HPC - Lab 05

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

Introduction

Ce rapport relate l'utilisation des outils Perf et Valgrind pour l'analyse de hotspots de programmes dans le cadre du cours High-Performance Computing de la HEIG-VD.

Topologie du système

La machine de développement, de test et de banchmark est basée sur un processeur AMD Ryzen 7 Pro 6850U, a 8 coeurs physiques et 16 threads. Chaque coeur dispose de 32 KiB de cache L1 données et 32 KiB de cache L1 instructions, de 512 KiB de cache L2 par cœur (4 MiB au total) et d'un cache L3 global de 512 KiB. Le système possède 32 GB de RAM DDR4. L'OS utilise le kernel Linux 6.14.3.

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 fichier 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 ai 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 fonction 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 dues à l'écriture de données dans le fichier de sortie. La performance est donc liée aux opérations IO. Valgrind ne décèle aucun problème de mémoire.

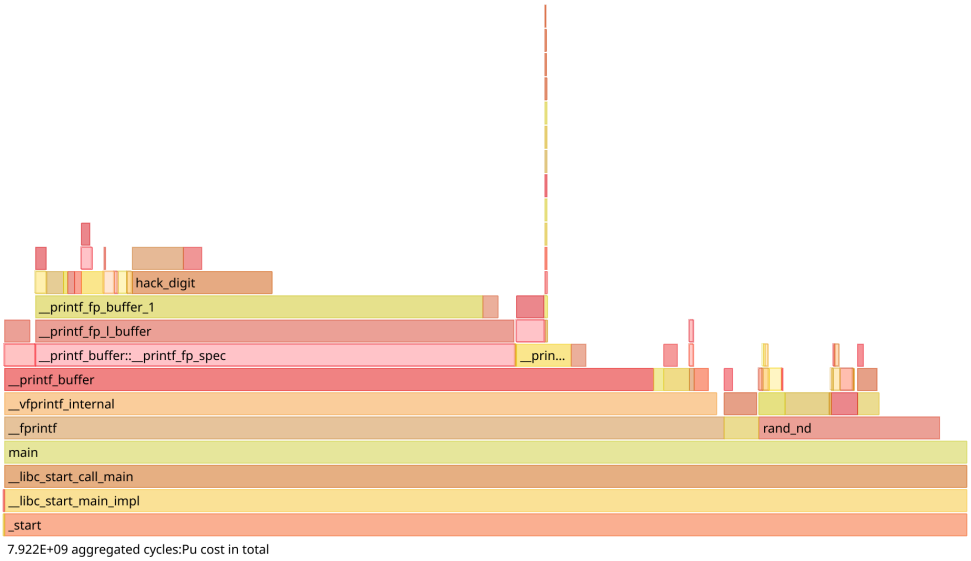

Figure 1: Flame Graph de l'exécution du programme `create-sample`

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.

`analyze`

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 a effectuer des comparaisons. Le programme souffre cependant d'un manque d'efficience majeur car la fonction `getcity` 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 abime 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.

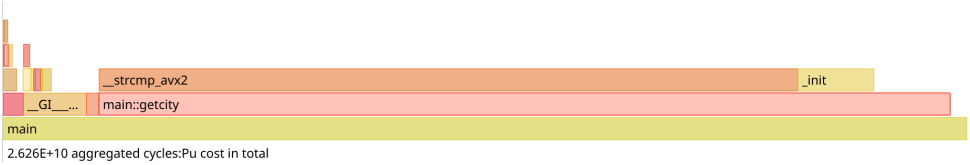

Figure 2: Flame Graph de l'exécution du programme `analyze`

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 en $(O(\log n))$ qui réduirait drastiquement le temps d'exécution (surtout pour un grands jeu de données).

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.

