

## HPC - Lab 05

Aubry Mangold  
ISCS, HEIG-VD  
aubry.mangold@heig-vd.ch

### Introduction

### Familiarisation

Les deux exécutables fournis sont compilés avec les options `march=native` et `O2`. Ceci implique que le code sera déjà transformé pour être optimisé pour la machine sur laquelle il est compilé. La lecture du fichier exécutable compilé sur la machine de test permet d'observer l'utilisation de SIMD par le compilateur pour vectoriser les opérations (comparaisons dans le cas de l'exécutable `analyze`). Cette transformation implique que le programme invoquera moins de fois les fonctions qui ont été linéarisées.

Perf est exécuté avec les deux fichiers binaires fournis. La taille du jeu de données choisi est de 10000000. L'utilisation d'une plus petite quantité de données — impliquant un temps d'exécution plus court — ferait que l'échantillonnage effectué par Perf ne serait pas représentatif de l'exécution du programme. Le paramètre `--call-graph dwarf` est utilisé pour que les appels de fonctions soient tracés.

Les outils `memcheck`, `cachegrind` et `callgrind` de la suite Valgrind sont aussi exécutés sur les deux programmes fournis. La taille choisie pour le jeu de données est de 100000 éléments. Ce choix est fait pour qu'il y ait assez de données à analyser mais que le temps d'exécution ne soit pas trop grand (l'instrumentation des binaires par Valgrind augmente le temps d'exécution de manière significative).

### create-sample

Ce programme génère des données aléatoirement. Le Flame Graph Figure 1 généré avec les données de Perf indique que le programme est relativement bien optimisé et que les fonctions les plus chaudes sur le CPU sont `rand_nd` et `fprintf`. Les données Valgrind corroborent cette observation et indiquent que la charge d'exécution modérée est principalement due à l'écriture de données dans le fichier de sortie. La performance est donc liée aux opérations IO. Valgrind ne détecte aucun problème de mémoire.

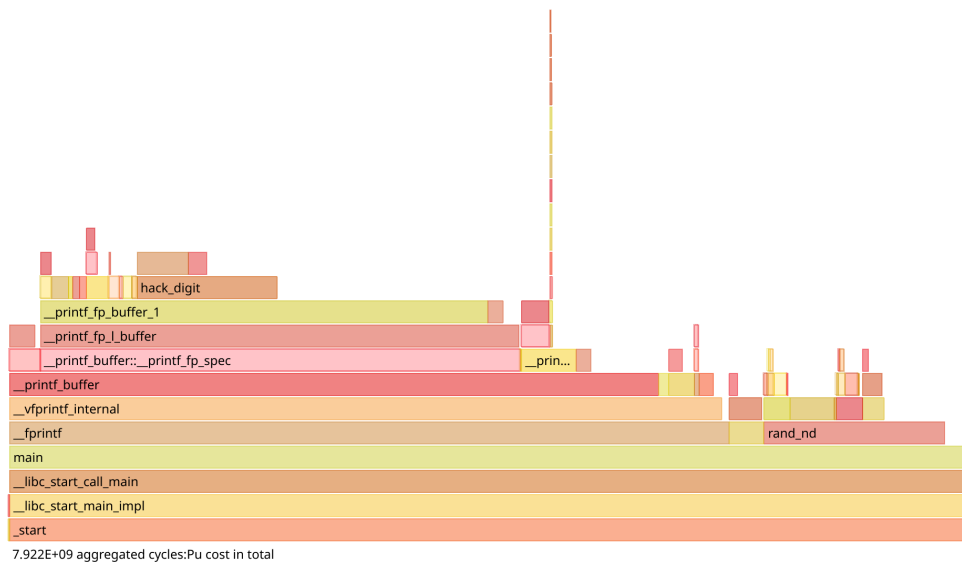


Figure 1: Flame Graph de l'exécution du programme create-sample

Si l'on voulait encore optimiser ce programme, on peut s'intéresser à la génération de nombres aléatoires qui pourrait être effectuée avec par exemple un xorshift. Cet algorithme est plus rapide que le `rand()` de C et serait ici acceptable car les nombres générés ne doivent pas être cryptographiquement sûrs. Au sujet de l'écriture de fichier chronophage, il serait possible de bufferiser les sorties et d'utiliser l'appel système `write` pour écrire le fichier d'un coup.

