunctionPass` implique une certaine sémantique de « localité » très utile pour appréhender le rôle de la passe.
- Il existe des niveaux encore plus fins comme les passes sur les boucles et même les blocs de base (nœuds du graphe de flot de contrôle, voir la section 4.1).

Plus le niveau est fin et plus les passes opèrent sur des petites sections de code avec des actions limitées, ce qui les rend plus susceptibles d'interagir (en particulier de modifier la même section du code). À l'inverse, des actions effectuées à différents niveaux sont souvent de nature différentes, et donc moins susceptibles d'interagir. Cette partition donne donc des premiers candidats de passes « indépendantes ».

Avoir des passes indépendantes est un résultat fort. Cela réduit la taille de l'espace de recherche du problème d'ordonnancement des passes et ouvre la voie à des approches par *clustering* [4]. Cela simplifie aussi (au moins localement) des interactions qui peuvent être complexes à décrire. Pour le développeur de passes qui cherche à comprendre comment une nouvelle passe s'intègre dans le système existant, l'indépendance est un des résultats les plus forts que l'on peut obtenir.

2.3 Exemple de séquence

Pour illustrer le rôle des différentes propriétés des passes, considérons la figure 4 qui montre un exemple simplifié pour l'exécution de deux passes d'optimisation, `licm` et `loop-unroll`.

3. Une des exceptions concerne les analyses d'alias ; il en existe une dizaine qui essaient toutes de calculer la même information (le recouvrement de pointeurs). Les spécifier explicitement permet de choisir lesquelles utiliser.

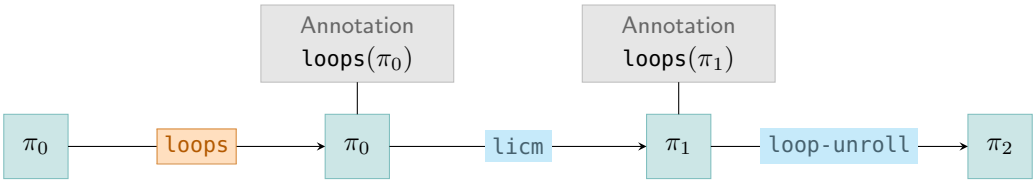


Figure 4 – Exemple d'une séquence de trois passes.

La première identifie dans les boucles du code invariant pour le déplacer à l'extérieur (*loop hoisting*), tandis que la seconde déroule les boucles pour limiter le nombre de sauts. Pour les exécuter, il faut d'abord identifier les boucles et donc exécuter l'analyse `loops`, qui annote le programme π_0 avec la liste des boucles.

On exécute alors `licm`, qui ne fait que déplacer du code et préserve la forme des boucles. On obtient un programme π_1 dont l'annotation `loops`(π_1) est identique à `loops`(π_0). On continue avec `loop-unroll`. Comme les petites boucles sont susceptibles d'être supprimées, l'annotation n'est pas préservée, et on obtient un programme π_2 sans annotation.

2.4 Limitations des métadonnées

Les métadonnées des passes présentées dans la figure 2 sont spécifiées dans le code par les développeurs, avec leur expérience du métier. Ce processus manuel a cependant des limites :

- À part dans des cas restreints comme l'analyse d'alias qui possède une interface unifiée, les passes ne bénéficient pas automatiquement d'améliorations dans les analyses ; il faut modifier le code.
- Les informations de préservation sont indiquées à la main et par nature incomplètes. Par exemple, pour prendre en compte l'ajout ou la mise à jour d'une analyse, il faudrait réconsidérer les métadonnées de toutes les optimisations. Il existe des cas spéciaux « préserve le graphe de flot de contrôle » (voir la section 4.1) et « préserve tout » mais ça ne couvre qu'une partie du besoin.
- De plus, ces informations ne sont pas directement vérifiables. Si une erreur se glisse dans la spécification, le compilateur peut produire du code erroné, et le bug serait difficile à détecter.

Ces limites n'affectent pas en tant que telle notre compréhension des passes, mais il est possible de les repousser de façon semi-automatique en cherchant les passes susceptibles de bénéficier d'une nouvelle analyse, ou en établissant empiriquement une liste « d'analyses préservées candidates » pour certaines optimisations. C'est une piste intéressante à explorer.

2.5 Exploitation de la carte dans Neo4j

La base de données extraite du code est une structure de données facile à utiliser dans les programmes, mais elle n'offre aucune méthode de requêtage et aucun outil de visualisation. Pour bénéficier de ces techniques, j'ai importé les données dans une base de données graphe, Neo4j [4].

Dans cette base, les nœuds du graphe sont les passes. Ils ont chacun un type, qui est `AnalysisPass` pour les passes d'analyse et le niveau de la passe pour les optimisations : souvent `ModulePass`, `FunctionPass` ou `LoopPass`. Les arêtes du graphe représentent les relations et sont également typées pour distinguer les dépendances, les analyses préservées, et d'autres types rares de relations que je n'ai pas évoqués.

Neo4j possède une interface web dans laquelle on peut requêter la base de données en langage Cypher et visualiser les résultats de façon interactive. Les détails de l'outil dépassent largement le cadre de ce rapport ; voici seulement quelques exemples pour suggérer la facilité d'accès à l'information.

4. https://neo4j.com/

- Obtenir toutes les dépendances de `licm` jusqu'au second niveau (figure 5) : typiquement utile pour comprendre ce que la passe étudie dans le programme et quelles pistes explorer pour l'améliorer.

```
match (p {name: "licm"})-[:requires*0..2]->(q) return p, q;
```

- Lister les 5 analyses qui sont la cible du plus grand nombre de relations de dépendance (figure 6) : cela permet d'identifier des points critiques dans le procédé d'optimisation.

```
match (p)-[:requires]->(q: AnalysisPass)
return q.name, count(p) order by count(p) desc limit 5;
```

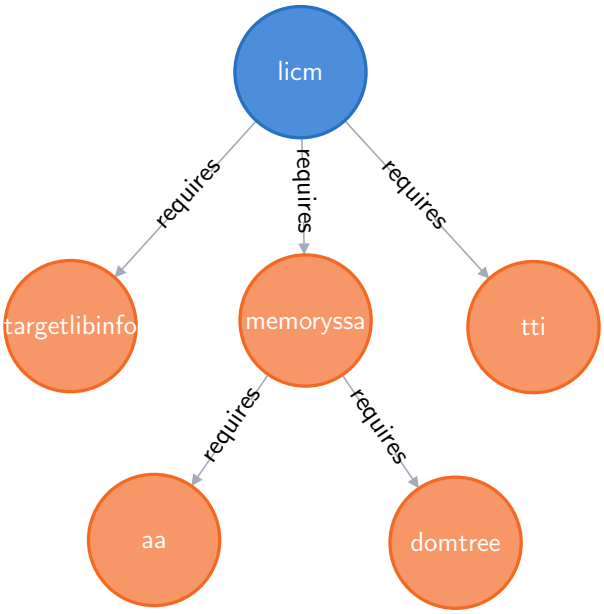


Figure 5 – Dépendances de `licm` jusqu'au deuxième niveau.

