TER Magistère

Valeran MAYTIE

Table des matières

1 Introduction

Le langage C est très utilisé aujourd'hui. En effet, il possède de nombreuses qualités, il est bas niveau donc très utile pour faire de l'embarqué. Il y a aussi beaucoup d'optimisations faites certaines sont complexes et rendent le code plus rapide et plus court. De plus, il existe depuis suffisamment longtemps pour avoir des compilateurs comme GCC ou Clang avec très peu de bugs. Malheureusement, il est toujours difficile de savoir si ces compilateurs sont sûrs [Bla07], car ils font énormément de transformations sur le code. Comme le compilateur n'est pas du tout transparent sur les transformations faites, il est difficile de savoir si celles-ci sont correctes. On peut donc se demander si nous pouvons avoir confiance dans les compilateurs fréquemment utilisés comme Clang ou GCC. En effet, il n'est pas dit que notre compilateur ne crée pas de dysfonctionnement dans notre programme. Par dysfonctionnement on entend le fait qu'un programme plante sans raison apparente ou encore pire, que le code source généré n'a pas le même comportement que celui prévu par le langage. Donc même si nous souhaitons prouver nos programmes avec des outils comme Why3 [FP13], si le compilateur génère des bugs lors de la compilation, l'exécutable ne sera pas sûr.

On pourrait trouver une solution assez rapidement, à savoir, directement regarder le code machine généré par le compilateur et vérifier instruction par instruction la justesse du code. Malheureusement, on remarque assez vite que cette méthode peut être très longue (grosse quantité d'instructions à lire pour quelques lignes de code) et pas forcément correcte, car un humain peut facilement se tromper sur ce genre de tâche.

Par contre, on pourrait écrire un compilateur vérifié formellement, c'est-à-dire, qu'il est prouvé informatiquement que chaque transformation préserve le comportement du programme. C'est là qu'intervient CompCert [Ler09], un compilateur écrit à l'aide de l'assistant de preuve Coq [BC13]. Il permet de compiler des programmes venant d'un large sous-ensemble du langage C. Ainsi, à partir d'un code source S, CompCert génère un code machine C ayant la même sémantique que celui prévu par le langage pour le code source S. Pour cela, il passe par de nombreux langages de transition (aujourd'hui on peut en compter une vingtaine). Comme chaque compilation d'un sous langage à un autre sous langage est prouvée formellement, nous sommes certains que la sémantique du code est préservée.

Concevoir un tel compilateur est difficile. De ce fait, on peut voir que CompCert fait nettement moins d'optimisations et rivalise difficilement au niveau des performances avec les compilateurs modernes comme GCC ou Clang. En conséquence, un programme compilé par CompCert est plus sûr, mais certainement plus lent. On se demande alors : À quel point CompCert est plus lent qu'un compilateur optimisant ? Est-il possible de trouver des solutions pour résoudre ce problème ?

Chercher à optimiser CompCert peut être intéressant. En effet, avoir un compilateur formel aussi performant que les compilateurs actuels peut être une énorme avancée pour le monde de l'industrie : avoir du code machine compilé avec CompCert aussi rapide qu'avec des compilateurs normaux pourrait pousser les entreprises à utiliser CompCert. On n'aurait plus besoin de réfléchir si le fait d'avoir un programme plus sûr, mais plus lent est utile ou non, car les compilateurs actuels restent plus rapides.

Pour commencer, on a supposé que le compilateur optimisant donnait un exécutable plus performant que CompCert. À partir de cette supposition, nous avons commencé à comparer les instructions générées par CompCert et GCC. Ensuite, nous avons rapidement fait un outil qui permettait de comparer les deux exécutables (récupération des instructions utiles). Après avoir trouvé une optimisation, on a regardé si celle-ci n'implique pas de trop grands changements dans CompCert. Une fois la première optimisation faite, il a fallu voir si ce changement a eu des conséquences dans les performances. Nous avons donc fait un outil qui permet de comparer les performances (cycle et nombre d'instructions) entre les différentes versions de CompCert et d'un compilateur optimisant comme GCC. Nous avons donc des tableaux qui nous donnent la comparaison avant et après le changement.

Ce rapport donnera une liste des optimisations faites par CompCert et décrira les optimisations rajoutées pendant ce stage. Comme le compilateur n'est pas du tout transparent sur les optimisations faites, il peut être utile de savoir ce qui est optimisé ou non. De plus, avoir une idée de ce qui est fait et aussi de ce qu'il reste à faire peut être utile pour faire avancer ce compilateur. Les optimisations seront surtout faites en x86 et certaines seront faites pour toutes les architectures (on le précisera à chaque fois)

On commencera par expliquer l'architecture de CompCert, quel sont les sous langages utilisés et leurs rôles (2). Ensuite nous verrons comment sont gérées les opérations arithmétiques et quelles modifications on a apportées (3). Enfin nous finirons par parler des opérations relatives au flot de contrôle dans une dernière partie (4).

2 Architecture de CompCert

CompCert passe par de nombreux langages intermédiaires. Dans cette partie, nous allons détailler les passes faites pour pouvoir localiser les optimisations faites. Nous allons aussi faire de courtes descriptions des langages pour comprendre les passes (Cette partie est résumée dans la figure 1 ci-dessous).

