Auteur externe

Zeste de savoir

Les optimisations des compilateurs

21 janvier 2019

Table des matières

1. Introduction

Comme vous le savez sûrement, les compilateurs effectuent des optimisations sur le code des programmes compilés. Ces optimisations peuvent impliquer le code source lui-même, ou l'assembleur généré par le compilateur. Beaucoup pensent que les compilateurs sont de vrais magiciens, et qu'il est inutile d'optimiser soi-même un code source. Mais on peut quand même se demander ce que peut faire le compilateur sur notre code source. Et c'est le but de cet article que de vous l'expliquer.

i

Une bonne connaissance de l'architecture des ordinateurs est requise pour la compréhension de ce cours. Pour comprendre l'essentiel du cours, une connaissance des concepts de base suffit : savoir ce qu'est une instruction, avoir quelques rudiments en assembleur ou sur le langage machine, connaitre la hiérarchie mémoire, etc. Quelques remarques isolées demandent des connaissances sur la micro-architecture des processeurs modernes : pipeline, exécution dans le désordre. De plus, les exemples de code seront en C/C++.

2. Optimisations de la vitesse d'exécution

Un compilateur peut optimiser plusieurs choses :

— la taille du programme ;
— la vitesse d'exécution ;

— la consommation énergétique du programme.

Si les compilateurs sont très bons pour optimiser la vitesse d'exécution, il n'en est pas vraiment de même pour la taille du code ou la consommation énergétique. Quoiqu'il en soit, il existe plusieurs façons pour optimiser la vitesse d'exécution :

— utiliser au mieux les instructions du processeur ;
— diminuer le temps processeur passé à faire des calculs ;
— virer des branchements et fonctions pour obtenir un code le plus linéaire possible ;
— utiliser au mieux la hiérarchie mémoire, que ce soit au niveau des registres, du cache ou de la DRAM.

2.1. Instructions

Les premières méthodes tentent d'utiliser au mieux les instructions fournies par le processeur.

2.1.1. Ordonnancement des instructions

Tout d'abord, le compilateur est capable de changer l'ordre des opérations pour profiter au mieux du fonctionnement en pipeline[1] des processeurs modernes. Et à ce petit jeu, aucun programmeur ne peut rivaliser.

2.1.2. Choix des instructions

La seconde méthode consiste à utiliser les instructions machine les plus adaptées, histoire d'obtenir quelques simplifications. Par exemple, il est possible de remplacer certaines instructions par d'autres plus rapides. L'exemple le plus connu est celui des multiplications/divisions par une puissance de deux constante qui se transforme facilement en décalage.

Une autre méthode consiste à fusionner des suites d'instructions élémentaires en une seule instruction. Par exemple, prenons le cas d'un processeur qui possède l'instruction MAD qui permet d'effectuer le calcul `A + B * C` en une seule instruction. Dans un code source écrit dans un langage de haut-niveau, on ne peut pas utiliser celle-ci directement, et c'est au compilateur de fusionner une addition avec une multiplication en une seul instruction MAD.

Il est strictement inutile de faire ce genre de chose à la main, à une exception près : les processeurs de traitement de signal. Sur ces processeurs, coder en assembleur permet d'utiliser les modes d'adressage *modulo* et *bit-reverse* que le compilateur ne peut pas utiliser facilement.

1. Une architecture de processeur en pipeline permet de traiter simultanément plusieurs instructions.

2.1.3. Vectorisation

Le compilateur peut aussi tenter d'utiliser des instructions vectorielles quand c'est possible. Les techniques qui permettent au compilateur de découvrir des suites d'instructions qui peuvent profiter des instructions SIMD[2] des processeurs modernes sont ce qu'on appelle de la **vectorisation**.

Et à ce petit jeu, un bon programmeur donne un résultat nettement meilleur qu'un compilateur : ces derniers ont beaucoup de mal à optimiser les codes vectoriels. Bien sûr, de telles optimisations ne sont pas évidentes, mais elles peuvent permettre d'obtenir des codes de 3 à 4 fois plus rapides !

2.2. Calculs

L'optimisation des calculs se base sur un principe simple : certains calculs peuvent être simplifiés, voire totalement éliminés.