Analyse	Utilisateurs	Description
`domtree`	66	Construction de l'arbre des dominateurs
`targetlibinfo`	52	Liste des fonctions LLVM disponibles sur chaque architecture
`act`	38	Passe interne traquant les hypothèses en cours
`tti`	31	Coût des instructions LLVM pour chaque architecture
`loops`	29	Détection des boucles dans le graphe de flot de contrôle

Figure 6 – Les 5 analyses possédant le plus d'utilisateurs directs.

Cette carte interactive constitue la base de mon exploration des passes de LLVM. Le but des sections suivantes est de la compléter avec des relations indiquant comment les passes interagissent les unes avec les autres.

3 Mesure de l'impact en performances

L'enjeu de l'étape d'optimisation est généralement de rendre le programme produit plus rapide. Pour un développeur, évaluer la qualité d'une passe ou d'une modification d'une passe nécessite ultimement de tester les performances des programmes compilés. Il faut de plus être capable de détecter des petites différences de performance car l'impact d'une seule passe est toujours limité.

Ces tests permettent parfois de détecter que les performances attendues ne sont pas atteintes ; par exemple les analyses de pointeurs implémentées dans LLVM 3.5 se comportaient très mal avec les boucles intensives sur les tableaux [6, chapitre II.4]. Être capable de diagnostiquer la régression en performances est essentiel pour résoudre ce type de problèmes.

La mesure naturelle pour ce test est donc le temps d'exécution des programmes. Compte tenu de la grande variété des cas traités par les passes, il est d'usage de considérer des ensembles de programmes formant des bancs de test (*benchmarks*) qui couvrent une large gamme de situations pour évaluer les performances du compilateur, du système, ou du processeur.

Le projet LLVM possède sa suite de tests utilisée pour effectuer les tests de non-régression du compilateur, et une partie de la suite est utilisée en tant que *benchmark* de performance. Elle inclut des *benchmarks* classiques comme PolyBench [5] et l'ancien FreeBench [6], des tests contribués par différents utilisateurs ou universités, et des applications complètes comme des simulations de physique et un anti-virus. C'est cette partie que j'appelle « suite de tests » dans la suite.

3.1 Protocole de mesure des performances

Julian Bruyat fournit dans ses outils des programmes pour compiler la suite de tests de LLVM avec des options d'optimisation personnalisées, grâce à un système qui réécrit les règles de compilation avant de compiler. J'y ai ajouté des outils pour tester les programmes avec l'outil de mesure `perf` et collecter des statistiques. Il y alors cinq étapes pour tester la performance d'une séquence d'optimisations :

1. Configurer la suite de tests.
2. Réécrire les règles de compilation, notamment pour indiquer la séquence de passes à appliquer.
3. Compiler l'ensemble des programmes.
4. Configurer un système de test utilisant l'outil `perf` [7] qui est capable de mesurer de nombreuses statistiques sur l'exécution du programme à l'aide du noyau.
5. Exécuter les programmes et collecter les statistiques.

Le choix important ici est l'outil de mesure de performances. LLVM possède son testeur intégré, `llvm-lit`, mais il n'est adapté que pour les tests de non-régression et ne mesure le temps d'exécution qu'en calculant la différence d'heure entre avant et après le lancement du processus. `perf` est un outil plus fiable car il obtient ses statistiques avec la collaboration du noyau du système d'exploitation, qui lui-même utilise les compteurs de performance du processeur. Ce placement au bas niveau permet d'avoir des informations précises et peu bruitées, dont trois sont notables ici :

- Le **temps d'exécution du programme**, qui est le temps écoulé entre le lancement et l'arrêt du programme (*wall-clock time*, par opposition au temps total d'exécution qui se multiplie si plusieurs cœurs sont utilisés) ;
- Le **nombre d'instructions exécutées**, ce qui diffère du temps d'exécution car toutes les instructions ne prennent pas le même temps à s'exécuter, en particulier lorsqu'il faut accéder à la mémoire ;

5. https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/
6. https://web.archive.org/web/20061201010111/http://www.freebench.org/
7. https://github.com/brendangregg/perf-tools

- Le **nombre de cycles d'horloge** écoulés pendant l'exécution.

La figure 7 montre un exemple typique de statistiques d'exécution pour la commande `sleep 1` de Linux. Il est intéressant de noter que le temps n'est pas tout à fait égal à 1 seconde car le processus ne démarre pas immédiatement ; heureusement, pour deux variantes du même programme ce coût peut être considéré comme constant (car la taille du binaire et la liste des bibliothèques à charger ne change quasiment pas) et ne gêne donc pas la *comparaison* entre les exécutions.

```
% perf stat sleep 1
 Performance counter stats for 'sleep 1':

          379,572 cycles:u          # 0.766 GHz
          269,554 instructions:u    # 0.71  insn per cycle

       1.001916142 seconds time elapsed
```

Figure 7 – Extrait du rapport d'exécution de `perf`.

3.2 Limitations de la mesure du temps d'exécution

Cette section montre les résultats que l'on peut obtenir en appliquant le protocole ci-dessus à la passe `loop-simplify`. On mesure l'accélération des programmes en termes de temps d'exécution (temps d'exécution des programmes compilés sans optimisations divisé par le temps d'exécution des programmes compilés avec `loop-simplify`) pour un certain nombre de répétitions.

Pour la plupart des programmes, j'ai fait les statistiques sur 5 exécutions pour limiter le temps de test des 300 binaires de la suite. Sur mon ordinateur personnel équipé d'un Core i3, le test complet nécessite environ trois quarts d'heure. La figure 8 montre l'accélération obtenue et l'écart-type sur 5 mesures, les programmes étant triés par accélération moyenne croissante.

