LINFO2241 : P2

Louvain-la-Neuve, 13 novembre 2022

Groupe 17

ALLEGAERT	*Hadrien*	0799-18-00
	hadrien.allegaert@student.uclouvain.be	
DEGRAEUWE	*Jérémy*	2389-16-00
	jeremy.degraeuwe@student.uclouvain.be	

1 Introduction

Dans cette deuxième partie il nous est demandé d'optimiser la première version de notre serveur pour profiter, notamment, de la mise en cache du CPU.

Comme demandé, nous avons implémenté une multiplication matricielle avec la méthode de ligne par ligne et un *loop unrolling* pour la boucle interne.

La taille de la clé étant soit 8 soit 128 (multiple de 8), faire du *loop unrolling* de 8 permet de ne pas avoir à mettre en place de *strip mining*, c'est donc ce que nous avons fait.

2 Notation et remarque préliminaires

Dans cette première partie du rapport, nous allons principalement étudier 4 versions de l'implémentation :

- Version 1 (V1) : Multiplication ligne * colonne, sans *loop unrolling*
- Version 2 (V2) : Multiplication ligne * colonne, avec *loop unrolling* de 8 sur la boucle interne
- Version 3 (V3) : Multiplication ligne * ligne, sans *loop unrolling*
- Version 4 (V4) : Multiplication ligne * ligne, avec *loop unrolling* de 8 sur la boucle interne

Analyser ces 4 versions permet de bien mettre en évidence les effets de chacune des méthodes ainsi que leur interaction.

Puisque les résultats de la commande `perf` ne sont pas exactes, nous avons pris toutes les mesures une dizaine de fois. Les données et graphes qui seront présentés sont donc toujours le résultats de 10 tests identiques par jeu de paramètres.

Ces prises de mesures ont été réalisées sur les machines de la salle Intel, en local.

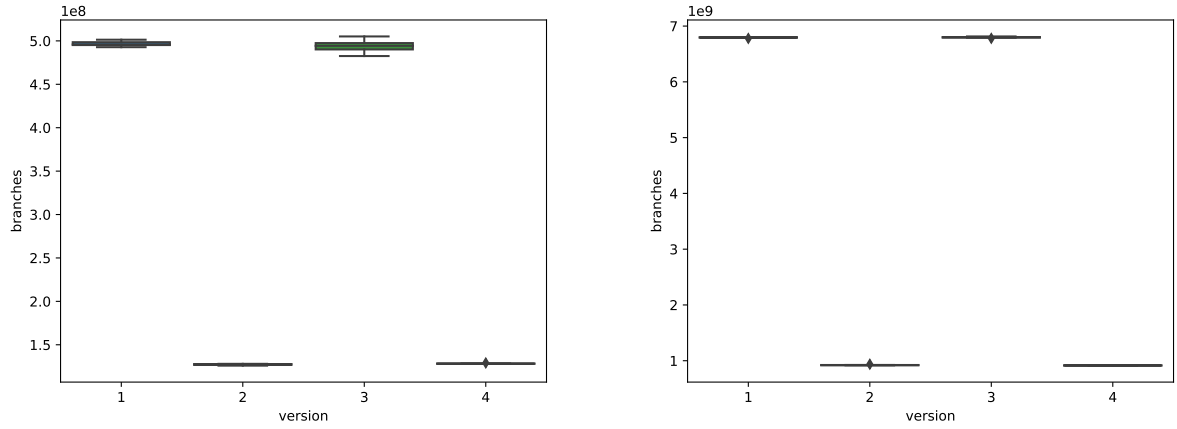
Enfin, pour être certain de ne pas "polluer" le code avec une partie inutile, nous avons tiré profit de l'argument de compilation `-OPTIM=X` afin de ne compiler **que** le code réellement nécessaire, et non pas compiler toutes les versions et choisir lors de l'exécution celle à utiliser.

3 Type de multiplication et *loop unrolling*

Nous allons nous servir d'une série de statistiques afin de repérer, mettre en évidence et interpréter les effets de chaque méthode.