2.2.1. Simplifications algébriques

Un compilateur peut, sous certaines conditions, effectuer des simplifications algébriques. De nombreuses optimisations de ce genre sont possibles pour les entiers : `a * b + a * c` qui se transforme en `a * ( b + c )`. Cependant, ces optimisations ne sont pas toujours possibles : par exemple, calculer - (a - b) avec des entiers ne peut pas être simplifié en - a + b à cause d'une possibilité d'*overflow* assez bien cachée. Quelque chose de similaire a lieu pour les flottants : les opérations flottantes ne sont pas associatives, ni distributives. Mais certains compilateurs ne se gênent pas pour faire les simplifications, à partir d'un certain niveau d'optimisation.

On trouve aussi la même chose pour les booléens, sous certaines conditions. Par exemple, l'expression `if(!a && !b)` peut être simplifiée en `if ( ! (a || b) )`. Mais ces optimisations ne sont pas toujours possibles, modifier l'ordre des comparaisons ou en supprimer peut ne pas donner le même résultat. Dans un tel cas, sachez que simplifier ces expressions à la main permet d'améliorer la lisibilité de votre programme (même si c'est souvent une amélioration marginale).

2.2.2. Élimination des calculs constants

Certains calculs dans un programme sont totalement constants. Par exemple, des calculs du style : `int a = 2 * 3.1415926535` sont simplifiables. De même, un compilateur peut réduire de nombreux calculs du type `a + 0`, `a * 0`, `a * 1`, etc. Dès qu'une expression a toutes ses opérandes constantes, un compilateur peut parfois calculer le résultat lui-même et remplacer l'expression par son résultat : on parle de **constant folding**.

Mais attention, cela n'est valable que dans un cas bien précis : celui des expressions et fonctions qui donnent le même résultat pour des opérandes identiques. Il existe des calculs qui ne respectent pas cette règle, notamment certaines fonctions : ce sont des expressions ou fonctions

2. Cela signifie qu'une même instruction est appliquée simultanément sur plusieurs données afin de produire plusieurs résultats.

dites **impures**. Et un compilateur n'a pas vraiment de moyens pour savoir si une fonction est pure ou impure : il devrait regarder et analyser son code, ce qui demanderait trop de ressources pour un résultat qui n'en vaut pas le coup. Dans ces conditions, le compilateur considère toutes les fonctions comme étant impures. La conséquence, c'est que de très nombreuses optimisations seront rendues impossibles si jamais on trouve une fonction au mauvais endroit dans le code source.

2.2.3. Propagation des constantes

Les compilateurs sont aussi capables de propager ces constantes. Cela veut dire remplacer les noms de variables dont la valeur est constante par la constante elle-mêmes.

Par exemple, le code suivant :

```
1 const int a = 2;
2 const int b = 6;
3 const int c = 8;
4
5 int delta = b*b - 4*a*c;
```

peut être remplacé par :

```
1 const int a = 2;
2 const int b = 6;
3 const int c = 8;
4
5 int delta = 6 * 6 - 4 * 2 * 8;
```

On peut remarquer que cette optimisations se marie particulièrement bien avec le *constant folding*. Dans l'exemple du dessus, combiner les deux optimisations permet de calculer le résultat directement :

```
1 const int a = 2;
2 const int b = 6;
3 const int c = 8;
4
5 int delta = - 28;
```

2.2.4. Suppression des calculs redondants

Certains calculs sont inutiles ou redondants : il arrive souvent que des expressions calculent la même valeur que ce soit dans le code source ou dans la représentation intermédiaire d'un

programme. Dans ce cas, une valeur est calculée plusieurs fois à des endroits relativement proches, par des suites d'instructions séparées.