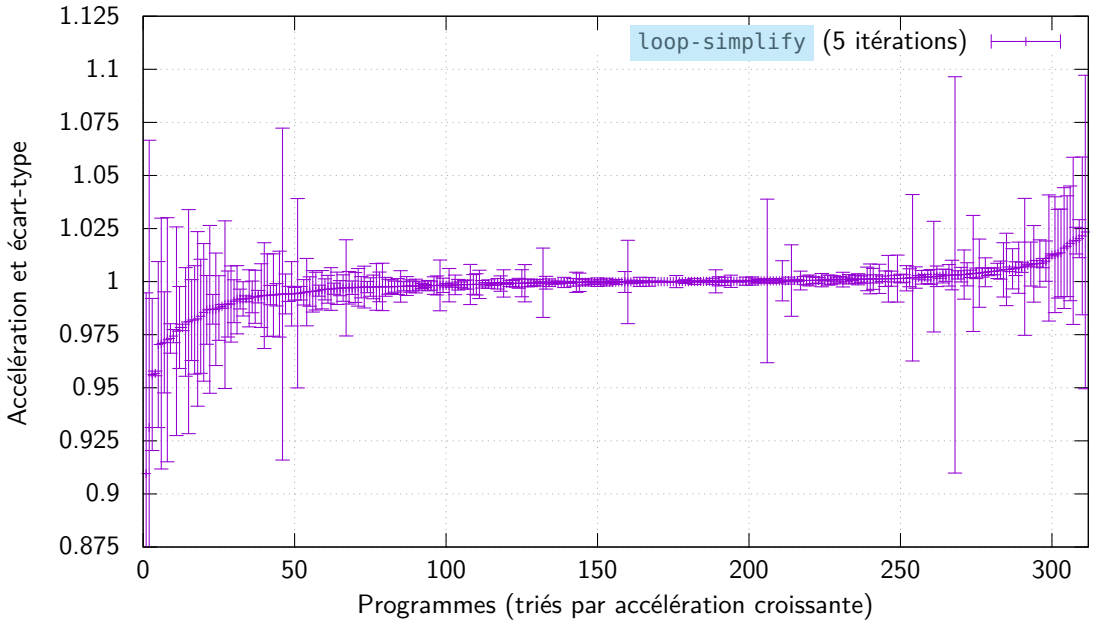


Figure 8 – Accélération des programmes de la suite de tests sous `loop-simplify`.

D'après la longueur des intervalles-type, il est clair que **cette mesure de temps est trop imprécise** pour repérer l'accélération d'une seule optimisation. Ce n'est pas entièrement inattendu car l'accélération complète de `O3` (qui contient 60 transformations) se mesure généralement entre 20% et 30%, donc l'effet d'une seule transformation se perd dans le bruit de mesure.

On notera également la symétrie de la courbe, indiquant que **les programmes sont aussi bien ralentis qu'accélérés**. Les chiffres ne sont pas significatifs ici à cause des grands écarts-type, mais on retrouve cet effet de façon consistante sur des mesures plus fines.

3.3 Utilisation des ressources de Calcul Canada

Le problème de mesure de performances est connu pour être subtil. J'ai réalisé mes mesures sur mon ordinateur personnel dans l'environnement le plus contrôlé possible (pas de serveur graphique, pas d'autre processus que le test, un cœur libre pour absorber les fluctuations de charge) mais la complexité des systèmes électroniques et d'exploitation rend très difficile d'isoler précisément l'exécution d'un seul programme.

Une tentative pour résoudre ce problème a été d'utiliser les ressources de Calcul Canada [8] pour effectuer des tests de performance. Il s'agit d'une grille de calcul Canadienne disposant de plusieurs grappes ; la grappe Cedar [9] que j'ai utilisée dispose essentiellement de nœuds de 32 cœurs avec 128 Go de RAM, équipés de processeurs Intel Broadwell. À part les répertoires temporaires en RAM, tout l'espace de stockage est monté en réseau.

L'environnement d'exécution des serveurs est plus contrôlé que celui de mon ordinateur personnel, mais deux problèmes se posent :

- La latence élevée des disques montés en réseau ne permet pas de profiter pleinement de la puissance de calcul de la grille. Dans l'ensemble la vitesse d'exécution des tâches était comparable à celle de ma machine (45 minutes pour un test complet avec 5 répétitions), ce qui n'est pas suffisant pour collecter des statistiques sur un grand nombre de répétitions car il y a beaucoup de séquences d'optimisations à tester.
- L'administrateur des serveurs n'autorise pas les utilisateurs non privilégiés à utiliser l'interface d'événements du noyau par laquelle `perf` collecte ses statistiques. Il fallait donc se contenter des informations fournies par l'ordonnanceur du noyau, qui se limitent au temps d'exécution.

Cela m'a amené à considérer un autre critère de performance suffisamment précis pour détecter des variations sur une seule transformation, décrit dans la prochaine section.

3.4 Résolution par le nombre d'instructions

Des trois mesures rapportées par `perf` qui sont liées de près ou de loin au temps d'exécution, celle qui est la mieux définie et la plus stable en pratique est le nombre d'instructions exécutées. C'est une mesure assez éloignée du temps d'exécution car les instructions sont exécutées dans le désordre (*out-of-order execution*), à l'avance (prédiction de branchements), ne prennent pas toutes le même temps (accès mémoire) et n'ont même pas un temps d'exécution fixe (caches).

Dans le même temps, c'est une mesure qui abstrait tout autant de détails matériels certains de varier dès qu'on change de version du processeur, de RAM, ou de paramètres d'alimentation. Le compromis me paraît pertinent pour évaluer la qualité de l'optimisation, une opération statique indépendante du matériel équipant les nombreuses machines susceptibles d'utiliser les binaires, d'autant plus qu'il ne semble pas y avoir de consensus autour d'une « bonne » méthode de mesure.

8. https://www.computecanada.ca/about/
9. https://docs.computecanada.ca/wiki/Cedar

La figure 9 montre l'accélération de `loop-simplify` en nombres d'instructions exécutées. On retrouve le même profil avec une majorité de programes non transformés, et des programmes accélérés comme ralentis, avec cette fois des écarts-type bien plus fins.