3.1 Instructions et branchements

Un premier élément qu'on peut regarder est le nombre d'instructions de branchement. En effet, c'est un facteur primordial car il intervient doublement : dans le nombre total d'instructions en évitant des instructions *compare* et *jump*, et aussi en permettant d'éviter des erreurs de spéculation. Voici, à la figure 1, les résultats pour les 4 versions citées précédemment, pour des clés de 8 et de 128.



(a) Nombre de branchements pour une clé de dimension 8

(b) Nombre de branchements pour une clé de dimension 128

FIGURE 1

L'effet de diminution du nombre d'instructions et de branchements est important pour les deux versions qui profitent du *loop unrolling* (les graphes pour le nombre total d'instructions sont identiques à ceux présentés ci-dessus). En revanche, on ne voit pas de différence significative entre la multiplication ligne * colonne et ligne * ligne sur ces facteurs.

3.2 Cache

Ce qu'on va observer dans cette partie, c'est que l'utilisation d'une clé de dimension 8 ne va pas permettre de tirer autant profit du cache qu'une clé de 128. En effet, même avec une multiplication ligne * ligne, le changement de sous-multiplications de matrices est trop fréquent, ce qui empêche de profiter du chargement en cache des données des sous-matrices.

Pour une clé de 8, nous obtenons pour le pourcentage de *miss* sur la statistique `cache-misses` le résultat suivant :

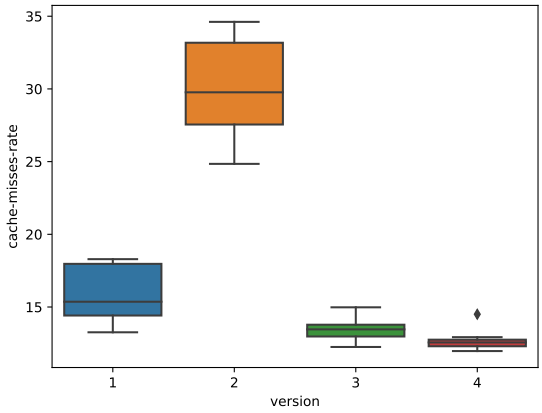


FIGURE 2 – Pourcentage de *miss* sur l'accès au cache pour une clé de 8

On voit que les améliorations ne sont pas significatives sur ce facteur pour une clé de 8. Et on remarque même que la méthode du *loop unrolling* seule dégrade ce facteur.

En revanche, pour une clé de 128, on remarque une très nette amélioration sur l'accès au cache L1, passant d'environ 50% de *miss* pour la multiplication ligne * colonne à moins de 5% pour la multiplication ligne * ligne (figure 3). Ceci est d'ailleurs le facteur le plus déterminant dans la performance globale du programme, et donc la modification ayant amené le plus grand gain de score sur INGInious.

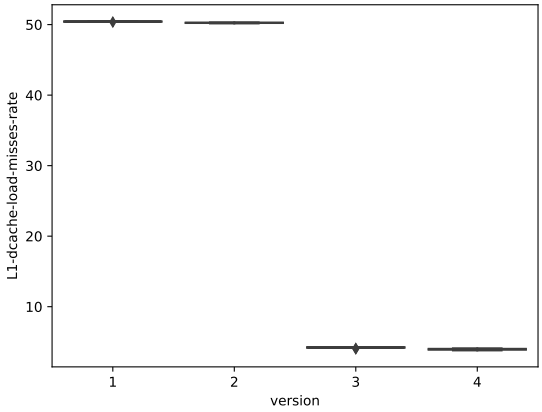


FIGURE 3 – Pourcentage de *miss* sur l'accès au cache L1 pour une clé de 128

Une analyse plus détaillée à l'aide des commandes `perf record` et `perf report` montre, comme on peut s'y attendre, que tout se joue dans la boucle la plus interne, là où se calcule les valeurs de la multiplication matricielle :