Par exemple, prenons ce cas de figure :

```
##include <limits>

float solve_secondDegre_equation(float a, float b, float c)
{
    if (b*b - 4*a*c > 0)
    {
        return ( -b + sqrt (b*b - 4*a*c) ) / (2 * a) ;
    }
    else if (b*b - 4*a*c == 0)
    {
        return -b / (2 * a) ;
    }
    else
    {
        return std::quiet_NaN();
    }
}
```

Comme vous le voyez, le delta `(b*b - 4*a*c)` est recalculé plusieurs fois. Un compilateur pourrait remplacer automatiquement ce code redondant par ceci :

```
##include <limits>

float solve_second_Degre_equation(float a, float b, float c)
{
    float temp = 2 * a ;
    float temp2 = b*b - 4*a*c ;

    if (temp2 > 0)
    {
        return ( -b + sqrt (temp2) ) / temp ;
    }
    else if (temp2 == 0)
    {
        return -b / temp ;
    }
    else
    {
        return std::quiet_NaN();
    }
}
```

Le compilateur peut stocker cette valeur dans une variable, variable qui est alors réutilisée en lieu et place du résultat des expressions redondantes. Expressions redondantes qui sont alors supprimées.

Cependant, le compilateur ne peut rien faire si la valeur est calculée par une fonction impure. Quand le compilateur voit que deux fonctions impures sont appelées avec des arguments identiques, il ne peut pas savoir si le résultat renvoyé sera le même et ne peut pas remplacer la seconde exécution de la fonction par une simple lecture de variable.

Il faut toutefois signaler qu'il y a des exceptions à cette exception. Par exemple, prenez le code suivant :

```
1 for (unsigned i = 0; i < strlen(string); ++i)
2 {
3     ...
4 }
```

Un bon compilateur C peut parfaitement optimiser de lui-même l'usage de la fonction strlen, et donner un code assembleur équivalent au code suivant :

```
1 unsigned length = strlen(string);
2
3 for (unsigned i = 0; i < length; ++i)
4 {
5     ...
6 }
```

Quoiqu'il en soit, éviter des expressions longues comme le bras est un plus clairement appréciable en terme de lisibilité, et factoriser des expressions dans des variables correctement nommées est alors d'une grande aide.

2.2.5. Réduction en force

Les boucles, et notamment celles qui parcourent un tableau sont aussi une source d'optimisations.

Prenez ce code source :

```
1 for (int i = 0; i < ArraySize; ++i)
2 {
3     Array[i] = i;
4 }
```

Cela ne se voit pas au premier abord, mais il y a une multiplication cachée dans cette boucle. Pour cela, il faut savoir que l'adresse de l'élément d'indice i est calculée en effectuant le calcul

suivant : `Adresse de base du tableau + ( indice * taille d'un élément )`. Cette adresse se calcule donc à partir de l'indice de boucle avec une équation de la forme : `Constante A + indice * Constante B`. Dans une boucle, toute variable dont la valeur suit une équation du type A + (indice * B) s'appelle une **variable d'induction**.

Et bien ce genre de calcul peut se simplifier : il suffit de remarquer qu'à chaque tour de boucle, on ne fait qu'augmenter l'adresse de la quantité B. Le code se simplifie alors en :

```
1 char* base = A;
2
3 for (int i = 0; i < ArraySize; ++i)
4 {
5     base += B;
6     *pointer = i;
7 }
```

Autrefois, cette optimisation était effectuée à la main. D'ailleurs, nombreux sont les programmeurs C qui pensent qu'effectuer cette optimisation à la main est encore utile de nos jours. Toutefois, aujourd'hui, le compilateur sait automatiquement reconnaitre les variables d'induction. Dans ce cas, il remplace cette équation par une addition, comme vu au-dessus, ce qui permet de supprimer une multiplication. Cette optimisation s'appelle la **réduction en force**.

2.2.6. Invariants de boucles

Les boucles sont définitivement une source d'optimisations assez importante. Il faut dire qu'un calcul dans une boucle sera exécuté de nombreuses fois (une fois par itération), ce qui fait que supprimer un seul calcul peut faire gagner quelques milliers de cycles d'horloge. Dans ces conditions, il faut sortir un maximum de choses des boucles.

Or, il arrive que certains calculs fournissent le même résultat à chaque tour de boucle : on parle d'**invariant de boucles**. Dans ces conditions, il vaut mieux sortir ces invariants de la boucle, histoire de ne les exécuter qu'une seule fois. Le compilateur sait détecter ces invariants : ils ne dépendent pas de l'indice de la boucle, ni de variables qui en dépendent indirectement. Inutile de faire cela à la main, donc. Sauf dans le cas des fonctions dont le compilateur ne peut dire si elles sont pures ou impures, encore une fois.