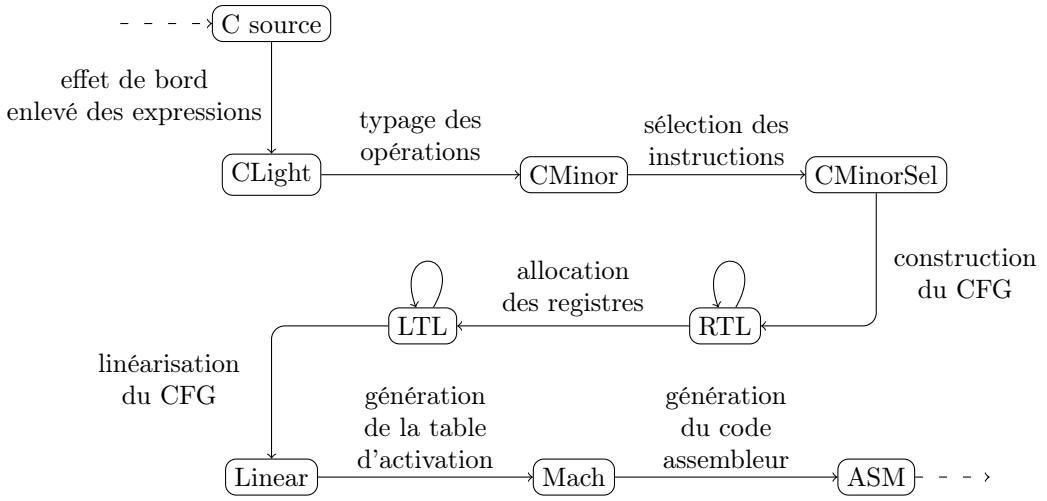

FIGURE 1 – Les passes et les langues intermédiaires de Compcert

CompCert a 3 grandes classes de langages intermédiaires. De CSource à CMinoreSel, ce sont des langages qui ont la même structure que le langage C (sous forme d'arbre). Ensuite nous avons RTL et LTL qui sont sous forme de graphe de flot de contrôle. Les opérations sont des codes à 3 adresses. C'est un code composé de 3 variables et une opération ($a := b$ op c). Enfin, les 3 derniers (Linear, Mach, ASM) sont des langages linéaires. Ils sont donc composés d'une liste d'instructions avec des labels et des instructions de branchements pour les sauts. Ce sont les langages les plus proches du langage machine.

Tout d'abord, le code source C est transformé en un premier langage appelé C source qui est le premier langage intermédiaire. Les expressions de C source sont presque toutes identiques aux instructions de C.

Ensuite, pour passer de C source à CLight, CompCert va enlever les effets de bords des expressions. CLight est donc un langage "pure", donc les affectations, les déclarations de variables et les appels de fonctions ne sont plus dans les expressions.

Le code CLight est transformé en code CMinor en passant par C#Minor. De CLight à C#Minor, CompCert va simplifier les boucles (structure plus simple à manipuler). Il va aussi typer les opérations. On aura donc la même opération pour des types différents. Par exemple pour l'addition on a `Oadd`, `Oaddf`, `Oaddsf`, `Oaddl`. Ensuite de C#Minore à CMinor CompCert va allouer la pile pour les variables locales. On a donc un pointeur pour accéder à la pile (`Oaddrstack`). La traduction de CLight à C#Minor se trouve dans le dossier cfrontend, ensuite tout le reste se trouve dans le dossier backend. C'est le dernier langage à ne pas dépendre de l'architecture.

CMinor est ensuite traduit en CMinorSel. C'est un langage qui ressemble beaucoup à CMinor, il contient des instructions et un mode d'adressage spécifiques à l'architecture. Cette traduction va optimiser la structure conditionnelle `if/then/else` en utilisant si possible des déplacements conditionnels (`cmov` en x86). La transformation se situe en partie dans le dossier backend mais aussi dans les dossiers des architectures.

CMinorSel est traduit en RTL. Pour cela on transforme l'arbre de CMinor en un graphe de flot de contrôle. Les opérations sont sous la forme de code à 3 adresses. Il est composé d'une infinité de pseudos registres pour préparer l'allocation de registres. Il y a beaucoup de passe de RTL vers RTL. Elles sont faites pour faire des optimisations comme la propagation de constante ($2 + 1 \to 3$), la suppression des variables globales statiques non utilisées, ...

RTL est alors traduit en LTL. Les pseudo registres sont traduits en registres spécifiques à l'architecture. S'il manque des registres, LTL utilisera la pile. La coloration de registre est faite en Ocaml et vérifiée après.

LTL est linéarisé donc transformé en Linear. Le graphe de flot de contrôle est transformé en une liste

d'instruction linéaire pouvant s'apparenter à de l'assembleur. On génère ensuite la table d'activation pour les appels des fonctions. Pour cela Linear est transformé en Mach. Et enfin à partir de Mach le code assembleur est généré puis assemblé en code machine.

3 Arithmétique

Les opérations arithmétiques, telles que la multiplication et la division sont largement utilisées et peuvent être optimisé pour les rendre plus rapides. Dans cette partie nous allons examiner comment CompCert génère ces opérations et expliquer les modifications apportées pour les optimiser. Toutes ces modifications sont réalisées pour l'architecture x86.

