Université Claude Bernard Lyon 1

.

Projet d'ouverture à la recherche

–

Décroissance numérique : quels écosystèmes logiciels pour l'informatique frugale ?

–

Rapport

PETIT Lucas, STAVIS Thomas

Encadré par : MOY Matthieu, SALAGNAC Guillaume

Mots clés : Durabilité, Sobriété numérique, Langages de programmation, Compilation, Informatique minimaliste, Mémoire, Linux

Dépot Github : github.com/ljsp/POR_Decroissance_Numerique

Master 1 Informatique

Université Claude Bernard Lyon 1 – Site la Doua

6 juin 2023

Remerciements

Nous remercions nos encadrants, Matthieu Moy et Guillaume Salagnac, pour leur suivi très régulier, leurs enseignements très utiles, et les discussions très intéressantes sur l'informatique, la frugalité, ainsi que le monde de la recherche, tout au long de l'année. Nous avons pu, grâce à eux et à ce projet, développer notre conscience écologique ainsi que notre savoir en informatique.

Résumé

L'extraction de ressources, leur exploitation, leur consommation, ont une empreinte carbone qui augmente les effets du dérèglement climatique. De plus, la production de matières nécessaires aux industries ne peut continuer à augmenter, ni même à être constante, à long terme, compte tenu du stock limité que notre planète peut nous fournir. Comment continuer un minimum ces activités dans un scénario où les ressources sont épuisées ?

C'est en informatique, plus spécialement, que nous nous posons cette question : quels écosystèmes logiciels pour l'informatique frugale ? Cette question nécessite une étude très poussée de nos systèmes numériques. C'est pour cela, dans le cadre de notre projet d'ouverture à la recherche, que nous faisons le choix de ne nous intéresser qu'à l'utilisation de la mémoire au quotidien, dans des terminaux d'utilisateurs et de développeurs moyens.

Après avoir défini ce qu'est la mémoire physique, rappelé comment fonctionne un système d'exploitation tel que Linux, et défini que c'est le pic que mémoire qui nous intéresse lors de la compilation, l'exécution, et l'interprétation d'une application dans différents langages, nous développons alors des outils pour mesurer ces pics, utilisant les principes de conteneurisation, d'échantillonnage et d'interception d'appels systèmes, avant de les valider.

Des premières expériences sont alors réalisées pour définir dans les grandes lignes le comportement de ce pic de RAM pour différents types de programmes selon les langages, et exposer l'écart de consommation entre certaines applications ayant pourtant la même fonctionnalité de base.

Table des matières

1 Introduction

1.1 Contexte

Les émissions de kgCO2eq, appelées régulièrement empreintes carbones, causées par l'extraction de matières premières, la fabrication et l'utilisation d'objets complexes, ainsi que leur traitement en fin de vie, traduisent une augmentation de la concentration en gaz à effet de serre (GES) de l'atmosphère, qui est un des principaux vecteurs d'accroissement du réchauffement climatique et de toutes les conséquences qui lui sont associées. De plus, la Terre comporte une quantité de matières premières limitée. Il est donc difficile d'imaginer une exploitation à un rythme aussi effrénée à long terme.

C'est dans ce contexte de régulation des émissions de GES et de limitation de l'exploitation de ressources, pour éviter ou diminuer les conséquences néfastes que cela engendre, que nous nous penchons sur l'impact du numérique. En effet, selon The Shift Project, en 2019, le numérique est responsable de 4 % des émissions mondiales de GES, et cela continue de croître de 2 % an. Cela est en partie dû à la complexité des objets numériques : par exemple, la fabrication et l'utilisation d'un smartphone de 150 g émettent 85 kgCO2eq. Or, le numérique est un vaste champ dans lequel un très grand nombre de facteurs se distribuent les émissions, ce qui ne rend pas la tâche aisée.

1.2 Objectifs

Le but général est donc de chercher une manière de diminuer l'impact écologique du numérique tout en limitant l'effet rebond. L'idée est de cerner la machine avec la plus petite configuration possible pour laquelle des activités numériques restent réalisables, ce qui bornera les spécifications, la taille et les performances des composants de cette machine, et donc la quantité de matières premières nécessaires à sa construction, ce qui réduira son empreinte carbone. Nous cherchons alors comment programmer la partie logicielle de cette machine, pour en déduire les configurations minimales. Cela implique d'étudier les langages de programmation et les environnements de production pouvant coder ces logiciels.