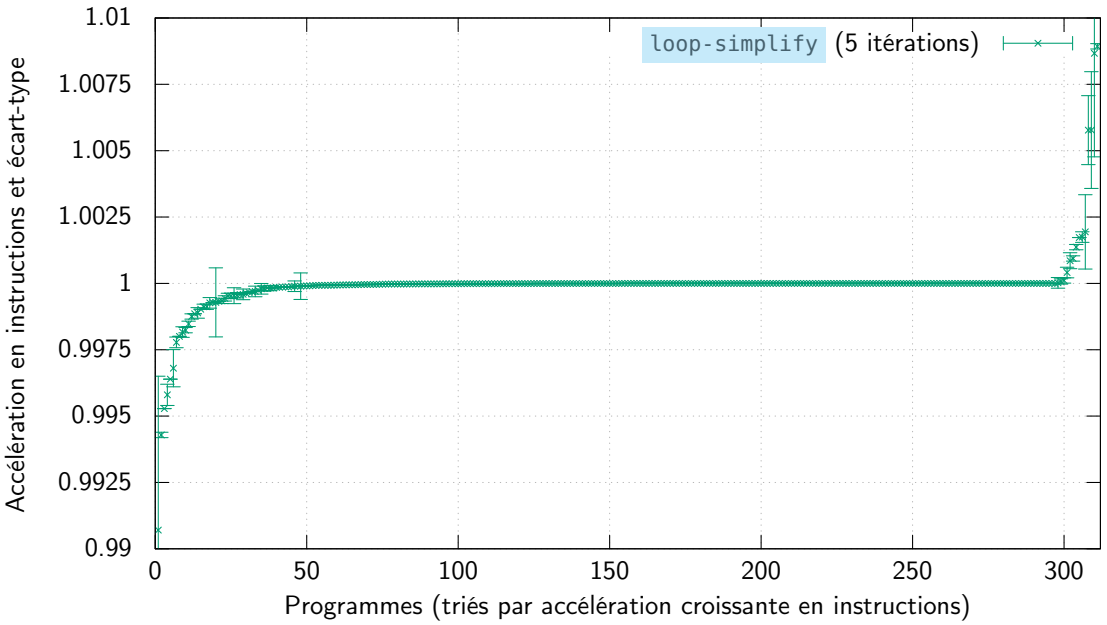


Figure 9 – Accélération en nombre d'instructions sous `loop-simplify`.

Ce phénomène de ralentissement est extrêmement intéressant et fait partie des perspectives, notamment pour comprendre quand et comment une transformation peut ralentir un programme ; et bien entendu car cela révèle une opportunité d'améliorer le compilateur. Cela se produit sur quasiment toutes les transformations inviduelles et semble indiquer que la meilleure séquence d'optimisations dépend beaucoup du programme d'entrée, justifiant les approches d'apprentissage machine pour la résolution du problème d'ordonnancement des phases. [3]

La vitesse d'exécution des programmes compilés est l'objectif de toute la séquence d'optimisation ; la mesurer est donc une étape importante dans tout travail sur les passes. Cette information n'était jusque-là accessible que de façon très imprécise dans le testeur intégré de LLVM. Un outil de mesure plus fin permet de mieux comprendre l'impact d'une modification de passe sur les programmes et donc de mieux apprécier le système d'optimisation dans son ensemble.

4 Conflits et contributions d'édition

Pour comprendre la façon dont les passes opèrent, mesurer la performance des programmes n'est pas suffisant. C'est en lisant le code qu'on peut comprendre spécifiquement ce que chaque transformation fait. La notion de *différence de programme LLVM* (que j'abrégerai *diff*) formalise l'idée d'une transformation en termes d'opérations sur le code.

La structure des programmes LLVM ne se limite pas à leur représentation textuelle. À l'instar d'outils comme Gumtree [7], je cherche des informations de haut niveau telles que « créer une fonction `f()` », et non des informations textuelles telles que « insérer le texte "`define i32 @f() {}`" ». Une telle notion de *diff structuré* existe déjà dans LLVM grâce à l'outil `llvm-diff`. J'ai exploré les limites de l'outil et du format, puis les ai modifiés pour l'appliquer à la détection de *conflits d'édition* entre des passes.

4.1 Format des programmes LLVM

Les programmes LLVM sont représentés sous forme de graphes de flot de contrôle (CFG) en forme SSA (*Static Single Assignment*). Le CFG représente les différents chemins d'exécution sous la forme d'un graphe : chaque nœud est un bloc d'instructions exécutées de haut en bas, et se termine par un saut (ou l'instruction de retour de fonction). On place une arête d'un bloc B vers un bloc B′ si l'exécution de B se terminer par un saut vers B′.

À titre d'illustration, la figure 10 montre le CFG (pas en forme SSA) de l'algorithme d'Euclide naïf. On voit le flot se séparer après le test du `if` (bloc B_2) et former un cycle autour de la boucle `while`. L'analyse `loops` sert à détecter ce genre de cycles.

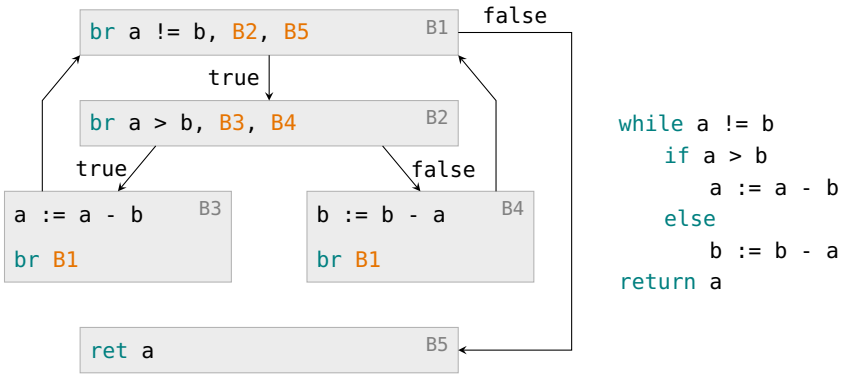


Figure 10 – Graphe de flot de contrôle et pseudo-code de l'algorithme d'Euclide naïf.

Un programme est dit en forme SSA lorsque chaque variable n'est assignée qu'à un endroit du code ; si on réassigne, on crée une nouvelle variable. Cette forme unique est utilisée par toutes les passes, ce qui simplifie grandement le développement et l'intégration de nouvelles passes dans LLVM. Dans ce rapport, la conséquence importante est que LLVM identifie chaque variable avec l'instruction qui lui assigne sa valeur, donc les variables dans les opérandes de calcul sont en fait des références vers des instructions. Cela complique les raisonnements sur le code, par exemple dans les analyses *dataflow* qu'il faut adapter car la notion de variable est perdue [8].