2.2.7. Suppression des calculs inutiles

Il arrive aussi que certains calculs soient inutiles. Par inutiles, on veut dire que leur résultat n'est pas utilisé dans le programme. Plus précisément, ces résultats sont écrits dans une variable qui n'est jamais lue. Il suffit que la variable soit dans une fonction pour que le compilateur puisse détecter qu'une variable ne sera jamais utilisée. Le compilateur peut alors supprimer les instructions qui ne servent qu'à calculer ce résultat, et ne servent à aucun autre calcul. Ces calculs inutiles apparaissent souvent après que la propagation de constantes a fait son travail.

2.3. Optimisations liées aux branchements

Les branchements sont une véritable plaie pour les processeurs modernes. Ils pourrissent le pipeline, sont parfois difficiles à prédire par l'unité de prédiction de branchement, etc. Mais ils sont aussi une véritable plaie pour le compilateur : les branchements empêchent certains déplacements de calculs : il est impossible de déplacer un calcul avant un branchement, de le faire sortir de la fonction qui le contient, etc. En conséquence, le compilateur cherche toujours à éliminer les branchements.

2.3.1. Inlining

Une optimisation bien connue est l'**inlining**. Cette technique consiste à éliminer les appels de fonction d'un programme : le corps de la fonction est intégralement recopié à chaque endroit où celle-ci est appelée. Il faut remarquer que cette technique a tendance à fortement augmenter la taille du code : au lieu d'un seul exemplaire de la fonction, celle-ci est recopiée en autant d'exemplaires qu'il y a de sites d'appel. Aussi, elle n'est effectuée que pour des fonctions relativement petites.

Cela permet d'éliminer l'appel de la fonction, ainsi que tout le code qui prépare l'appel. Dans certains cas, cela permet aussi d'améliorer l'usage de la mémoire cache (pas de sauts dans la mémoire). L'*inlining* permet aussi à d'autres optimisations de fonctionner plus efficacement, comme la propagation de constantes. Il permet aussi d'obtenir des blocs de code linéaires de « grande taille », permettant au réordonnancement d'instructions de fonctionner au mieux.

2.3.2. Bound checking

Dans certains langages de programmation, les accès à un tableau avec un indice invalide vont lever une exception ou faire planter le programme. Pour cela, le compilateur insère des comparaisons qui vérifient la validité de l'indice. Ces comparaisons prennent évidemment du temps de calcul, et le compilateur dispose d'optimisations pour en éviter qui seraient inutiles. Il peut notamment prouver que certains morceaux de code ne peuvent pas donner lieu à des accès en dehors du tableau, pour éviter d'ajouter les tests de validité de l'indice. C'est notamment le cas avec ce code :

```
1 int sum = 0;
2
3 for (int i = 0; i < Array.size(); ++i)
4 {
5     sum += Array[i];
6 }
```

2.3.3. Branchements dont la condition est constante

Une autre source d'optimisations provient des branchements de type `if..else` dont la condition est constante. Les conditions toujours vraies ou fausses apparaissent souvent suite à la propagation de constantes. Si jamais une condition se retrouve être toujours vraie ou fausse, alors le bloc `if..else` qui correspond est supprimé et seule la portion qui sera toujours exécutée sera alors intégrée dans le programme.

Par exemple, le code suivant :

```
1  int a;
2
3  if (true)
4  {
5      a += 6;
6  }
7  else
8  {
9      a += 9;
10 }
```

sera transformé en :

```
1  int a;
2
3  a += 6;
```

2.3.4. Fusion de branchements

Maintenant, prenons le cas d'un branchement A, dont l'instruction/adresse de destination est un autre branchement B, qui lui-même pointe sur une instruction C. Au cours de l'exécution, le branchement va envoyer le processeur sur B, qui va aussitôt brancher vers C : autant brancher directement vers C. Dans ce cas, le compilateur va modifier l'adresse de destination du branchement A pour qu'il pointe directement sur l'instruction C.