Dans ce projet, en particulier, nous allons décortiquer le sujet, étudier la littérature du domaine, définir quels outils sont nécessaires, les développer et les valider, ainsi que réaliser des expériences préliminaires pour s'assurer du sens de notre démarche.

1.3 Partis pris

Les machines numériques sont fréquemment divisées en 3 parties, terminaux, centres, et réseaux, chacune étant divisée en 2 sous-parties, fabrication et utilisation. Notre approche, se penchant surtout vers un informatique grand public et quotidien, a contrario de l'embarqué ou du logiciel d'infrastructure, nous nous intéressons à la partie utilisation de terminaux, par des particuliers et développeurs.

Dans ce cadre, le facteur limitant le plus évident pour pouvoir exécuter un programme est la quantité de mémoire vive, la Random Access Memory (RAM). Or, la quantité indispensable de RAM pour exécuter un programme réalisant une même tâche peut varier en fonction du langage de programmation. Nous cherchons donc un moyen de mesurer le pic de RAM utilisé par

des programmes type, dans des environnements de développement différents, et pour une liste non-exhaustive de langages, en les compilant, les exécutant, les interprétant.

Pour effectuer ces mesures, des outils d'estimations du pic de RAM d'un processus sont donc nécessaires. L'environnement impactant le développement et les mesures, nous avons décidé d'utiliser le système d'exploitation Linux, car il est open source, qu'il met à disposition des application pratiques, et que son fonctionnement détaillé est plus facile à observer.

Il s'est également posé la question de faire fonctionner nos outils sur une machine virtuelle ou sur le matériel physique. Même si cela réduit leur reproductibilité, c'est la deuxième option qui a été choisie, car plus proche de nos cas d'utilisation, mais elle a nécessité un dual-boot sur nos machines.

2 Prérequis

Vocabulaire

Langages compilés : comme C et C++, ce sont des langages pour lesquels un compilateur traduit le code source en code machine exécutable. Cette compilation offre de meilleures performances lors de l'exécution du programme.

Langages interprétés : comme Python et JavaScript, à l'opposé des langages compilés, ces langages sont exécutés, ligne par ligne, par un interpréteur lors de l'exécution, pouvant rendre ces langages moins performants.

Plugins : ce sont des extensions logicielles qui augmentent les capacités d'un programme existant.

Bibliothèque : une collection de fonctionnalités prédéfinies qui facilitent la réutilisation du code et favorisent l'efficacité du développement, en évitant la redondance d'écriture de code.

La mémoire dans un système d'exploitation

Dans un système d'exploitation comme Linux, les programmes utilisent des fonctions spécifiques pour gérer leur mémoire. Par exemple, "malloc" en C ou "new" en C++ sont utilisés pour allouer de la mémoire. Mais lorsque l'espace déjà alloué est épuisé, ces fonctions font appel au noyau du système d'exploitation, via des appels systèmes comme "mmap" ou "brk", pour obtenir plus de mémoire.

Le noyau réserve alors ce qu'on appelle de la "mémoire virtuelle". Il s'agit d'un engagement d'allouer de l'espace mémoire, mais l'espace n'est réellement réservé que lorsque le programme tente d'y accéder, en lecture ou en écriture. Cette réservation se fait par blocs de 4 Kio, connus sous le nom de "pages".

De plus, le noyau Linux autorise ce qu'on appelle l'overcommit de la mémoire. Cela signifie qu'il peut promettre d'allouer plus de mémoire virtuelle que la mémoire RAM réellement disponible. C'est un moyen d'utiliser la mémoire plus efficacement, mais cela peut aussi causer des problèmes si la mémoire promise ne peut pas être fournie. C'est la Memory Management Unit (MMU) qui se charge de traduire les adresses virtuelles en adresses physiques. En outre, le noyau décide

quelles pages mémoire sont chargées en RAM et lesquelles sont envoyées en mémoire "swap" (sur le disque dur).

