
Laboratoire 3 : Optimisations de compilation

High Performance Coding - 2023

Francesco Monti

8.04.2023



Partie 1 - Optimisations de compilation

Sibling and Recursive Tail Call Removal

```
int loop(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return loop(n-1) * 2;  
    }  
}
```

Dans cet exemple d'optimisation, on est présenté avec une fonction récursive qui va prendre un nombre en paramètre et mettre 2 à la puissance de ce nombre. C'est également une fonction récursive terminale (tail call) car la fonction est appelée à la fin de la fonction. Ce genre de fonction peut poser problème si le nombre donné devient trop grand, car on va avoir un stack overflow. Pour éviter ce problème, on peut utiliser une fonction itérative qui va faire la même chose, mais sans utiliser la pile, comme montré dans `loop1()`. Si on connaît veut enlever directement la boucle on peut également faire du *Strength Reduction* et utiliser une opération bitshift pour faire la même chose, comme montré dans `loop2()`. De base le compilateur n'optimise pas la fonction, mais on peut spécifier le flag `-foptimize-sibling-calls` pour que le compilateur optimise les appels récursifs terminaux, comme montré dans `loop3()`. Dans ce cas le compilateur boucle tant que le paramètre `n` est plus grand que 0, et il va multiplier le résultat (préalablement initialisé à 1) par 2 à chaque itération.

Lien sur Compile Explorer : **Recursive Tail Call**

Function Inlining

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Cette fonction échange simplement deux valeurs entre elles. Elle est utilisée dans une boucle pour échanger les valeurs de deux tableaux entre-eux. Dans une version non optimisée la fonction est simplement appelée à chaque itération de la boucle. Une optimisation possible est d'*inline* le corps de la fonction dans la boucle, comme montré dans `swap1()`. Dans ce cas le compilateur va copier le corps de la fonction à chaque itération de la boucle, ce qui va augmenter la taille du code, mais va éviter les appels de fonctions. Une autre optimisation possible est d'utiliser le flag `-finline-functions` qui va demander au compilateur d'optimiser les fonctions en les *inlinant* si possible, comme montré

dans `swap2()`. Dans ce cas le compilateur va *inliner* la fonction `swap()` dans la boucle, ce qui va éviter les appels de fonctions et va également réduire la taille du code. Malheureusement, dans ce cas le compilateur n'arrive pas à *inliner* la fonction, soit parce qu'il trouve que ça vaut pas la peine, soit parce qu'il n'y arrive pas. Si on rajoute le *codeword* `inline` (ou le *built-in* `__inline__`) *Compile Explorer* ne compile pas la fonction `swap2()` mais il l'appelle quand même dans la boucle `main2()` ce qui mène à une erreur d'exécution.

Mais même si la fonction n'est pas *inlinée* par le compilateur elle est quand même plus petite en taille que la version optimisée manuellement. C'est parce que le compilateur va optimiser la fonction en utilisant des registres pour les variables locales, alors que la version optimisée manuellement va utiliser la pile pour les variables locales. C'est une des optimisations faite par l'option `-O1` de GCC.

Lien sur *Compile Explorer* : **Function Inlining**

Loops Unrolling

```
extern void check(int);

void main0(int **a){
    for (int i = -1; i <= 1; ++i){
        for (int j = -1; j <= 1; ++j){
            check(a[i][j]);
        }
    }
}
```

Ce code montre un exemple typique de boucle qui va effectuer une même opération à chaque itération. Dans notre cas, la fonction `check(int)` va être exécutée sur tous les voisins de la case `a[0][0]`. Une optimisation possible est de dérouler la boucle, c'est-à-dire de répéter le corps de la boucle plusieurs fois, comme montré dans `main1()`. Dans ce cas le compilateur va répéter le corps de la boucle 9 fois, une fois pour chaque itération de la double boucle. Cela va augmenter la taille du code, mais va éviter les sauts et les comparaisons à chaque itération de la boucle. Une autre optimisation possible est d'utiliser le flag `-funroll-loops` qui va demander au compilateur d'optimiser les boucles en les déroulant si possible, comme montré dans `main2()`. Dans ce cas le compilateur va dérouler la boucle que 3 fois, ce qui est permet déjà de réduire la taille du code et d'éviter les sauts et les comparaisons à chaque itération de la boucle.

Une autre possibilité qu'offre le déroulement de boucle est celui de permettre une éventuelle parallélisation du code. Si les appels à la fonction `check()` sont indépendants et ne modifient pas de données, on pourrait les appeler en parallèle. Cela permettrait d'augmenter la performance du code, mais il faut faire attention à ne pas avoir de *racing conditions*. Encore une fois le compilateur en sait

plus que nous et il peut décider de dérouler la boucle ou non, en fonction de la machine cible et de la taille de la boucle.

Lien sur Compile Explorer : **Loops Unrolling**

Partie 2 - Optimisations de l'algorithme

Maintenant nous pouvons essayer d'optimiser notre code A* en utilisant les optimisations de l'algorithme que nous avons vu en cours. Avant de regarder ce que le compilateur va faire, on va essayer de parcourir le code et de voir ce qu'on peut optimiser.