2.3.5. Déroulage de boucles

Comme dernière optimisation, le compilateur peut aussi effectuer des transformations sur les boucles. La transformation la plus basique est ce qu'on appelle le **déroulage de boucles**. Dérouler des boucles consiste à effectuer plusieurs tours de boucle d'un seul coup, en recopiant le corps de la boucle en plusieurs exemplaires. Cette technique est clairement utile pour diminuer le nombre de branchement exécutés : vu qu'on exécute les branchements à chaque itération, diminuer le nombre d'itérations permet d'en diminuer le nombre également. Évidemment, le

nombre d'itérations est modifié de manière à obtenir un résultat correct, histoire de ne pas faire des itérations en trop.

Exemple, prenons cette boucle :

```
int indice;

for (indice = 0; indice < taille_tableau; indice = indice + 1)
{
    a[indice] = b[indice] * 7;
}
```

Celle-ci peut être déroulée comme suit :

```
int indice;

for (indice = 0; indice < taille_tableau ; indice = indice + 4)
{
     a[indice] = b[indice] * 7;
     a[indice + 1] = b[indice + 1] * 7;
     a[indice + 2] = b[indice + 2] * 7;
     a[indice + 3] = b[indice + 3] * 7;
}
```

Pour obtenir un nombre d'itérations correct, les compilateurs utilisent généralement deux boucles : une qui est déroulée, et une autre qui traite les éléments restants. Par exemple, si je veux parcourir un tableau de taille fixe contenant 102 éléments, je devrais avoir une boucle comme celle-ci :

```
int indice;

for (indice = 0; indice < 100; indice = indice + 4)
{
     a[indice] = b[indice] * 7;
     a[indice + 1] = b[indice + 1] * 7;
     a[indice + 2] = b[indice + 2] * 7;
     a[indice + 3] = b[indice + 3] * 7;
}
for (indice = 100; indice < 102; ++i)
{
     a[indice] = b[indice] * 7;
}
```

Mais certaines boucles gagnent à être écrites de cette manière à la main, avec quelques optimisations. En effet, si je prends cette boucle :

```
unsigned somme = 0;

for (int i= 0; i < 10000; ++i)
{
     somme += tab[i];
}
```

le compilateur me donnera ceci :

```
unsigned somme = 0;

for (int i=0; i < 10000; i =  i + 4)
{
     somme += tab[i];
     somme += tab[i+1];
     somme += tab[i+2];
     somme += tab[i+3];
}
```

Or, le code le plus optimisé est celui-ci :

```
unsigned somme1 = 0;
unsigned somme2 = 0;
unsigned somme3 = 0;
unsigned somme4 = 0;

for (int i=0; i < 10000; i =  i + 4)
{
     somme1 += tab [i];
     somme2 += tab [i+1];
     somme3 += tab [i+2];
     somme4 += tab [i+3];
}

int somme = somme1 + somme2 + somme3 + somme4;
```

La raison est simple : les additions dans le corps de la boucle sont indépendantes les unes des autres, ce qui permet à un processeur superscalaire[3] ou à exécution dans le désordre d'effectuer chaque addition en parallèle. Et le même principe peut s'appliquer pour toute opération qui utilise une variable accumulateur dans une boucle, sous réserve que le calcul le permette (cela demande un calcul associatif).

3. Un processeur superscalaire est capable d'exécuter plusieurs instructions simultanément parmi une suite d'instructions.

2.3.6. Fusion de boucles

Enfin, dernière optimisation : la **fusion de boucles**. Cette technique consiste à fusionner deux boucles qui itèrent avec les mêmes indices en une seule.

Par exemple, le code suivant :

```
for (int j = 0; j < 100; ++j)
{
    array1[j] = 25;
}
for (int i = 0; i < 100; ++i)
{
    array2[i] = i;
}
```

donnera ceci :

```
for (int i = 0; i < 100; ++i)
{
    array1[i] = 25;
    array2[i] = i;
}
```

2.3.7. Loop unswitching

On a vu plus haut que les compilateurs sont capables de sortir certains calculs des boucles. Les calculs en question sont des calculs qui ne dépendent pas de l'indice de la boucle, que ce soit directement ou indirectement : on parle d'invariants de boucle. Il est possible d'utiliser l'extraction des invariants de boucle pour des conditions.