Il est aussi important de comprendre deux mesures clé de l'utilisation de la mémoire : la taille de l'ensemble virtuel (Virtual Set Size, VSS) et la taille de l'ensemble résident (Resident Set Size, RSS). La VSS est la taille totale de la mémoire virtuelle qu'un processus peut utiliser. Grâce à l'adressage sur 64 bits, cette taille est presque infinie. D'autre part, la RSS est la taille de la mémoire qui est réellement utilisée en RAM. Même avec un adressage sur 64 bits, cette taille est limitée par la quantité de mémoire RAM disponible.

Les processus enfants

Si on observe le fonctionnement de certains programmes, leur mesure peut s'avérer ardue s'ils créent des processus enfants. Ces processus sont des instances de programme séparées qui sont lancées par le processus parent, principal, et qui peuvent exécuter des tâches en parallèle ou indépendamment du processus parent. Ils consomment aussi des ressources système, notamment de la RAM et du temps processeur. Par conséquent, lors de l'évaluation de la consommation totale d'un programme, il est essentiel de prendre en compte non seulement les ressources utilisées par le processus principal, mais aussi celles consommées par tous ses processus enfants. Cela a donc nécessité une attention particulière pour développer nos outils de mesures.

La conteneurisation

C'est une méthode de virtualisation au niveau du système d'exploitation qui permet d'exécuter une application, ses dépendances, ainsi que les threads et sous-processus lancés par celle-ci, dans un environnement autonome appelé "conteneur". Il est souvent possible de contrôler les ressources de cet environnement, comme la RAM ou le pourcentage de CPU utilisés. Contrairement aux machines virtuelles, qui nécessitent leur propre système d'exploitation complet, les conteneurs partagent le système d'exploitation de la machine hôte, ce qui les rend plus légers et plus rapides à démarrer.

Développement

Le développement et les tests se sont fait sur une machine avec un processeur Intel Core i7-856U ou Intel Core i5-1035G1, avec 8Gb de RAM et sur Ubuntu 22.04.2 LTS.

Pour les langages utilisés voici la liste de leur version : Java openjdk 17.0.5, GCC et G++ 11.3.0, Clang et Clang++ 14.0.0, Python 3.10.6 et rust 1.69.0.

3 Études bibliographiques

3.1 An Empirical Comparison of 7 Programming Languages

Cette étude des années 80 compare sept langages de programmation : C, C++, Java, Perl, Python, Rexx et Tcl. Pour cela, ils utilisent un même programme à implémenter pour chaque

langage, ce qui permet une comparaison équitable. Il en ressort :

— Écrire un programme en Perl, Python, Rexx ou Tcl prend généralement moins de temps que d'écrire le même programme en C, C++ ou Java. De plus, le programme final est généralement deux fois plus court.

— Il n'y a pas de différence notable en termes de fiabilité entre les différents langages.

— Les programmes écrits en langages de script utilisent environ deux fois plus de mémoire que ceux écrits en C ou C++. Les programmes Java, quant à eux, utilisent trois à quatre fois plus de mémoire que les programmes C ou C++.

— Les programmes C et C++ sont plus rapides pour la phase d'initialisation du programme, qui consiste à lire un fichier de dictionnaire de 1 Mo et à créer une structure de données interne de 70K entrées. Ils sont trois à quatre fois plus rapides que les programmes Java et cinq à dix fois plus rapides que les langages de script.

— Pour la phase principale du programme, les programmes C et C++ sont seulement deux fois plus rapides que Java. Les programmes de script ont également tendance à être plus rapides que Java.

Cet article nous a donc aiguillés sur les langages les plus intéressants en termes de performances, à savoir C et C++, mais il possède quelques lacunes comme les données utilisées. En effet, pour chaque langage, il n'y pas le même nombre d'implémentations et les niveaux entre les développeurs est également variable ce qui pourrait nuire aux résultats.

3.2 A Comparative Study of Programming Languages in Rosetta Code

Cette étude, datant de 2015, propose de répondre à la question : quel est le meilleur langage de programmation ? La question étant trop floue et la réponse évidemment subjective, elle propose quand même de comparer qualitativement 8 langages de programmation, choisis en fonction de leur popularité et de leur paradigme de programmation, sur 750 programmes différents provenant de la base de code open source Rosetta. Des mesures de ces langages en fonction de critères éclectiques, dont l'utilisation de la mémoire, sont alors réalisées.