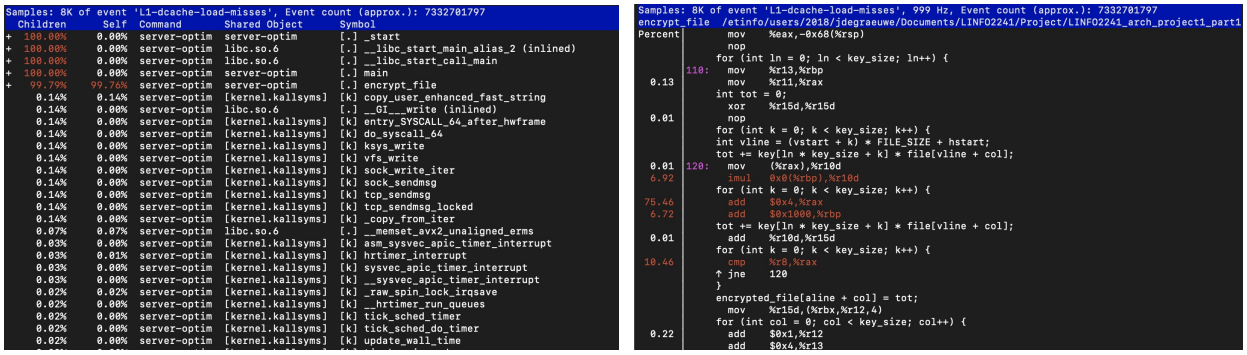


FIGURE 4 – Résultats du `record` sur le cache L1 pour la version 1 et une clé de 128

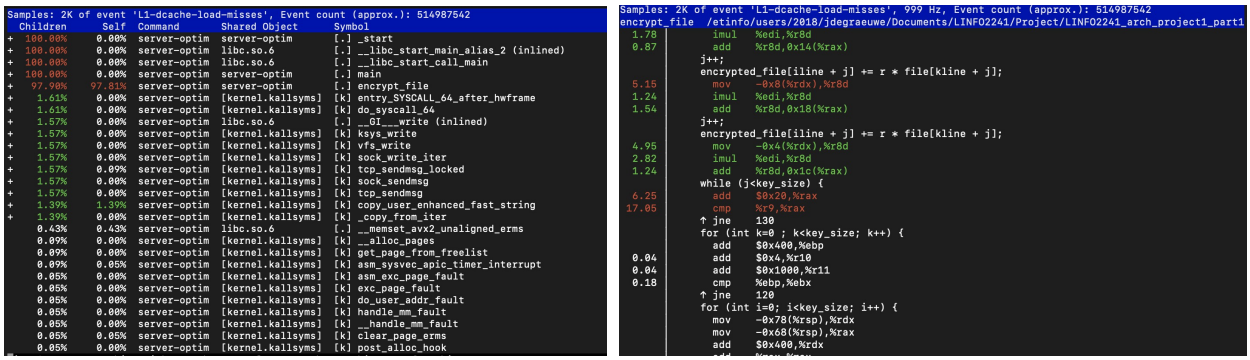
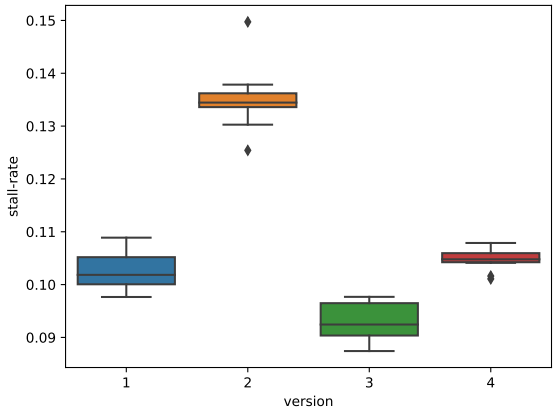


FIGURE 5 – Résultats du `record` sur le cache L1 pour la version 4 et une clé de 128

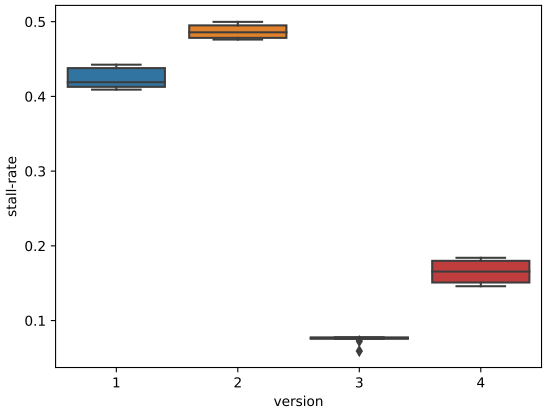
Ces résultats sont donc explicitent et correspondent à ce dont on s'attendait : dans tous les cas, les *miss* d'accès au cache L1 se situe à plus de 97% dans le chiffrement, et la plus grande partie dans la plus interne de la multiplication matricielle. Et comme vu à la figure 3, l'implémentation ligne * ligne permet d'augmenter fortement le taux de *hit* (pour une clé de 128).