Par exemple, le code suivant :

```
for (i = 0; i < 1000; i++)
{
    if (w)
    {
        y[i] = 0;
    }
    else
    {
        x[i] += y[i];
    }
}
```

deviendra :

```
if (w)
{
    for (i = 0; i < 1000; i++)
    {
        y[i] = 0;
    }
}
else
{
    for (i = 0; i < 1000; i++)
    {
        x[i] += y[i];
    }
}
```

2.3.8. Inversion de boucle

Enfin, dernière optimisation : transformer les boucles `while` (ou `for`, ne soyons pas raciste), en boucle `do..while` entourées par un `if`. L'astuce consiste à vérifier avec le `if` si la boucle doit être exécutée au moins une fois, et à exécuter la boucle `do..while` si c'est le cas. Si jamais le compilateur peut prouver que la boucle est exécutée au moins une fois, alors le `if` devient inutile. Cette optimisation permet de supprimer une exécution d'un branchement, sans compter qu'elle permet aussi à certaines optimisations de fonctionner au mieux, notamment la suppression des variables d'induction.

2.4. Optimisations liées à la hiérarchie mémoire

Si les branchements et les calculs sont une source d'optimisations, il ne faut pas oublier que la majorité des instructions d'un programme consistent en des accès mémoire. De nos jours, un bon usage de la hiérarchie mémoire est une condition *sine qua non* pour obtenir de bonnes performances, que ce soit pour un compilateur ou un programmeur. Autant dire que les compilateur disposent de quelques optimisations sur le sujet. Mais pour autant, les compilateurs ne peuvent pas utiliser d'algorithmes *cache oblivious* à votre place, et encore moins changer des tableaux de structures en structures de tableaux. Soyez vigilant et optimisez à la main !

2.4.1. Rematérialisation

Il arrive souvent que l'on n'ait pas assez de registres pour stocker des variables temporaires. Dans ce cas, le contenu de certains registres est alors sauvegardé sur la pile pour une utilisation ultérieure : on parle de **spilling**. À ce petit jeu, les programmeurs assembleurs disposent d'une optimisation dont aucun compilateur actuel n'est capable : ils peuvent sauvegarder les registres, non pas sur la pile, mais dans les registres MMX ou SSE. Mais si cette optimisation n'est pas possible automatiquement, il vaut mieux recalculer certaines variables temporaires que de les

mémoriser temporairement dans un registre : on parle de **rematérialisation**. Cette optimisation est en quelque sorte l'inverse de celle qui supprime les expressions redondantes.

2.4.2. Loop spliting

Cette technique consiste à faire l'inverse de la fusion de boucle : des boucles qui itèrent sur des tableaux différents sont scindées en deux. Ainsi, chaque boucle travaillera indépendamment sur chaque tableau d'un bloc (on parcourt un tableau puis l'autre). Cette technique améliore la localité et la gestion du cache comparé à une seule boucle qui piocherait les données dans un tableau puis dans l'autre. Suivant la situation, cette optimisation sera plus ou moins efficace par rapport à la fusion de boucle : le compilateur dispose d'heuristiques pour déterminer quand utiliser telle ou telle optimisation suivant la situation.

2.4.3. Échange de boucles

Certains langages stockent les tableaux de tableaux ligne par ligne : des données d'une ligne sont consécutives en mémoire. Dans ces conditions, il vaut mieux parcourir les tableaux ligne par ligne, histoire de profiter de la mémoire cache du processeur et de diverses techniques de *prefetching*. D'autres langages fonctionnent colonne par colonne.

Dans ce cas, le compilateur peut parfois échanger l'ordre des boucles pour parcourir les tableaux ligne par ligne, ou colonne par colonne (selon le langage). Mais cela ne fonctionne que quand le compilateur peut prouver que la modification n'a pas d'effet de bord, ce qui n'est pas toujours possible. Par exemple, le compilateur n'a pas le droit de le faire dans un cas pareil :

```
for (int i = 0; i < nbLignes; ++i)
{
    for (int j = 0; j < nbColonnes; ++j)
    {
         tab[i][j] = fonction(i,j);
    }
}
```

Quand on sait que cette optimisation permet de gagner assez facilement en performances, parfois d'un facteur 10, mieux vaut être prudent et la faire à la main.