4.2 Notion de diff de programmes LLVM

Pour formaliser le passage d'un programme à un autre lors de l'exécution d'une transformation, on veut maintenant calculer pour deux programmes LLVM une séquence d'opérations transformant le premier

en le second. Cela se fait d'abord au niveau du CFG avec des opérations de graphe, puis dans chaque bloc avec des opérations proche de texte. Pour chaque fonction, on liste :

- Les blocs du graphe ajoutés, supprimés, et modifiés.
- Pour chaque bloc modifié, les instructions ajoutées, supprimées et modifiées entre l'ancienne et la nouvelle version.

Le diff d'un bloc seul fonctionne sur le même principe qu'un diff de texte ; on identifie un ensemble d'instructions inchangées (par exemple avec un calcul de plus longue sous-séquence commune), et ce qui reste est soit supprimé, soit inséré, soit modifié. La figure 11 montre un exemple simplifié où une multiplication et une addition sont réordonnancées, et la valeur de retour modifiée.

Il est important de voir ici que `%1` et `%2` sont des noms pour les instructions et non les variables, car la forme SSA est utilisée. C'est pour cela qu'on **compare les programmes à α-équivalence près**. Mais unifier correctement les instructions est souvent difficile.

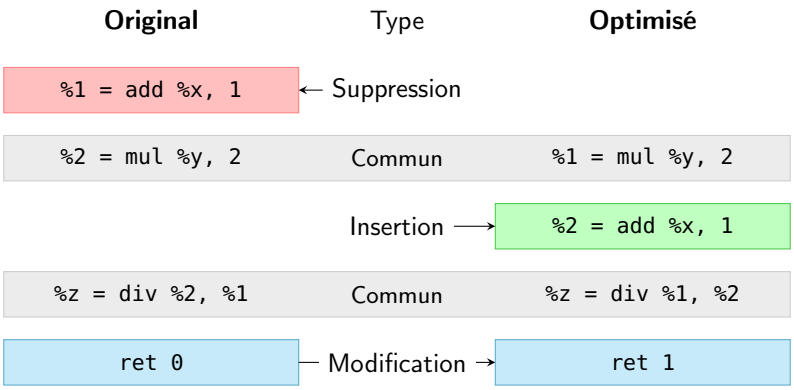


Figure 11 – Diff pour le réordonnancement de deux opérations.

L'outil `llvm-diff` implémente un algorithme de calcul de diff dont le principe est décrit dans la figure 12 pour deux fonctions f et g. C'est un parcours de graphe en largeur où les blocs sont unifiés au fur et à mesure ; pour deux blocs, le diff est un calcul de distance d'édition excepté que l'égalité d'instructions est remplacée par un critère d'unification.

Algorithme diff(f, g)

F ← File vide
Enfiler dans F la paire (bloc d'entrée de f, bloc d'entrée de g)
Tant que F est non vide
    Extraire la paire (B, B′) de F
    Calculer la distance d'édition entre B et B′ à α-équivalence près
    Afficher les instructions supprimées, insérées, et modifiées
    Si l'instruction finale est commune **alors**
        $B_1 \dots B_n \leftarrow$ Liste des destinations en sortie de B
        $B'_1 \dots B'_n \leftarrow$ Liste des destinations en sortie de B′
        Enfiler dans F les paires (B_1, B'_1) ... (B_n, B'_n)
    Fin
Fin

Figure 12 – Vue d'ensemble de l'algorithme de `llvm-diff`.

Il est important de voir que l'on essaie en fait de résoudre deux problèmes en même temps : calculer la distance d'édition, et trouver l'unification entre les instructions de gauche et de droite qui minimise cette distance. Contrairement au calcul de la distance de Levenshtein, ici les instructions peuvent ou non être unifiables selon les décisions précédentes sur leurs opérandes. `llvm-diff` ignore en partie cette situation et peut être mis en défaut sur des exemples construits à la main, mais qui n'apparaissent quasiment jamais en pratique.

À l'origine, la sortie de `llvm-diff` est uniquement prévue pour être lue par des humains, et il n'y a pas de patch associé. L'algorithme ne semble décrit ou prouvé dans aucune publication. J'ai modifié l'outil pour pouvoir effectuer des traitements automatiques sur les diffs, notamment en fournissant plus d'informations permettant d'évaluer la **distance entre les programmes**, et en ajoutant un système d'identification unique des instructions pour discriminer des instructions identiques à différents endroits du code.

4.3 Conflits

Les diffs sont un outil puissant pour analyser le travail effectué par une transformation sur un programme. Comme première approche, j'ai cherché à identifier des situations où des passes sont « indépendantes » comme discuté dans la section 2.2.

Une première notion d'indépendance est la **commutativité des transformations**, par exemple avoir pour deux optimisations `p` et `q`, $p \circ q = q \circ p$. C'est cependant un résultat très fort avec plusieurs défauts : il est issu de la comparaison de deux programmes et ne donne pas beaucoup d'information sur le comportement des passes, et il est rare.

À la place, j'ai utilisé une notion classique de **conflit** sur les diffs. Les définitions varient, mais l'idée intuitive d'un conflit est que deux modifications ne peuvent pas être appliquées simultanément. Dans le gestionnaire de version Pijul [10] par exemple, les patchs qui ne commutent pas sont dits en conflit. Ici, je ne sais pas calculer la commutation purement à l'aide des diffs (pour des raisons détaillées plus loin) donc je définis manuellement les cas de conflit :

- Si un bloc d'instructions du CFG est supprimé d'un côté et modifié de l'autre ;
- Si dans un bloc, un groupe de lignes *modifié* (au sens de la figure 11) d'un côté est partiellement supprimé de l'autre ou le lieu d'une insertion de l'autre ;
- Si dans un bloc, deux insertions ont lieu au même endroit.

La figure 13 montre comment les conflits sont définis en termes de diffs : pour deux passes p et q et un programme π, on calcule les diff entre π et ses versions optimisées $p(\pi)$ et $q(\pi)$, notés $\Delta\big(\pi, p(\pi)\big)$ et $\Delta\big(\pi, q(\pi)\big)$. On détermine ensuite l'intersection entre ces deux diffs, c'est-à-dire l'ensemble des conflits sur tout le fichier.