3.1 Multiplication par une constante

Dans la plupart des architectures, faire une multiplication peut être coûteux. CompCert réalise déjà quelques optimisations pour ça. Si la constante est égale à 1 Compcert renverra seulement la variable. Si la constante est égale à 0 il mettra le registre à 0. Si la constante c est une puissance de 2 il fera un décalage à gauche de $\log_2(c)$.

Il rajoute une optimisation s'il y a deux bits à 1 en position i et j. Pour faire cette multiplication CompCert additionnera la variable décalé de i à gauche et la variable de j à gauche. Par exemple $x * 10$ sera transformé en $(x << 3) + (x << 1)$.

En x86 pour faire ces additions et ces décalages CompCert va utiliser le mode d'adressage Intel avec l'instruction *lea*. Ce mode d'adressage est composé de 2 registres et 2 constantes. Il est noté `a(r1, r2, d)` et il réalisera le calcul $a + r1 + r2 * d$ avec $d \in \{1, 2, 4, 8\}$

GCC et CompCert ont deux façons de faire différentes. Par exemple pour la multiplication par 10 on a :

GCC	CompCert
`leaq (x + x * 4), y`	`leaq (x * 8), y`
`addq y, y`	`leaq (y + x * 2), y`

GCC va faire un *lea* en moins ce qui est légèrement mieux car, l'instruction *add* et plus courte et plus rapide. L'instruction *add* à une latence de 1 et *lea* a une latence de 3 si elle contient 3 éléments dans son adresse [Fog22, §Intel Skylake]. On peut aussi trouver des cas avec des constantes qui ont plus de deux bits gérés par GCC 1. Par exemple pour la constante 7 GCC fait $x * 7 = x * 8 - x$.

3.2 Division par une constante

CompCert réalise déjà de très bonnes optimisations pour la division par une constante.

Si une variable est divisé par 1 alors CompCert ne fera pas de division.

Si la variable est une puissance de 2 on note la constante c. Si la variable est non signée, il suffit de faire un décalage à droite de $\log_2(c)$. Cependant, si la variable est signée on peut avoir un problème d'arrondi. Pour régler ce problème, il faut d'abord tester si la variable est négative, si c'est le cas il faut lui ajouter $c - 1$ puis faire le décalage. Sinon, il suffit juste de faire le décalage. Le test et l'addition se font à l'aide de l'instruction *cmovl* qui va faire un déplacement de registre si la comparaison des deux nombres est négative. On a donc comparé la variable avec 0 pour déterminer son signe.

Sinon CompCert va transformer la division en multiplication. Pour faire la division $\lfloor n/d \rfloor$ avec d une constante, nous cherchons les deux constantes m et n tel que $\lfloor n/d \rfloor = \lfloor n \times m/2^{32+l} \rfloor$ la recherche est détaillé dans [GM94] ou dans [War13, §10]. Il suffira donc de multiplier n par m de prendre les 32 bits de points fort et de faire un décalage de l. Pour prendre les 32 bits de points fort de $m \times n$ CompCert à une opération *Omulhu* pour les non signés et *mulhs* pour les signés qui va générer l'instruction x86 *mul*. Comme pour les puissances de deux, pour les entiers signés il faut faire attention à l'arrondi. CompCert générer un code qui va tester si la variable est négative, si c'est le cas il rajoute 1 au registre s'il est négatif, pour que l'arrondi se fasse vers 0, sinon il ne fait rien. Pour faire cela CompCert va additionner le bit de signe du résultat avec le résultat. Pour récupérer le bit de signe il suffit juste de faire un décalage logique de 31.

Grâce à cette optimisation CompCert a des performances assez proches de GCC.

	GCC	CompCert
cycles	1.74	3.77
instructions	12	14

TABLE 1 – Performance du calcul $x/30 + x/10 + x/3$ de GCC et CompCert

3.3 Instruction de multiplication et bit de point fort

Pour récupérer les bits de points fort de la multiplication *Omulhu* va générer l'instruction *mul* qui est une instruction de multiplication entre le registre en paramètre et un registre prédéfini et qui va faire le calcul sur deux registres de retours, eux aussi prédéfinis. On a donc cherché à la remplacer par *imul* qui prend en paramètre deux registres pour les multiplier et mettre le résultat dans le premier registre. Cette instruction a une latence de 1 comparer à 3 pour l'autre instruction [Fog22, §Intel Skylake]. Il y aura donc moins de pression sur les registres ce qui peut améliorer les performances du programme si plusieurs multiplications sont faite à la suite. Ensuite on fait un décalage de 32 sur le résultat pour récupérer les bits de point fort.

Pour faire cela on a modifié les fonctions *mulhu* et *mulhs* dans *x86/selectOP.vp*. La modification se trouve donc entre CMinor et CMinorSel. On a donc rajouté un cas si l'architecture x86 gère les entiers 64bits. Si c'est le cas on ne génère plus l'instruction *Omulhu* mais le décalage de 32 du résultat de *Omull* avec comme paramètres les registres 64bits contenant les deux expressions à multiplier. Il faut prendre les bits de points faibles de cette expression pour éviter les débordements.