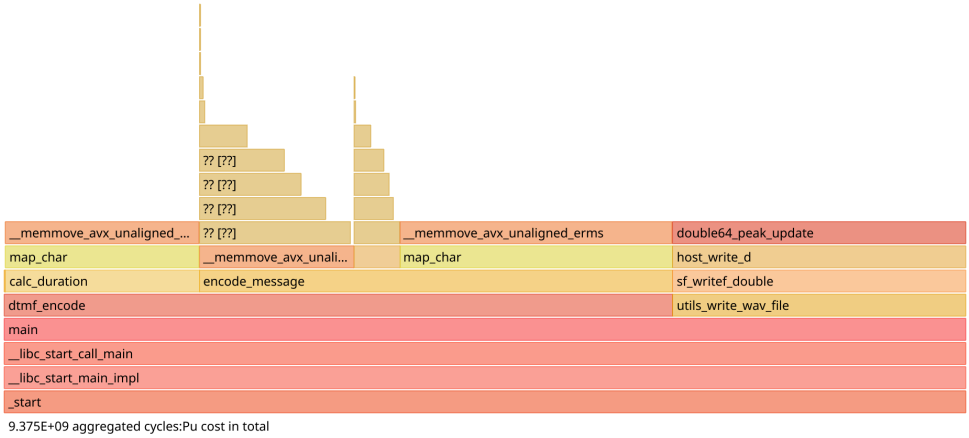

Figure 3: Flame Graph de l'exécution de l'encodage par le programme DTMF

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` — a 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).

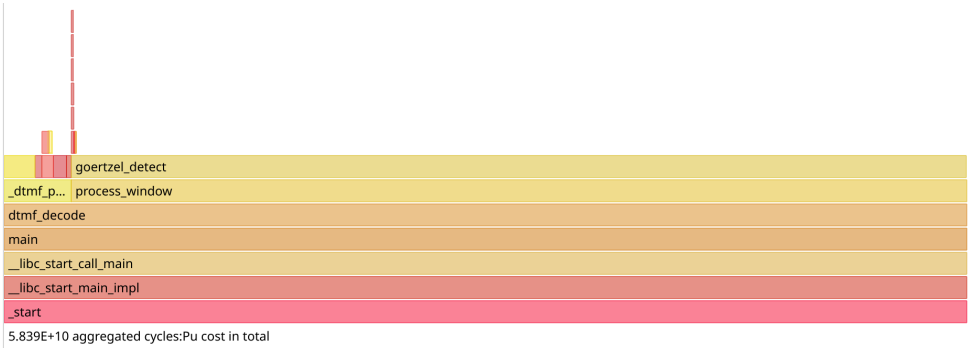

Figure 4: Flame Graph de l'exécution du décodage par le programme DTMF version Goerzel

Memcheck ne détecte pas de fuite de mémoire. Ce résultat est a 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 passe-bande et de détection Goertzel. Ce résultat fait sens puisque ces deux fonctions effectuent des calcules 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 Ω 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 préprocessing 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 Fourrier 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 prétraitement 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 prétraitement — à l'instar du décodeur basé sur l'algorithme Goertzel — la fonction de filtre passe-bande est la plus chaude sur le processeur.

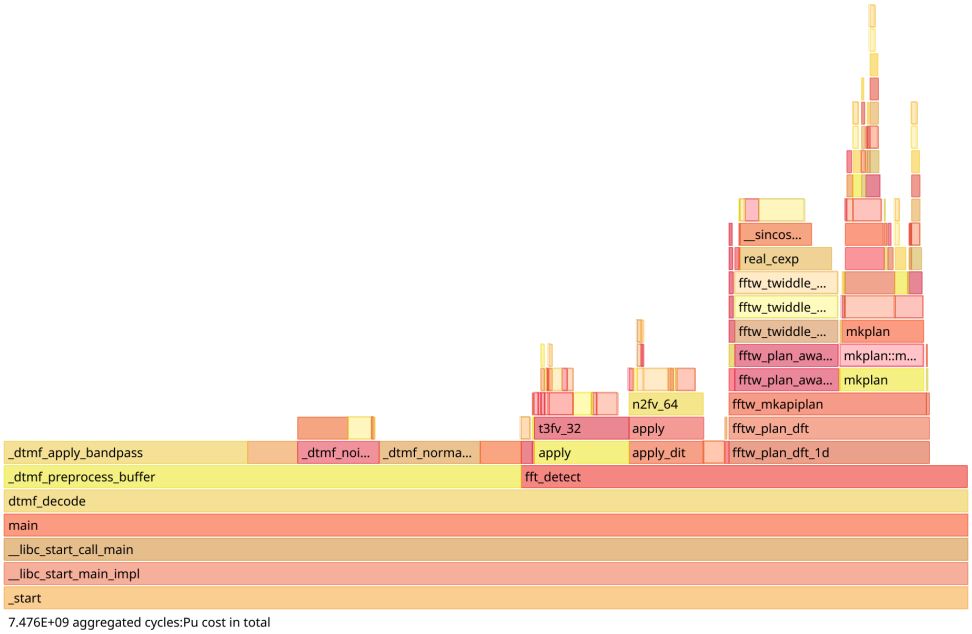

Figure 5: Flame Graph de l'exécution du décodage par le programme DTMF version FFT