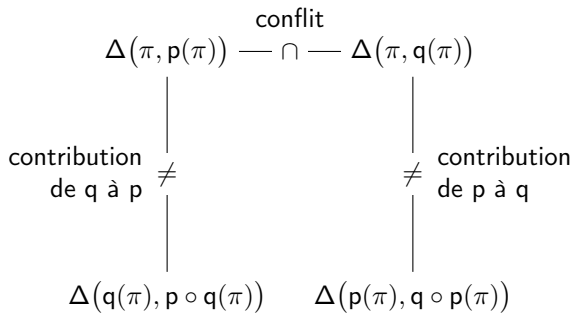


Figure 13 – Représentation des conflits et contributions en termes de diff.

10. https://pijul.org/

L'intérêt de cette méthode par rapport à la commutativité est double : d'une part on dispose du code qui provoque un conflit, d'autre part on peut choisir une notion simple de « taille » de conflit (par exemple le nombre d'instructions dans chaque zone de conflit) et en déduire un **calcul de distance entre diffs**. L'indépendance de deux passes n'est donc plus une propriété binaire difficile à obtenir en pratique à cause des nombreux détails techniques des passes, mais une mesure numérique. Elle nous donne des informations sur la compatibilité des transformations, bien qu'elle n'indique rien sur la vitesse du programme.

4.4 Contributions

Cependant, ces conflits ne sont pas la même relation que la commutativité et ne sont pas la seule relation intéressante ; on trouve aussi l'idée de **contribution**, qui modélise les situations où une transformation p produit du code qu'une autre transformation q peut optimiser. Cela représente l'idée intuitive que p doit être exécutée avant q. C'est là aussi une information précieuse que l'on souhaite pouvoir fournir aux développeurs de passes qui intègrent leur travail dans le système.

L'idée de contribution est notée sur la figure 13 et formalisée par une comparaison de diffs : d'un côté l'action de p sur le programme original $\Delta\big(\pi, \mathsf{p}(\pi)\big)$, et d'un autre son action sur le programme déjà optimisé par q, $\Delta\big(\mathsf{q}(\pi), \mathsf{p} \circ \mathsf{q}(\pi)\big)$. Si ces deux diffs sont différents alors le fait d'exécuter préalablement q contribue au travail effectué par p.

Mais cette comparaison n'est pas facile : en effet, $\Delta\big(\pi, \mathsf{p}(\pi)\big)$ est un ensemble d'opérations à effectuer sur π (pour obtenir $\mathsf{p}(\pi)$), tandis que $\Delta\big(\mathsf{q}(\pi), \mathsf{p} \circ \mathsf{q}(\pi)\big)$ est un ensemble d'opérations à effectuer sur $\mathsf{q}(\pi)$ (pour obtenir $\mathsf{p} \circ \mathsf{q}(\pi)$). Transposer les opérations d'un programme vers un autre est une opération appelée **rebasage**. Elle est courante par exemple dans Git, mais :

- Rebaser n'est pas toujours possible ; si le système est bien construit alors la possibilité est équivalente à l'absence de conflits. Mais l'absence totale de conflits risque d'être un cas minoritaire.
- L'algorithme de rebasage pour le code LLVM nécessite de résoudre à nouveau les problèmes d'α-équivalence qui existaient lors du calcul du diff. C'est-à-dire qu'il faut nouveau résoudre deux problèmes en même temps, et dans un cas plus subtil que celui du calcul du diff.

Compte tenu de la complexité algorithmique de la tâche et de la portée de ce stage, je me suis contenté de la détection de conflits pour évaluer l'indépendance entre des passes d'optimisation. L'étude d'un algorithme de diff et d'un algorithme de rebasage qui traitent proprement de l'α-équivalence est une perspective future.

4.5 Application à licm et loop-unswitch

Pour appliquer mes outils de calcul de diff, j'ai choisi d'étudier `licm` et `loop-unswitch`, deux passes connues pour devoir s'exécuter dans cet ordre exact. En effet, la seconde transforme les boucles contenant des conditions invariantes, et c'est la première qui est chargée de détecter tout le code invariant.

Séquence 1	Séquence 2	Fichiers inchangés	Distance
`O3`	(rien)	49 sur 1970	6516957
`loop-unswitch`	(rien)	1682 sur 1970	66491
`licm loop-unswitch`	`licm`	1683 sur 1970	10854
`licm loop-unswitch`	`loop-unswitch licm`	1685 sur 1970	11903

Figure 14 – Statistiques de `licm` et `loop-unswitch` sur des diffs.

À strictement parler, c'est une *contribution* et je ne sais pas la calculer. Mais la distance entre programmes obtenue à partir des diffs permet déjà de faire des observations. La figure 14 compare différentes paires de séquences et montre le nombre de fichiers identiques pour les deux optimisations, ainsi que la distance totale estimée à partir des diffs.

Les trois dernières lignes du tableau montrent quasiment les mêmes fichiers modifiés pour différentes positions de `loop-unswitch`, ce qui est inattendu. De plus, la distance entre les programmes d'origine et les programmes optimisés est plus importante lorsqu'elle est exécutée *sans* `licm`, ce qui contredit nos attentes. Plusieurs pistes peuvent expliquer ce problème :

- L'acquisition du diff ou le calcul de la distance contient un défaut caché. C'est l'hypothèse conservative, mais elle ne permet pas d'expliquer les nombres quasi-identiques de fichiers inchangés (car `llvm-diff` ne se trompe pas quand le calcul du diff aboutit).
- `licm` ne parvient pas à prouver que les conditions intéressantes pour `loop-unswitch` sont invariantes sur les programmes de la suite de tests ; dans ce cas il peut y avoir de la surface d'optimisation en améliorant `licm`.
- La suite de tests ne contient tout simplement pas assez de conditions invariantes pour couvrir les cas d'usage de `loop-unswitch` ; dans ce cas il est possible d'étendre la suite de tests pour couvrir une plus grande partie du code de LLVM.

Ce cas particulier démontre la nécessité de construire des outils pour détecter des interactions ou non-interactions inattendues, susceptibles de cacher des problèmes d'implémentation. La combinatoire de 80 passes est trop vaste pour être explorée à la main, mais on peut chercher des pistes de façon automatique.

5 Application : Étude de l'impact d'une analyse

Cette section montre une application simple des outils que j'ai construits pour mesurer **l'impact d'une analyse** sur les transformations qui suivent. C'est un problème pragmatique pour le développeur de passes, à la fois pour tester l'efficacité de son travail et pour justifier de son intérêt à la communauté LLVM qui intègre les changements.