3. Optimisations de la taille du code

Si la vitesse des programmes peut être optimisée, les compilateurs peuvent aussi jouer sur la taille du programme. La taille d'un programme en elle-même n'est pas importante pour des programmes PC puisque la majorité d'entre eux possèdent plusieurs gibioctets[4] de RAM, ce qui laisse une grande marge. Par contre, diminuer la taille du code a des effets indirects sur les

performances en économisant de la mémoire cache. Mais dans la réalité, les optimisations de la taille du code entrent souvent en conflit avec les optimisations de vitesse d'exécution.

3.1. Choix des instructions

La première optimisation consiste à choisir les instructions les plus courtes quand on peut utiliser plusieurs instructions pour une même opération. L'exemple classique sur les processeurs x86 est la mise à zéro d'un registre : il vaut mieux effectuer un XOR qu'un MOV. Par exemple, XOR EAX EAX est plus court que MOV EAX 0. Le seul problème, c'est que les instructions courtes ne sont pas forcément les plus rapides, or le compilateur a tendance à choisir des instructions plus longues mais plus rapides.

3.2. Jonction de branchements

Il est possible que dans un `if...else`, certaines lignes de code soient recopiées à la fois dans le `if` et dans le `else`. Il arrive de plus que le résultat de ces lignes de code soit totalement indépendant du contenu du `else` ou du `if`.

Prenez cet exemple :

```
1  if (a > 0)
2  {
3      --a;
4      ++b;
5  }
6  else
7  {
8      ++a;
9      ++b;
10 }
```

On voit bien que la ligne `++b` peut être factorisée vu qu'elle est totalement indépendante du contenu du `if` et du `else`.

Un compilateur peut effectuer automatiquement cette transformation, ce qui donne :

```
1  ++b;
2
3  if (a > 0)
4  {
5      --a;
6  }
7  else
```

4. 1 gibioctet (Gio) = 2^{30} octets = 1.024 Mio = 1.073.741.824 octets

```
8  {
9      ++a;
10 }
```

3.3. Abstraction de procédure

Une autre technique très utile est l'**abstraction de procédure**. Cette méthode consiste à détecter des morceaux de code présents en plusieurs exemplaires dans un code source et les transformer en fonction : les exemplaires sont alors retirés du programme, et sont remplacés par un appel à la fonction créée. On peut voir cette technique comme l'exact inverse de l'*inlining*.

Autant vous dire que décider quand inliner ou utiliser l'abstraction de procédure est généralement un véritable casse-tête. Généralement, les compilateurs disposent de paramètres de compilation pour spécifier s'il faut optimiser pour la taille du code ou pour la vitesse : l'abstraction de procédure est utilisée quand on choisi l'option taille du code, alors que l'*inlining* est utilisé quand on choisi l'option vitesse.

3.4. Suppression de code mort

Un compilateur peut aussi détecter les morceaux de code qui ne seront jamais exécutés. Par exemple, il est capable de remarquer que le code d'un `else` ou d'un `if` dont la condition est constante ne sera jamais exécuté et peut alors supprimer ce code. Cela permet de gagner de la place assez facilement, surtout quand la propagation de constante fait son petit effet.

3.5. Autres optimisations

Il faut aussi signaler que presque toutes les optimisations qui permettent de simplifier les calculs ou de virer les calculs inutiles permettent de diminuer la taille du code. Enfin, il faut préciser que la fusion de boucles permet de diminuer la taille du code, vu que des branchements sont supprimés.

4. Conclusion

Et voilà, nous avons vu une partie des optimisations que tout compilateur digne de ce nom effectue automatiquement sur votre code source. Bien sûr, un compilateur peut effectuer d'autres optimisations un peu plus complexes sur votre code source, mais les optimisations vues au-dessus sont un aperçu assez intéressant, suffisant pour avoir une bonne idée de ce qui se passe quand vous compilez un programme.

Liste des abréviations

MMX MultiMedia eXtensions. 13

SIMD Single Instruction Multiple Data. 3

SSE Streaming SIMD Extensions. 13