3.3 Cycles et *stalls*

En observant la statistique `cycle_activity.stalls_total`, on peut aussi se rendre compte que malgré les avantages cités ci-dessus, on observe que le `loop unrolling` (implémenté dans les versions 2 et 4) a aussi pour effet d'augmenter le rapport "nombre de cycles en *stall*" sur "nombre de cycles". Ceci est visible à la figure 6 et s'explique comme ceci : puisque la boucle ne doit pas vérifier la condition sur la variable de boucle à chaque fois, l'enchaînement entre l'incrémentation de la variable et son utilisation est directe, sans instruction de type *compare*, entraînant donc l'attente du résultat de l'incrémentation.



(a) Pourcentage de cycles en état de *stall* pour une clé de 8



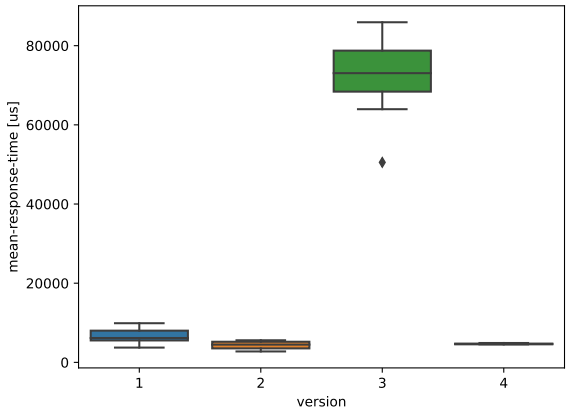
(b) Pourcentage de cycles en état de *stall* pour une clé de 128

FIGURE 6

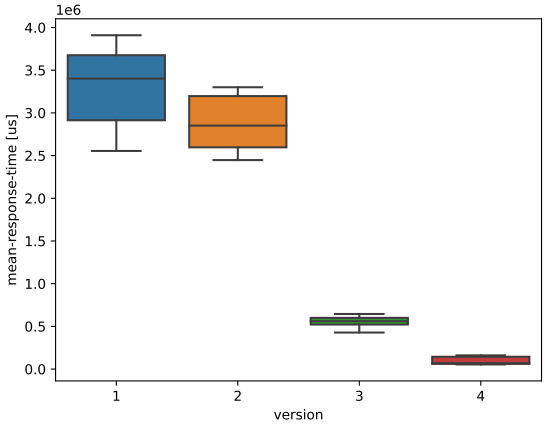
Cela a donc pour effet de réduire le nombre moyen d'instructions terminées par cycle, mais étant donné que le nombre d'instructions total est fortement diminué par cette méthode de *loop unrolling*, la méthode reste bénéfique.

3.4 Temps de réponse moyen

À l'aide de notre client, nous avons pu calculer le temps de réponse moyen, pour les 4 versions et pour des clés de 8 et de 128. Voici les résultats :



(a) Temps de réponse moyen pour une clé de 8



(b) Temps de réponse moyen pour une clé de 128

Pour une clé de 128, ces résultats confirment les observations précédentes, avec un effet positif mais modéré d'un *loop unrolling* et d'un effet bénéfique très marqué de la multiplication ligne * ligne. Nous n'avons pas repéré d'incompatibilité entre ces deux méthodes, ce qui se confirme dans les résultats avec une version 4 plus rapide que les 3 autres.

Pour une clé de 8, ce même constat est de vigueur, sauf pour la version 3. Nous avons tenté de repérer quelle pouvait être la cause de cette mauvaise performance en terme de temps de réponse moyen pour la version

implémentant la multiplication ligne * ligne sans *loop unrolling*, mais nous n'avons pas su faire de lien logique avec l'implémentation [1].