Pour prouver la modification réalisée dans *mulhu*, on a dû changer la preuve de *eval_mulhu* dans *x86/SelectOpproof.v*. L'énoncé de la preuve est : *binary_constructor_sound mulhu Val.mulhu*. En dépliant la définition on a cet énoncé :

$$\begin{aligned}\forall\ x\ v_x\ y\ v_y,\ &\text{eval\_expr}\ \ldots\ x\ v_x\ \rightarrow\\ &\text{eval\_expr}\ \ldots\ y\ v_y\ \rightarrow\\ &\exists v, \text{eval\_expr}\ \ldots\ (\text{mulhu}\ x\ y)\ v \wedge \text{Val.lessdef}\ (\text{Val.mulhu}\ v_x\ v_y)\ v\end{aligned}$$

Val.mulhu représente la sémantique de *mulhu*. Comme c'est un opérateur binaire on a deux arguments (x et y). On commence par supposer que v_x est la valeur de x et v_y la valeur de y. Ensuite on a défini v l'évaluation de *mulhu x y*. Cette valeur v est défini comme la traduction des opérations rajoutée vers leurs sémantiques définies dans le module *Val*. Il a fallu montrer que *mulhu x y* s'évalue bien en v, c'est une preuve assez facile à faire. Ensuite on veut montrer que *Val.mulhu* v_x v_y est moins défini que v (*Val.lessdef*). Dans notre cas comme *Val.mulhu* est défini tout le temps on veut montrer que les deux valeurs sont égales :

```
Val.loword
    (Val.shrlu (Val.mull (Val.longofintu x) (Val.longofintu y))
      (Vint (Int.repr 32))))
    = Some (Val.mulhu x y)
```

Comme *Val.mulhu x y* est défini comme la division de $x \times y$ par 2^{32} on a dû montrer un lemme intermédiaire qui prouve l'égalité entre $x >>_{64} y$ et $x/2^y$. En appliquant ce lemme la preuve est presque fini, il reste à prouver des transformations entre *Int64* et *Int* à l'aide de lemme rajouté dans le module *Int*. Une preuve similaire a été réalisé pour prouver la modification sur *mulhs*.

	CompCert	CompCert modifié
cycles	3.77	1.59
instructions	14	10

TABLE 2 – Performance du calcul $x/30 + x/10 + x/3$ de CompCert et CompCert modifié

Grâce à cette modification CompCert va utiliser une instruction avec 3 fois moins de latence pour faire la multiplication (on observe bien ce facteur 3 dans le tableau 2).

3.4 Constante de 64bits pour la multiplication

Après l'optimisation de la multiplication, la constante généré peut parfois dépasser 32 bit, or l'instruction *imul* ne support pas les constantes de 64 bits. Compcert va donc mettre cette constante dans la mémoire ce qui peut avoir de mauvaise conséquence sur les performance du programme. On veut donc forcer CompCert à utiliser un registre et pour cela on utilise le constructeur *Omull* de CMinor pour le forcer à multiplier 2 registres et enfin pour éviter la propagation de constante dans RTL on va utiliser encore une fois le même constructeur si la constante est trop grande. Ces deux optimisations ont été prouvées sans problèmes car elles font très peut de transformations (changement de constructeur avec la même sémantique).

3.5 Instruction de déplacement de registre

Les processeur ont pour la plus part des instructions qui permettent de déplacer le contenu des registres dans d'autres zones mémoires (registres, adresse). On veut aussi pouvoir stocker une constante dans un registre. Pour cella on peut utiliser l'instruction *mov* qui possède beaucoup de variante selon la taille d'arrivée ou de destination. CompCert ne fait pas toujours les choix les plus optimisant. Par exemple pour mettre une constante dans un registre on a l'instruction *Pmovq_ri* dans le langage ASM, CompCert va forcément traduire cette instructions par un déplacement de 64bits (*movq*, *movabsq*). On a donc rajouter une condition qui va tester si la constante tient dans 32 bits et si s'est la cas on utilisera un déplacement de 32 bits (*movl*).

	CompCert	CompCert modifié
cycles	1.11	0.44
instructions	6	6

TABLE 3 – Performance de r_i = `0xFFFFFFFF` avec i de 0 à 2, de CompCert et CompCert modifié

Dans l'exemple CompCert utilise *movabsq* pour mettre `0xFFFFFFFF` dans r_i grâce à l'optimisation il va utiliser *movl*. Cette instruction est plus courtes ce qui facilite le décodage des instructions.

3.6 Instruction *lea, add* et *sub*

Pour faire des addition et des soustraction CompCert utilisera systématiquement l'instruction *lea*. Au lieux d'utiliser cela CompCert pourrait utiliser les instructions *add* et *sub* qui sont des instructions plus courtes (1 octets en moins). Le code assembleur est aussi plus lisible donc plus facile à déboguer. Cette optimisation n'a pas encore été prouvé.

3.7 Comparaison avec XOR

Quand une condition contient un xor qui rend un booléen (1 ou 0) CompCert va faire le xor puis comparer la valeur avec 0 ou 1. Cependant, on peut simplifier cette condition, car si un xor est vrai (resp. faux) il suffit de regarder si les deux registres sont égaux (resp. différents). On peut appliquer ce raisonnement à une suite de xor. On peut exécuter tous les xor sauf le dernier et tester si le résultat de tous les xor est égal au dernier registre.