On observe une "flèche" dans Figure 1. Cette dernière est due à des chaînes d'interruptions (dans notre cas de la relocalisation de données) enchâssées qui doivent être traitées par le processeur.

## analyse

Ce programme trie les mesures. Le Flame Graph Figure 2 tiré des données de Perf et les données de Valgrind indiquent que beaucoup de temps de processeur est passé à effectuer des comparaisons. Le programme souffre cependant d'un manque d'efficacité majeur car la fonction `getc` effectue une recherche linéaire  $O(n)$  dans un tableau non trié pour chaque ligne lue, entraînant une complexité quadratique. Cela explique la forte quantité de références à la fonction trouvées par Callgrind et Perf. Bien que le tri final avec `qsort` soit optimal, il est mis en abîme par la recherche inefficace. L'output de Memcheck indique l'absence de fuites mémoire. Cachegrind indique que la localité des données est probablement médiocre, dégradant l'utilisation du cache.

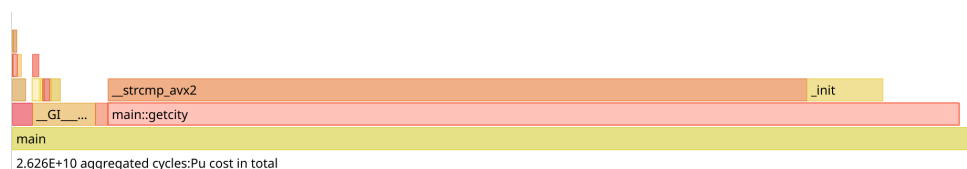


Figure 2: Flame Graph de l'exécution du programme analyze

Une refonte avec une table de hachage (de complexité  $O(1)$  pour les insertions/recherches) ou un tri préalable des entrées pour permettre une recherche binaire ( $O(\log n)$ ) réduirait drastiquement le temps d'exécution (surtout pour un grands jeu de données).

## Observations

L'utilisation de Perf et de la suite d'outils Valgrind a permis de mettre en lumière les hotspots du code. A ETENDRE.

## DTMF

A l'instar du programme de familiarisation, les outils Perf et Valgrind sont aussi utilisés sur la suite de programmes encodeurs et décodeurs DTMF. Les paramètres sont ajustés a des valeurs appropriées afin de trouver un équilibre entre un temps d'exécution suffisamment grand pour que les outils puissent faire leur travail sans que trop d'attente soit nécessaire. Pour Perf, une taille d'échantillon de 260 caractères et choisi. Pour Valgrind, en mode encodage, une taille d'échantillon de 36000 caractères est utilisées contre 3600 en mode décodage.

## Encodage

Le programme analysé est l'encodeur DTMF qui traduit des caractères en une tonalité. Perf et Valgrind ont été utilisés pour analyser le programme. Les données de Perf ont été utilisées pour générer le Flame Graph présenté à Figure 3.