4 Autres optimisations

Pour le concours et pour aller plus loin, nous avons effectué d'autres optimisations. Commençons dans un premier temps par énumérer les quelques améliorations mineures, avant de présenter l'autre multiplication matricielle que nous avons implémentée.

4.1 Améliorations mineures

Le code effectuant le chiffrement était jusqu'alors situé dans un autre fichier et exécuté à l'aide d'un appel à la fonction `encrypt_file()` [2]. Nous l'avons déplacé directement en place dans le code du serveur, ce qui évite donc des instructions de type `jump`, des changements d'environnement etc.

Nous avons également remplacé l'utilisation de la variable `file_size` par une constante. Nous avons aussi déterminé une constante de valeur $1024 * 1024 * 4$ afin d'éviter ce calcul à de nombreux endroits dans le code.

Comme nous allons le voir en détails juste après, la boucle qui est désormais la plus interne (et donc la plus répétée) ne fait plus `key_size` étapes mais bien `file_size` étapes (avec évidemment une boucle entière de gagnée, sinon cela n'aurait aucun intérêt).

Avec un *loop unrolling* de 8, cette boucle sera exécutée 128 fois, et donc nous avons remplacé la condition `"j < file_size"` par `"__builtin_expect(j < file_size,1)"` afin d'informer le compilateur que dès le premier tour de boucle, il est préférable de prévoir de la continuer.

De cette manière, nous forçons l'instruction de type `likelyJump` et nous nous assurons de ne pas mal anticiper la suite du programme à cet endroit (évitant ainsi des *stalls*).

Pour terminer sur les améliorations mineures, nous avons aussi stocker les quelques variables utilisées dans les boucles comme `hline`, `vline`, etc, dans une structure. Accompagnée de la commande `#pragma pack(push, 16)`, cela permet d'assurer la localité spatiale en mémoire, ce qui est utile étant donnée la localité temporelle de ces variables dans notre code.

4.2 Multiplication matricielle

Nous avons amélioré la multiplication ligne * ligne implémentée dans notre version 4. En effet, nous avons remarqué que la boucle la plus interne utilisait une valeur fixe dans la clé, et une ligne du fichier de base pour remplir une ligne du fichier chiffré (lignes de taille `key_size`).

En chargeant une ligne de taille `key_size` du fichier, nous chargeons en réalité tout le bloc mémoire et donc ce qui est amené en cache est une ligne bien plus longue. Avec l'implémentation de la version 4, nous ne tirions pas profit de cette ligne déjà en cache.

Dans la nouvelle version, nous traversons cette fois toute la ligne du fichier de base pour écrire toute la ligne du fichier chiffré en lien avec cette même valeur fixe de la clé. Ce qui permet de profiter du cache d'avantage.

5 Commentaires finaux

Les performances générale de l'algorithme ont augmentée de manière radicale entre la version non optimisée et la version optimisée, qui atteint un score d'environ 280 sur la tâche INGInious (il est possible que nos scores ne soient pas si élevés au moment de la soumission finale car seule la dernière version est sauvegardée, et le score semble avoir un petit facteur aléatoire dans certains cas).

1. Ces mesures ont été prises de nombreuses fois, sur des machines différentes, et ont été à chaque fois identiques.
2. Vous pouvez d'ailleurs le voir sur les figures 4 et .

Nous n'arrivons toujours pas a comprendre les effets négatif exposés dans la section 3.4, vis-à-vis de la version 3 en ligne * ligne, pour une clé de taille 8, qui, instinctivement, devrait être au moins aussi bonne que la version 1 en ligne * colonne.

6 Conclusion

La version 4 de notre algorithme a montré les avantages de l'utilisation du cache et *loop unrolling*, et la version 5, c'est-à-dire celle composée de toutes les améliorations présentées à la section 4 est encore plus performante (de manière moins significative que la version 4 ne l'est par rapport à la version 1).

Avec les quatres versions présentées en début de rapport, nous montrons bien les effets de l'adaptation de l'algorithme de multiplication matricielle et du `loop unrolling`. Nous pensons que les résultats, excepté pour le temps de réponse moyen de la version 3 avec une clé de taille 8, montrent bien la progression attendue lors de l'utilisation de méthodes d'optimisation du cache, du nombre d'instruction, du nombre de cycles, etc.