Pour faire cette optimisation on va se placer encore une fois entre CMinor et CMinorSel car c'est ici que CompCert va faire des optimisations sur les `if`. On modifiera la fonction *condexpr_of_expr*, car c'est la fonction qui transforme les expressions en condition.

On a juste rajouté un cas qui teste si l'opération est un xor et qu'il est bien comparé à un zéro, car sinon l'optimisation est fausse. Si l'expression respecte les conditions on va renvoyer la comparaison entre la partie gauche et la partie droite. Cette transformation n'a pas été prouvé

	GCC	CompCert	CompCert modifié
cycles	163.92	167.21	161.72
instructions	470	505	495

TABLE 4 – Performance des conditions composées de xor

Pour tester cette modification on a dû faire en sorte que chaque branche fasse un appel de fonction pour éviter que la condition soit transformée en condition ternaire. Nous avons alterné les branches pour limiter les prédictions de branchements.

On peut observer de légères différences de performance : a un peu plus rapide après modification et moins d'instructions utilisées.

3.8 Instructions *cmp* et *test*

Quand on fait une comparaison par 0 en x86 CompCert va utiliser l'instruction `cmp 0, r`. À la place on pourrait utiliser `test` qui prend un octet de moins que `cmp`. Cette instruction va faire un *et* bit à bit et va mettre à jour les registres d'états `SF, ZF, PF`. La mise à jour de `ZF` est intéressante, car s'il est mis à vrai quand on exécute `test r, r` ça veut dire que le registre r est égal à 0. On a donc rajouté une condition, quand la constante est égale à 0 on générera `test` au lieu de `cmp` dans *Asmgen.v* (Mach → Asm). Surprenamment cette condition était déjà présente quand la condition est une comparaison signée. Cette modification a été prouvée.

Cette optimisation va réduire la taille du code machine, ce qui peut avoir une légère impacte sur les performances.

4 Flot de contrôle

Le flot de contrôle d'un programme va représenter les sauts et les appels de fonction fait par le programme. Les sauts peuvent venir des boucles et des conditions.

4.1 Condition

Les conditions sont très importante dans les programmes. Elles permettent de prendre en compte des calculs fait précédemment pour modifier les comportement du programme.

4.1.1 Conditions ternaires

Le langage C contient des conditions ternaires (`(c)?e1:e2`) utiles pour écrire des conditions simples. CompCert va utiliser des déplacements conditionnels pour éviter de faire des branchements inutiles et coûteux pour juste affecter une valeur à une variable. Mais utiliser ce déplacement conditionnel nous force à évaluer les deux expressions. Donc si une des expressions a des effets de bord, évaluer les expressions va changer la sémantique du programme. Donc CompCert va d'abord vérifier que les expressions sont sûrs et aussi qu'elles ne sont pas trop coûteuses. Si l'évaluation des deux expressions est plus longue qu'un branchement CompCert fera le branchement. Cette vérification se fait entre CMinor et CMinorSel dans le fichier *Selection.v* dans la fonction *if_ conversion*.

Nous pouvons remarquer qu'on peut écrire cette condition ternaire normalement avec un bloc if et else et quand même faire l'optimisation. CompCert va faire cette optimisation pour les affectations de variables si toutes les conditions sont respectées.

— Pas d'effets de bords
— Même variable affectée
— Pas une variable globale

CompCert ne fait pas l'optimisation quand la destination n'est pas une variable local.

Si la destination est une variable globale on pourrait utiliser un registre de transition, car un déplacement conditionnel ne peut pas ce faire vers une adresse. Ensuite CompCert peut écrire le contenu du registre à l'adresse de la variable globale.

Enfin si la dernière instruction est un `return`. On peut faire un déplacement conditionnel vers le registre de retour.

4.1.2 Conditions avec *return*

CompCert génère déjà des déplacements conditionnels quand il y a la même affectation dans les deux blocs de la condition. Cependant, on a aussi vu qu'il n'en génère pas quand les deux blocs finissent par un

`return`. Pourtant, un `return` se fait en deux temps, l'affectation du registre de retour et le retour. On peut donc voir un `return` comme une affectation. De plus comme une optimisation similaire est déjà faite pour les affectations ça n'a pas été trop difficile de rajouter cette optimisation.

Pour faire cette optimisation nous allons nous situer entre CMinor et CMinorSel et plus précisément dans la fonction $if\_conversion$. Comme cette fonction fait une disjonction de cas sur les types de déclarations (`SCskip, SCassign, SCother`) on a rajouté un type qui s'appelle `SCreturn` pour déterminer si une branche finit par un `return` ou non. Ensuite nous avons rajouté le cas où les deux branches finissent par un `return`. Ensuite nous avons construit une fonction semblable à $if\_conversion\_base$ qui s'appelle $if\_conversion\_return$. Cette fonction va tester si les deux blocs sont sûrs (pas d'effets de bord) et va utiliser l'heuristique appelé $if\_conversion\_heursistic$ faite en Ocaml pour éviter de faire l'optimisation si jamais les deux blocs font trop de calculs. Si toutes les conditions sont réunies on va utiliser la fonction sel_select_opt pour transformer la structure en condition ternaire. Pour utiliser cette fonction on doit connaitre le type de retour, car le déplacement conditionnel n'existe pas toujours selon les types et l'architecture (On va donc utiliser une fonction dans *SelectOP.vp* qui est un fichier spécifique à l'architecture). On a donc dû faire remonter le type de retour depuis la fonction d'entrée des transformations de fonction ($sel\_function$).