Inlining de fonctions

MinHeap

Pour rendre le code plus lisible et plus réutilisable, certaines fonctions ont été créées alors qu'elle ne contiennent que peu d'instruction. Ce serait intéressant d'*inline* ces fonctions pour réduire l'overhead et optimiser le programme. Une telle fonction est `swap_nodes(MinHeap, int)` dans `min_heap.c`, une implémentation de tas pour stocker les noeuds "ouverts". Cette fonction `swap_nodes` va simplement échanger deux noeuds du tas, et elle est appelée dans 4 autres fonctions. Une première possibilité serait de remplacer manuellement toutes les occurrences de `swap_nodes` avec le corps de la fonction. C'est une manière efficace d'optimiser mais pas très élégante. Et la maintenance du code devient plus compliquée. Une autre manière est de demander au compilateur de nous *inline* la fonction lui-même et rajoutant le mot-clé `inline` devant la signature de `swap_nodes`. On peut voir les différentes manières dans ce snippet sur *Compile Explorer*.

Dans ce snippet on peut voir l'implémentation de `swap_nodes(MinHeap, int)` aux lignes 19-24 de la source, et leur code assembleur aux lignes 1-10 de l'assembleur. Pour forcer la compilation de `swap_nodes` individuellement, on a ajouté

```
extern void swap_nodes(MinHeapNode *a, MinHeapNode *b);
```

au code pour la démonstration. Dans la suite du code on verra que cette fonction n'est jamais appelée.

On peut voir l'effet de la compilation avec `inline` qui va copier le corps de la fonction dans les endroits où elle a été appelée (lignes 17-30 et 56-68). Les deux "copies" ont presque le même nombre d'instructions que l'original (13 et 14 contre 10), mais elle enlèvent le besoin d'accéder à la pile. Finalement, une fonction manuellement optimisée est présente aux lignes 56-68 de la source. On peut voir que le compilateur va donner une version équivalente à la version précédente. Le choix va donc se porter sur la version la plus lisible et la plus facile à maintenir, c'est-à-dire la version avec le mot-clé `inline`.

A*

Dans l'implémentation de l'algorithme principal, la fonction `reconstruct_path(Grid *grid, Position **parent_set, Position **path)` sert à reconstruire le chemin trouvé à partir des parents de chaque noeud. Cette fonction est appelée à deux reprises, dans les deux fonctions de recherche. Il serait donc intéressant de l'*inline* également. Comme le corps de la fonction n'est pas grand et elle n'est appelée que deux fois en tout on a un gain de temps considérable par rapport à l'augmentation de taille du fichier compilé.

Loop Unrolling**Tab**

Une optimization possible qui pourrait également permettre d'ultérieures optimisations ou mise en parallèle de l'algorithme serait le déroulement de boucle. Dans notre cas on pourrait dérouler la boucle qui va checker les voisins d'un noeud et les traiter. Dans la méthode *tab* de résolution, on considère les 8 voisins d'un noeud, et on fait une double boucle pour les traiter. On pourrait dérouler cette boucle pour éviter les sauts et les comparaisons à chaque itération. Pour commencer on peut la dérouler une fois, et ne faire plus qu'une seule boucle `for` et dans celle-ci mettre 3 fois la fonction `check_neighbour()`. On peut voir le résultat dans ce snippet sur *Compile Explorer*.

Les lignes 1-106 de l'assembleur sont la compilation de la boucle actuelle (tronquée pour plus de lisibilité). Ensuite les lignes 107-194 sont la compilation de la méthode qui encapsule le traitement d'un voisin et les lignes 195-224 sont la version optimisée par le compilateur. La version manuelle vient par la suite et on peut voir qu'elle est tout de suite plus longue, mais on évite également une série de comparaisons et de jumps. On peut aller plus loin et pousser le déroulement au bout en éliminant la deuxième boucle `for` et en spécifiant tous les voisins directement. Dans notre cas la différence n'est pas très grande, mais on peut imaginer que pour un plus grand nombre de voisins, le gain de performance serait plus important. Il est intéressant de remarquer que GCC même avec l'option *-funroll-loops* ne veut pas le faire.

Un désavantage de créer une fonction supplémentaire est que on va faire plus d'appels à des fonctions et donc passer pas la pile. On pourrait coupler de déroulement à de l'inlining, si on est pas trop gênés par la taille de l'exécutable. On peut proposer au compilateur de le faire en mettant le mot-clé `inline` devant la signature de la fonction.

Struct

Une autre possibilité de *Loop Unrolling* est dans le traitement des voisins mais dans la méthode *struct* dans celle-ci on examine uniquement les 4 voisins adjacents, ce qui nous permet de dérouler la boucle une fois. On va donc passer de

```
check_neighbours(current, /* ... */);
```

à

```
Grid_Component *current_node = (Grid_Component *)current.data;
check_neighbours_struct(current, /* ... */, -1, 0, current_node->up);
check_neighbours_struct(current, /* ... */, 1, 0, current_node->down);
check_neighbours_struct(current, /* ... */, 0, -1, current_node->left);
check_neighbours_struct(current, /* ... */, 0, 1, current_node->right);
```

On peut voir que le code est plus verbeux et moins flexible, on doit faire attention à ne pas se tromper en mettant les offsets et en passant les arguments. Mais grâce à ça GCC va même décider d'*inline* les appels de la fonction, ce qui enlève le désavantage de devoir passer par la pile.