Je prends ici un exemple d'une passe importante, l'analyse d'alias. Cette analyse tente de prédire si deux pointeurs référencent la même zone mémoire. Une interrogation de la base de données graphe montre que la moitié des passes de `O3` s'en sert directement ou indirectement :

```
match (p {name: "aa"})<-[:requires*]-(q) return q;
```

Cette requête renvoie 9 passes d'analyse et 35 optimisations, on s'attend donc à observer un impact élevé si on modifie la qualité des résultats. Je l'ai modifiée pour renvoyer `MayAlias` (aucune information trouvée) à toutes les requêtes et donc la désactiver (une modification grossière). La figure 15 montre l'accélération de `O3` privé d'analyse d'alias par rapport à `O3`.

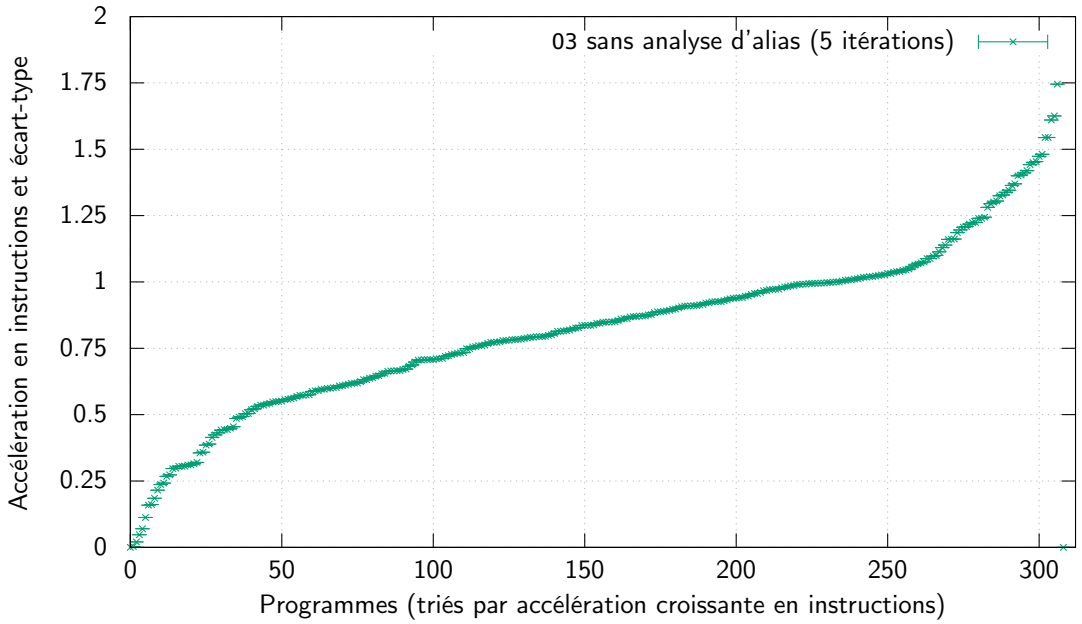


Figure 15 – Courbe de speedup de `O3` sans analyse d'alias par rapport à `O3`.

Sans surprise, c'est catastrophique pour la majorité des programmes, avec plus de la moitié des programmes perdant au moins 15% de vitesse. Certains programmes sont par contre plus rapides ; une hypothèse est que l'absence d'analyse d'alias a empêché `O3` de faire les transformations qui lui font d'habitude ralentir des programmes. Cela provient du phénomène de ralentissement observé à la fin de la section 3.

Bien que supprimer toute l'analyse d'alias soit une modification grossière, la section 3.4 montre que l'on peut détecter des variations de performance bien plus subtiles et donc tester facilement la pertinence des analyses pendant le processus de développement.

6 Perspective : Diff capturant la sémantique de l'édition

L'outil le plus formel dont je dispose pour étudier les transformations est le diff de programmes LLVM. Il est cependant difficile à calculer à cause de l'α-équivalence en jeu et bas niveau dans la structure de données. Il ne capture ni l'intention de la passe ni la sémantique de la transformation.

Une façon plus élégante de construire un diff tout en éliminant le problème de renommage serait d'instrumenter l'API pour **enregistrer les appels aux fonctions de transformation du programme** et ainsi capturer, au lieu de reconstruire, une séquence d'opérations menant du programme source au programme optimisé.

Le bénéfice de cette méthode est la remontée en abstraction qu'elle permet de faire. Là où le diff présenté dans la figure 11 ne trouve qu'une suppression et une insertion, intercepter un appel à la fonction `moveInstruction()` permet de retenir la sémantique de la transformation.

La figure 16 montre trois niveaux d'abstraction auxquels on peut considérer une même transformation, ici déplacer une instruction hors d'une boucle. En instrumentant l'API, on peut capturer automatiquement le niveau intermédiaire. Atteindre le niveau le plus élevé peut être fait en analysant l'ancienne et la nouvelle position de l'instruction déplacée, ou en modifiant la passe pour enregistrer l'intention de l'auteur.

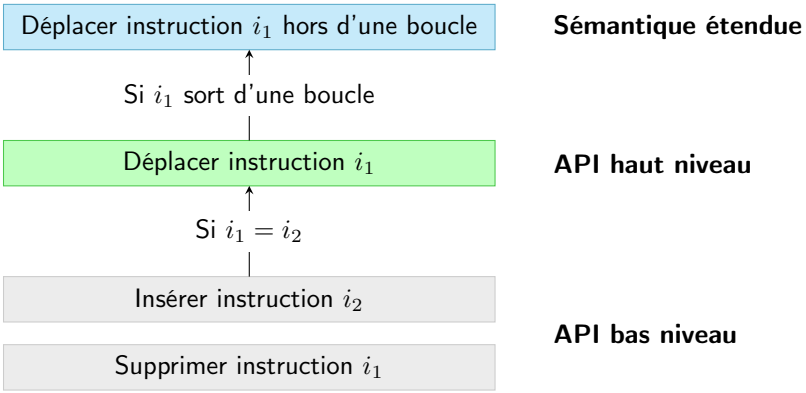


Figure 16 – Illustration des différents niveaux de sémantique d'une même opération.

Ce format paraît très prometteur car il est facile d'obtenir un enregistrement de l'API si l'on peut exécuter la passe, possiblement sans modifier le code de la passe. On peut ensuite construire des règles de réécriture pour remonter en abstraction et mieux expliquer le travail de la transformation.