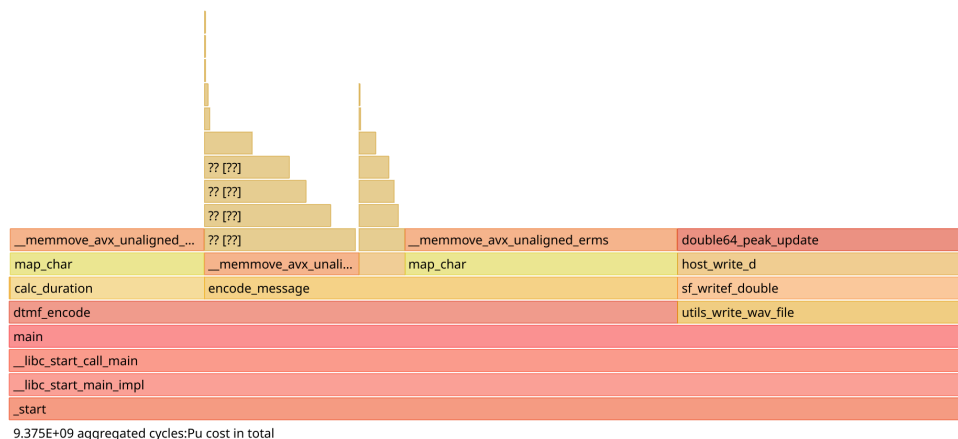


Figure 3: Flame Graph de l'exécution de l'encodage par le programme DTMF

Les données de Perf relève les informations suivantes sur les fonctions suivantes sont chaudes sur le processeur :

- `map_char`: la recherche linéaire dans `dtmf_table_global` pour mapper les caractères est très coûteuse.
- `utils_write_wav`: l'écriture du buffer audio dans le fichier WAV est coûteuse en raison d'opérations IO non optimisées.
- `encode_message`: des appels répétés à `map_char` et `memcpy` utilisent beaucoup de temps en CPU.
- `calc_duration`: la fonction effectue aussi des appels répétés à `map_char`.

Les outils de la suite Valgrind ne décèlent pas de fuites de mémoire. Call/Cachegrind révèle qu'une forte quantité de données sont utilisées et que la localité de ces dernières est bonne.

Des optimisations possibles incluent :

- `map_char` : utiliser une recherche binaire ou une table de hachage.
- `utils_write_wav` : bufferiser les écritures et utiliser un appel système `write` pour écrire le fichier en un bloc.
- `encode_message` : vectoriser ou paralléliser les appels à `memcpy` et `map_char`.
- `calc_duration` : mettre en place un cache pour éviter les appels répétés à `map_char`.

## Decodage Goerzel

Le programme analysé est le décodeur DTMF basé sur l'algorithme Goertzel. Les outils Perf et Valgrind sont utilisés afin d'identifier les hotspots du programme. Les données de Perf ont été utilisées pour générer le Flame Graph présenté à Figure 4.

Perf révèle un goulot d'étranglement dans le pré-traitement car la fonction `_dtmf_preprocess_buffer` investi beaucoup de temps de CPU à appliquer le filtre bandpass — à l'instar des autres filtres relativement peu coûteux. De plus, la majeure partie du temps d'exécution du programme est passé dans la fonction de processing de fenêtre qui appelle la détection Goertzel deux fois pour chaque fenêtre (fréquence haut et basse).

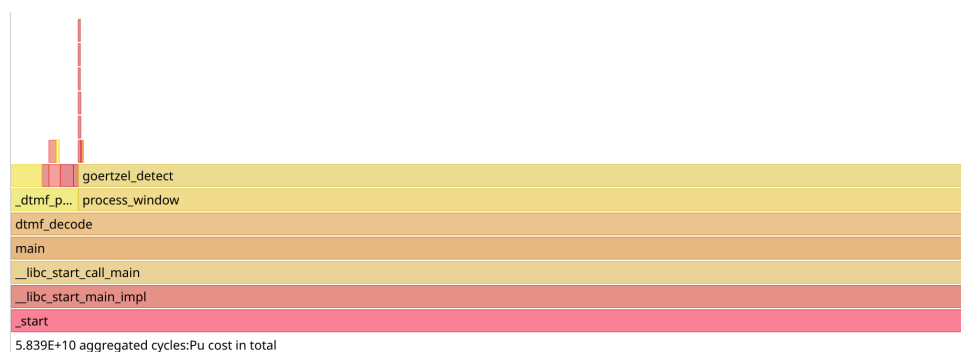


Figure 4: Flame Graph de l'exécution du décodage par le programme DTMF version Goerzel

Memcheck ne détecte pas de fuite de mémoire. Ce résultat est à prendre avec précaution car Libasan révèle des fuites de mémoire pour certains paramétrages. Callgrind indique une charge de calcul élevée pour les fonctions de filtre bandpass et de détection Goertzel. Ce résultat fait sens puisque ces deux fonctions effectuent des calculs mathématiques qui n'ont pas été optimisés.