Memcheck révèle que la librairie FFTW3 ne libère pas toutes les ressources utilisées. Cette technique est souvent utilisée dans les librairies C pour éviter de devoir re-allouer trop souvent de la mémoire. Il serait cependant possible de corriger ce comportement en invoquant explicitement les fonctions de nettoyage de la librairie FFTW3.

Quelques améliorations potentielles sont les suivantes :

- `_dtmf_apply_bandpass` : même remarques que pour dans le cas Goertzel.
- `fft_detect` :
  - Précalculer les indices de fréquences `low_idx` et `hi_idx`.
  - Vectorise le calcul des hypoténuses.

Optimisation du filtre bandpass

L'analyse a montré que le filtre passe-bande — utilisé par les 2 versions du décodeur — est un goulot d'étranglement majeur. L'optimisation proposée est donc de remplacer le filtre IIR par un filtre FIR vectorisé.

La première tentative d'optimisation de la fonction passe-bande est effectuée avec un filtre FIR. Cette solution a engendré des pertes de performance. Du au fait que FIR est moins efficace que IIR si pas linéarisé. Des performances similaires étaient pourtant attendues et il semble que l'implémentation n'était pas efficace.

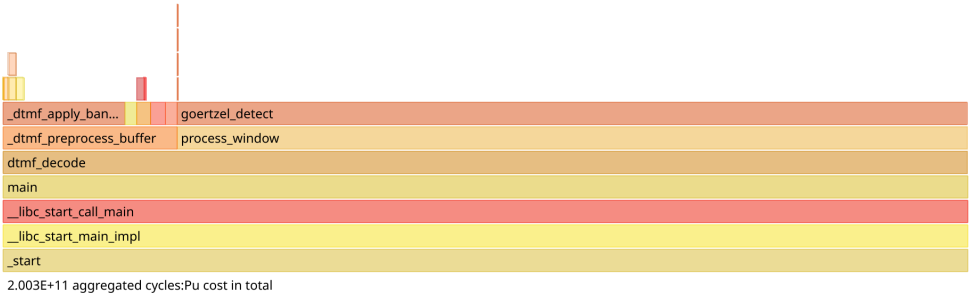

Figure 6: Flame Graph de l'exécution du décodage par le programme DTMF version Goerzel après modification du filtre passe-bande

Une deuxième tentative d'optimisation est faite en utilisant SIMD avec le filtre FIR. Cette solution n'a pas non plus abouti car le nouveau filtre s'avère être trop coûteux en temps de calcul. Les résultats obtenus au travers d'une analyse avec Perf et Valgrind présentés dans Figure 6 montrent que le filtre présente d'encore moins bonnes performances que la première tentative d'amélioration. Le code a été maintenu en commentaire dans le fichier `dtmf_common.c`.

Aucune autres optimisations n'ont été tentées.

Conclusion

L'utilisation de Perf et de la suite d'outils Valgrind permet de mettre en lumière les hotspots du code. Perf est un outil efficace pour identifier les fonctions chaudes sur le processeur et les chaînes d'interruptions. Valgrind permet de détecter les fuites de mémoire, d'analyser la localité des données et les chaînes d'appel. Les deux outils sont complémentaires et permettent d'optimiser le code de manière significative.

La création de Flame Graphs permet de visualiser les hotspots du code et d'identifier les fonctions qui consomment le plus de temps CPU. Ces graphiques sont particulièrement utiles pour comprendre le comportement du code et pour identifier les fonctions couteuses. L'utilisation de ces outils permet d'obtenir une vue plus générale du comportement du programme et permet ainsi de mieux comprendre les goulots d'étranglements, menant à la découverte d'opportunités d'optimisation qui seraient potentiellement passées inaperçues si on avait uniquement observé le code.

Ultimement, l'utilisation d'outils d'analyse est essentielle pour cibler les efforts d'optimisation sur les parties du code qui en ont le plus besoin.