Cela ouvre la perspective à des diagnostics semi-automatiques tels que « `licm` extrait 5% plus d'instructions des boucles lorsque la nouvelle analyse d'alias est utilisée », qui est une information très intéressante pour le mainteneur de la passe d'analyse d'alias.

De plus, **cette approche est entièrement incrémentale** et ne nécessite pas que des opérations à haut niveau d'abstraction ou des règles de réécriture soient spécifiées pour fonctionner. Au plus bas niveau, on retombe sur le diff usuel, et à partir de là, on peut construire pas-à-pas un format qui explique le comportement des passes par l'observation.

7 Conclusion

Durant ce stage, j'ai cherché à comprendre les problématiques de la recherche en compilation et du développement de passes avec LLVM. Le système d'optimisation est trop complexe pour être prouvé formellement, mais peut être étudié expérimentalement.

J'ai déployé des outils expérimentaux, notamment une analyse de code source et un protocole de mesure de performance, et des outils formels, en particulier des diffs de programmes LLVM, pour contribuer à répondre à des problématiques telles que :

- Identifier les points critiques de la chaîne d'optimisation, et leurs pistes d'amélioration ;
- Déterminer finement l'impact d'une passe ou d'une modification d'une passe sur la performance des programmes compilés ;
- Comprendre la transformation qu'une passe effectue sur un programme ;
- Détecter la présence d'interaction et de conflits entre plusieurs passes ;
- Vérifier une hypothèse sur l'interaction ou la non-interaction de plusieurs passes pour détecter des anomalies dans la chaîne d'optimisation.

Répondre à ces questions améliore notre compréhension du système d'optimisation du compilateur. On peut alors envisager des améliorations du système qui accélèrent encore les programmes optimisés.

7.1 Perspectives

Au fur et à mesure que j'ai exploré le sujet, de nombreuses pistes se sont présentées pour expliquer le comportement des passes ou simplifier le travail du chercheur et du développeur. Les outils que j'ai construits peuvent également être améliorés de différences façons. Parmi les perspectives principales, on notera :

- Construire un système de mesure de performances entièrement déterministe, posssiblement à l'aide d'un émulateur, pour normaliser les conditions de test, éliminer les variations, et obtenir une notion entièrement définie de performance.
- Proposer un algorithme pour `llvm-diff` qui tienne compte de l'α-équivalence et produise un diff sur toutes les entrées.
- Développer la notion de diff sémantique présentée dans la section 6. En plus d'éliminer le problème d'α-équivalence, ce diff explique structurellement mieux les transformations que celui étudié dans la section 4.
- Exploiter les propriétés (*features*) de programmes utilisées pour entraîner les modèles d'apprentissage machine pour caractériser automatiquement des classes de programmes. Cela permettrait par exemple de trouver (exemple factice) que « `loop-simplify` ralentit les programmes dont les boucles contiennent beaucoup d'appels de fonctions ».
- Étudier les situations où des optimisations ralentissent les programmes pour les rapporter à la communauté LLVM et les corriger.

A Utilisation des programmes

Une copie du dépôt construit avec Julian Bruyat est disponible sur ma page personnelle :

https://perso.ens-lyon.fr/sebastien.michelland/llvm/

Des fichiers README dans la plupart des répertoires fournissent le contexte dans lequel chaque programme est utilisé. Pour reproduire les expériences, il faut avoir ou compiler une version de LLVM. Le script `install.sh` s'en charge en plus de télécharger les sources utiles comme la suite de test.

Les fichiers utilisés ensuite sont majoritairement situés dans les dossiers suivants :

- **Dossier** `passdb`

  Ce dossier contient essentiellement le script `extract.py` qui extrait la base de données de passes du code source au format YAML. Il est ensuite utilisé pour convertir les données au format CSV. Ce format peut être importé dans Neo4j en copiant les requêtes de `passdb.cypher` dans l'outil interactif.

- **Dossier** `testsuite`

  Ce dossier contient le script `configure.py` qui est utilisé pour configurer la suite de test et réécrire les règles de compilation (étapes 1 et 2 de la section 3.1), et `configure-perf.py` qui est utilisé pour configurer les mesures de performance (étape 4). Il contient aussi toutes les versions compilées de la suite de test.

- **Dossier** `scripts`

  Ce dossier contient une variété de scripts utilisés pour compiler les programmes individuellement lors de la compilation d'une suite de tests. La plupart ne font qu'invoquer l'optimiseur avec différentes options.

- **Dossier** `perf`

  Contient un script générant un fichier de données pour Gnuplot à partir des mesures de `perf`.

- **Dossier** `llvm-diff`

  Ma version modifiée de `llvm-diff`, que l'on peut utiliser en faisant pointer `llvm/tools/llvm-diff` dessus.

- **Dossier** `diff`

  Les scripts dans ce dossier sont utilisés pour faire des calculs sur les diffs :
  - `delta.py` lit la sortie de `llvm-diff` dans une structure de données ;
  - `delta-size.py` affiche la taille d'un diff (distance entre programmes) ;
  - `delta-inter.py` calcule l'intersection de deux diffs (c'est-à-dire les conflits) ;
  - `testsuite-diff.sh` compare deux suites et donne les statistiques présentées à la figure 14.

La plupart des scripts affichent un message d'aide s'ils sont lancés sans argument.

Le dossier `notes` contient des notes accumulées avec Julian dans l'objectif de documenter le procédé de recherche.

Table des matières

Références

[1] C. Lattner and V. Adve, "LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.

[2] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," in *ACM SIGMICRO Newsletter*, vol. 13, pp. 125–133, IEEE Press, 1982.

[3] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, p. 96, 2018.

[4] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp : Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, p. 29, 2017.

[5] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, *et al.*, "Milepost gcc : Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[6] M. Maalej Kammoun, *Low-cost memory analyses for efficient compilers*. Theses, Université de Lyon, Sept. 2017.

[7] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and Accurate Source Code Differencing," in *Proceedings of the International Conference on Automated Software Engineering*, (Västeras, Sweden), pp. 313–324, 2014. update for oadoi on Nov 02 2018.

[8] H. Nazaré, I. Maffra, W. Santos, L. Oliveira, F. M. Quintão Pereira, and L. Gonnord, "Validation of Memory Accesses Through Symbolic Analyses," in *ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*, (Portland, Oregon, United States), pp. 791–809, Oct. 2014.