Grâce à cette modification CompCert va générer une condition ternaire si on a un `return` dans le bloc du `if` et du `else`. Malheureusement cette modification n'optimisera pas quand le deuxième `return` n'est pas dans un `else`.

```
if (x == 1){
  return 0;
} else {
  return 1;
}
```

(a) Code optimisé

```
if (x == 1){
  return 0;
}
return 1;
```

(b) Code non optimisé

Pour optimiser le code (b) il faudrait avoir dans la branche $ifso$ le bloc du if et dans la branche $ifnot$ ce qu'il y a après la condition. Il faudrait aussi que les deux branches finissent par un `return` et qu'elles respectent les conditions d'optimisations.

	GCC	CompCert	CompCert modifié
cycles	100.95	122.99	111.56
instructions	270	305	310

TABLE 6 – Performance des if-then-else composés de `return`

Ce genre de code est plus difficile à tester, car le processeur va faire des prédictions de branchement. Il va donc choisir une branche et l'exécuter en avance. Si le choix était correct il va continuer et il aura gagné du temps et si le choix n'était pas correct il reviendra en arrière.

À cause de cette prédiction, il faut tester un code qui passe par les deux branches pour éviter que le processeur choisisse toujours la bonne branche. On a donc fait le test de performance sur le code (a) et on a fait en sorte qu'il alterne entre la branche vraie et la branche fausse. Dans ce cas-là, la modification rend CompCert plus performant, car il ne va pas faire de saut. Il va exécuter plus d'instruction, car il va calculer ce qu'il y a dans les deux blocs.

Dans le cas où on exécute toujours le même bloc CompCert non modifié va fortement se rapprocher de CompCert modifié, car le processeur va s'adapter et va toujours prédire la bonne branche.

Cette modification n'a pas été prouvé.

4.1.3 Conditions paresseuses

Pour l'instruction de condition, CompCert va d'abord transformer la condition pour faire en sorte que le && et le || soit paresseux. Pour ça il va rajouter des conditions qui vont permettre de donner une variable vraie ou fausse en évaluant le moins d'expressions possibles. Cette méthode est utilisée pour faciliter la preuve donc il est difficile de faire des optimisations dans cette partie. GCC et Clang eux vont juste tester les conditions une par une et faire un saut selon la valeur et le connecteur logique.

Par exemple :

```
if (a || b){
  return 10;
} else {
  return 20;
}
```

```
var '$62';
if ('a' !=u 0) {
  '$62' = 1;
} else {
  '$62' = 'b' !=u 0;
}
if ('$62' !=u 0) {
  return 10;
} else {
  return 20;
}
```

Cette façon de gérer les opérateurs paresseux est assez couteuse, car il y a beaucoup de calcul. On pourrait à la place tester a, sauter si vrai, ou sinon tester b et sauter si vrai. Comme les conditions vont être optimisées en condition ternaire entre CMinor et CMinorSel, CompCert n'arrivera pas à faire des optimisations de branchement dans les optimisations RTL.

Dans l'exemple du dessus on peut remarquer que CompCert fait des sauts dans des blocs pour seulement faire un `movl`. À la place on pourrait utiliser le `cmove`. CompCert utilise déjà le déplacement conditionnel pour des affectations de variables. Faire cette transformation oblige le calcul des deux branches. Donc il ne faut pas qu'elles aient d'effets de bord.

4.2 Branchement

4.2.1 Boucle

Dans cette partie nous verrons la génération de code qui contient des boucles. Toutes les boucles vont être transformées en boucle infinie à partir de CLight. La condition est remplacée par un if et un *break* au début ou à la fin selon le type de boucle (`while`/`do-while`). C'est une très bonne représentation, car en général une boucle en assembleur est transformée aussi en boucle infinie avec une comparaison et un branchement conditionnel (si la condition est fausse). Donc comme CompCert génère un if, beaucoup d'optimisations de boucle passe par des optimisations de conditions (détaillé ci-dessus).

Les boucles for sont énormément optimisées par les compilateurs modernes. Malheureusement CompCert va faire très peu d'optimisation sur celle-ci. Par exemple, elles ne seront pas déroulées pour optimiser la prédiction de branchement [KA01]. Aussi deux boucles qui itèrent sur le même indice ne seront pas fusionnées, c'est une optimisation très difficile à réalisé. Il y a plein de paramètre à prendre en compte comme l'indépendance des boucles (la première boucle ne changera pas le comportement de la deuxième).

4.2.2 Switch-Case

Le switch est une structure conditionnelle faite pour comparer plusieurs fois une même expression avec des constantes. Si le compilateur compile bêtement un switch on risque de faire beaucoup de comparaisons inutiles et gâcher le potentiel d'optimisation d'une telle structure. Au lieu de dérouler un switch en une cascade de condition qui va comparer les expressions une à une, nous pouvons profiter des constantes et transformer un switch en un arbre binaire de recherche ou en une table de hachage.