Des optimisation potentielles seraient :

- `_dtmf_apply_bandpass` : utiliser un filtre FIR plutôt que IIR afin de pouvoir vectoriser la fonction, voir directement utiliser FFTW3 pour calculer le filtre.
- `goertzel_detect` :
  - Fusionner les calculs des deux fréquences en une seule passe.
  - Précalculer les coefficients  $k$  et  $\Omega$  pour toutes les fréquences DTMF à l'initialisation.
  - Vectoriser la boucle de calcul.
  - Mettre les fenêtres de Hamming en cache pour éviter de recalculer une même valeur.
- En générale, les fonctions de preprocessing et de traitement de fenêtre pourraient être inlinées afin de réduire le nombre d'appel de fonction et par conséquent de stack frames.

## Decodage FFT

Le programme analysé avec Perf et Valgrind est le décodeur DTMF basé sur l'algorithme de transformation de Fourier rapide. Les données de Perf sont utilisées pour générer le Flame Graph présenté à Figure 5.

On observe avec les données de Perf que la fonction `dtmf_decode` passe approximativement la moitié du temps de processeur au pré-traitement et l'autre moitié à l'application de la FFT au travers de la librairie FFTW3. Cette librairie est populaire et déjà bien optimisée et le Flame Graph ne révèle pas de comportement anormal lors de l'invocation de la FFT. Dans le cas du pré-traitement — à l'instar du décodeur basé sur l'algorithme Goertzel — la fonction de filtre bandpass est la plus chaude sur le processeur.

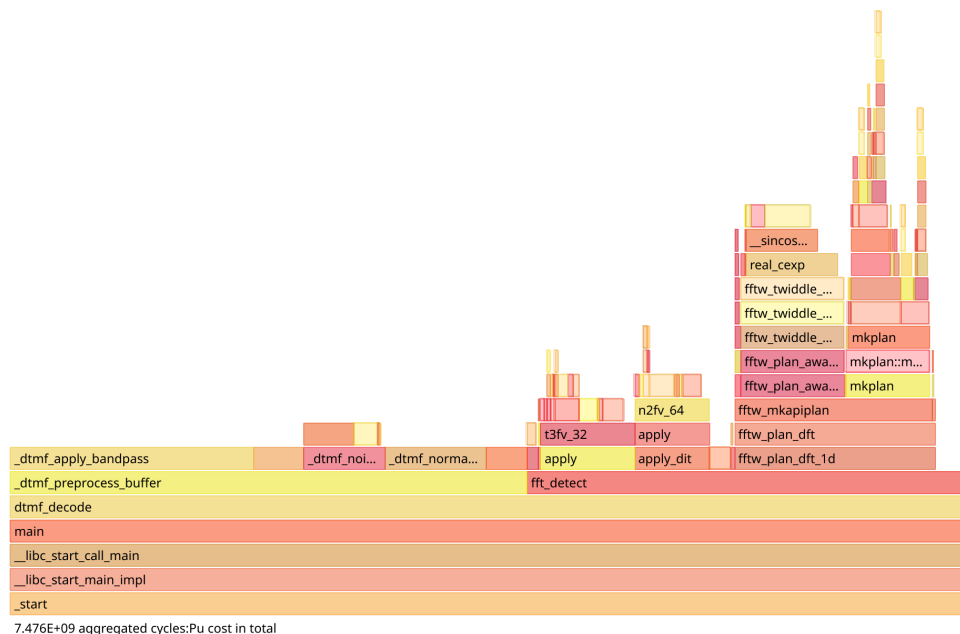


Figure 5: Flame Graph de l'exécution du décodage par le programme DTMF version FFT

Memcheck révèle que la librairie FFTW3 ne libère pas toutes les ressources utilisées. Cette technique est souvent utilisée dans les bibliothèques C

## Conclusion