L'arbre binaire permettrait de trouver le bon bloc en $\mathcal{O}(\log(n))$, et la table de Hachage en temps constant (très compliqué à mettre en place).

CompCert va transformer le switch en arbre binaire de recherche dans la transformation de CMinor à CMinorSel dans le fichier *selection.v*.

4.2.3 Fonctions

Appel de fonction Les appels de fonctions ne se résument pas à l'utilisation de l'instruction `call`. Il faut aussi passer les arguments à l'appelé. Gérer la pile utilisée par le corps de la fonction.

Les arguments sont passés dans des registres :

— entier : `rdi, rsi, rdx, rcx, r8, r9`

— flottant : `xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7`

les autres arguments sont placés sur la pile (le dernier est tout en haut de la pile).

Avec CompCert la table d'activation contient en premier une adresse vers le début de la table d'activation de la fonction appelante.

Ensuite, il va enregistrer les registres qui sont utilisés parmi `rbp, rbx, r12, r13, r14, r15` (registres sauvegardés par l'appelé).

Enfin la dernière partie allouée va servir pour la fonction si jamais l'allocation de registres n'a pas trouvé assez de registres, ou si la fonction appelle une fonction qui a besoin d'avoir des arguments sur la pile.

CompCert crée cette table d'activation en agrandissant d'abord la pile avec une soustraction sur le pointeur de pile. Il s'assure que la pile soit bien aligné sur 16 bits pour des raisons de performances d'accès à la mémoire [WEW05]. Ensuite, il utilise `mov` pour enregistrer les registres dont il a besoin. Pour désallouer la table d'activation CompCert va juste ajouter ce qu'il a soustrait au pointeur de pile. Il va ensuite appeler `ret` pour quitter la fonction.

GCC et Clang ne vont pas utiliser cette méthode à la place, ils vont juste utiliser l'instruction `push` pour rajouter des éléments sur la pile petit à petit. Pour désallouer ils vont utiliser la même méthode que CompCert (addition avec le registre de pile).

Comme dit précédemment CompCert va enregistrer le pointeur de la table d'activation de l'appelant, ce qui n'a pas grand intérêt, car ces deux tables sont l'une au-dessus de l'autre sur la pile. Nous avons supposé que ce pointeur est utilisé dans les preuves, car CompCert voit les tables d'activation comme une liste chainée. Ce pointeur serait donc l'adresse du prochain bloc.

De plus, quand la fonction ne contient pas d'autre appel de fonction, CompCert va quand même enregistrer le pointeur de la table d'activation de l'appelant, alors qu'on pourrait juste exécuter le code de la fonction sans aligner la pile, ce qui est fait par GCC.

Fonction récursive Une fonction récursive consomme beaucoup de mémoire, car à chaque nouvel appel on recrée une table d'activation, ce qui est coûteux en temps et en mémoire. On peut optimiser cela que si la fonction est récursive terminale, c'est-à-dire que tous les appels de la fonction se font dans un `return`. Si on respecte cette condition on peut optimiser cette fonction récursive en boucle [Cli98]. CompCert va faire cette optimisation à partir de RTL dans le fichier *tailcal.v* dans le dossier backend. Cette optimisation est très importante, mais elle contient un défaut. À chaque itération de boucle CompCert va charger la table d'activation puis la décharger. Nous n'avons pas encore réussi à trouver de raison à ce comportement.

5 Outils réalisés

Pour faire tous les tests, nous avons vite créé des outils pour automatiser les tâches répétitives comme la compilation de programme et les tests de performances.

5.1 Compilation de Programmes

Pour comparer le code assembleur généré par GCC CompCert et les nouvelles versions de CompCert nous avons développé un petit outil qui va prendre des compilateurs en arguments et un fichier à compiler. Il a aussi une petite interface qui va générer un code demandé dans un fichier C. L'outil va compiler le code donné avec les différents compilateurs puis avec un lexer il va enlever les instructions de debug de l'assembleur x86. Après avoir lancé le programme, il ne reste plus qu'à ouvrir les deux fichiers générés côte à côte et comparer les codes assembleurs.

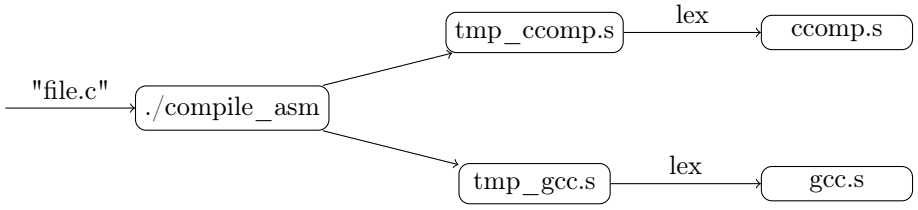

Figure 2 – Fonctionnement de compile_asm

Pendant le TER nous avons pu rajouter CompCert à Compiler Explorer grâce à l'autorisation de Xavier Leroy. C'est un outil très utile pour comparer les codes généré par différents compilateurs. On peut aussi choisir une architecture spécifique ce qui permet de faire des optimisations sur de nouvelles architectures.

5.2 Performance

L'outil qui mesure les performances est composé de deux parties.

La première partie va compiler les programmes (avec les compilateurs passés en arguments) les exécuter puis récupérer les performances et les mettre dans un tableau. Le programme est composé de plusieurs choses. On a une fonction à tester et une fonction vide avec la même interface que la fonction précédente. Ensuite, il y a deux fonctions qui vont faire une boucle sur les deux tests. Ensuite, dans le point d'entrée on va mesurer les performances des deux boucles puis on va les soustraire et récupéré le temps moyen d'exécution. On est passé par une fonction vide pour éviter de mesurer le temps d'appel d'une fonction et les calculs fait pour les paramètres. Toutes les fonctions qui ne sont pas en lien avec le code à tester sont compilées en `-O0` pour éviter toutes optimisations qui pourrait influencer les mesure.

Au début pour mesurer les performances, nous avons utilisé $perf$. Perf mesurait aussi le temps de chargement des programmes donc on a décidé de laisser le programme mesurer ses performances tout seul avec des appels systèmes. Le programme va donc afficher les performances de la boucle de test sur la sortie d'erreur. Le programme principal va pouvoir récupérer les données et les ranger dans un tableau.

On a dû rajouter des arguments spécifiques à GCC car quand on utilise une fonction externe GCC va utiliser la table PLT pour le lieur dynamique ce qui ralentit certains codes générés par GCC. On a donc rajouté les arguments `-fno-plt -fno-pic -fno-pie` à GCC. Cependant, même avec ces arguments, les appels de fonction ne seront pas tout à fait identiques avec GCC et CompCert mais les performances seront bien plus rapprochées.

La deuxième partie contient la fonction à tester et la mesure de performances. Tout d'abord pour mesurer les performances on a utilisé l'appel système $perf\_event\_open$ [1]. On a reproduit l'exemple en utilisant les arguments pour mesurer le nombre de cycles et le nombre d'instructions. Après avoir récupéré les données on les affiche sur la sortie d'erreur.

Ensuite comme GCC peut faire des optimisations en plus qui peuvent casser les mesure. Pour éviter ça on utilise une fonction externe vide qui peut être appelé dans des endroits clés pour bloquer l'optimisation ciblée.

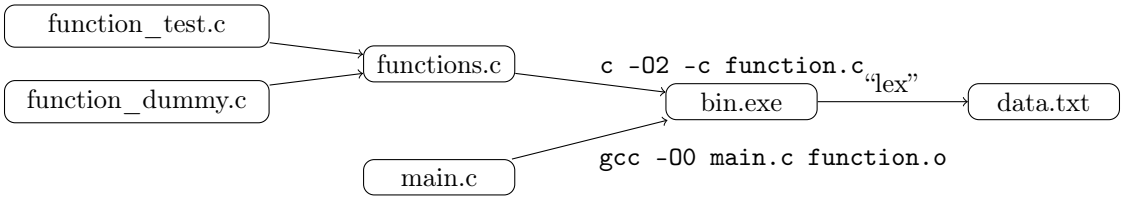

FIGURE 3 – Fonctionnement de auto_perf

6 Conclusion

Pour conclure, pendant cette première partie de TER j'ai pu explorer CompCert, mieux comprendre son fonctionnement et même faire quelques optimisations. Les optimisations ayant le plus de répercussions sont : le changement d'instruction pour la division par une constante et l'optimisation des conditions ternaires pour les `return`. Ce sont des bouts de code fréquemment utilisés.

Il y a encore beaucoup de chemin pour arriver à avoir un code machine aussi performant que ceux générés par un compilateur moderne. Par exemple optimiser les conditions paresseuses peut être très important, car tous les programmes en contiennent.

Il faudra aussi prouver toutes le reste des modifications avant de pouvoir les intégrer réellement à CompCert.

1. `https://man7.org/linux/man-pages/man2/perf_event_open.2.html`

Références

[BC13] Yves Bertot et Pierre Castéran. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[Bla07] Sandrine Blazy. « Comment gagner confiance en C ? » In : *Revue des Sciences et Technologies de l'Information-Série TSI : Technique et Science Informatiques* 26.9 (2007), p. 1195-1200.

[Cli98] William D Clinger. « Proper tail recursion and space efficiency ». In : *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, p. 174-185.

[Fog22] Agner Fog. *Instruction tables : Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Rapp. tech. 2022. URL : `https://www.agner.org/optimize/instruction_tables.pdf`.

[FP13] Jean-Christophe Filliâtre et Andrei Paskevich. « Why3—where programs meet provers ». In : *Programming Languages and Systems : 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, p. 125-128.

[GM94] Torbjörn Granlund et Peter L Montgomery. « Division by invariant integers using multiplication ». In : *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 1994, p. 61-72.

[KA01] Ken Kennedy et John R Allen. *Optimizing compilers for modern architectures : a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[Ler09] Xavier Leroy. « Formal verification of a realistic compiler ». In : *Communications of the ACM* 52.7 (2009), p. 107-115.

[War13] Henry S Warren. *Hacker's delight*. Pearson Education, 2013. Chap. 10.

[WEW05] Peng Wu, Alexandre E Eichenberger et Amy Wang. « Efficient SIMD code generation for runtime alignment and length conversion ». In : *International Symposium on Code Generation and Optimization*. IEEE. 2005, p. 153-164.