Les résultats intéressants pour nous se trouvent dans la partie RQ4, qui répond à la question suivante : quel langage utilise la mémoire le + efficacement ? Après avoir préalablement vérifié qu'utiliser la mémoire le plus efficacement signifie bien de mesurer le pic de RSS lors de l'exécution d'un programme, des points en lien avec notre projet ont été retenus :

— Leur figure 13 montrant le pic minimum de RAM utilisé par les programmes nous indique sa quantité minimale nécessaire à faire tourner un programme. On s'attendait donc à retrouver des résultats similaires, et ce fut le cas.

— Pour des langages ayant le même paradigme, la différence de RAM utilisée est faible. C'est aussi le cas dans certaines de nos mesures, cependant le Rust déroge à la règle, et nous n'avons pas pu expérimenter cet effet pour autant de langages.

— La moyenne de la RAM pour chaque langage et programme est très faible comparée au pic. Calculer le pic a donc bien plus de sens que de calculer la moyenne. Cependant, cela limite les outils de calcul utilisant de l'échantillonnage.

Enfin, certaines autres conclusions sont intéressantes, mais ne nous intéressent pas particulièrement dans le cadre de notre projet d'ouverture à la recherche. Elles peuvent être cependant très utiles dans le projet dans sa globalité, de même que dans la question de l'informatique frugale.

4 Contributions expérimentales

4.1 Développement

4.1.1 Échantillonnage : memTool

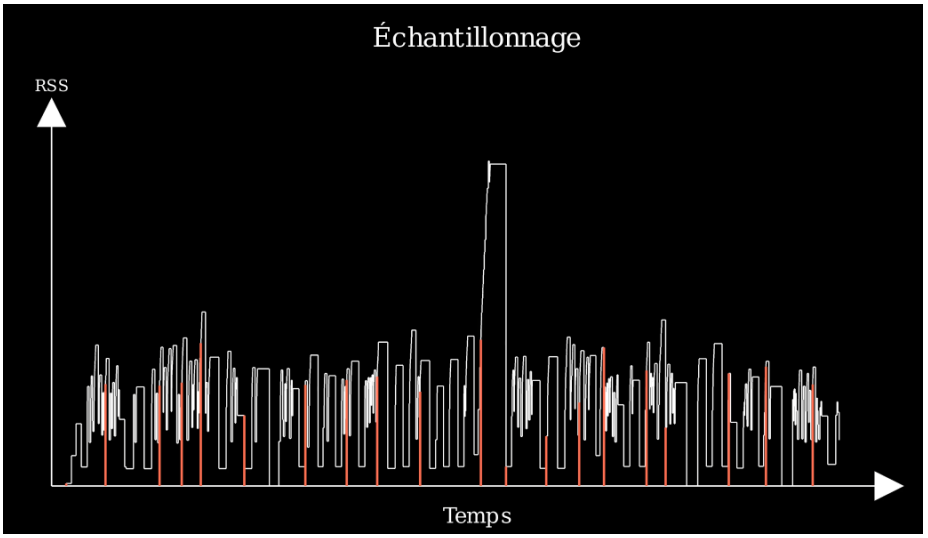


FIGURE 1 – Représentation de l'échantillonnage au cours du temps

Principe

L'échantillonnage consiste à interroger le noyau Linux à intervalles réguliers pour lui demander quelle quantité de mémoire est utilisée, et on stocke toutes les valeurs observées au cours de l'exécution du programme. Le problème que l'on peut avoir est lorsqu'un pic se produit entre deux échantillons. Pour avoir un résultat satisfaisant, il faudra qu'on échantillonne suffisamment rapidement pour avoir des mesures satisfaisantes.

Implémentation

Algorithm 1 Pseudo code pour memTool

```
0: procedure memTool
0:   procedure measure_memory(pid)
     Obtenir les informations sur le processus par son PID
     Récupérer VSS et RSS du processus
     Pour chaque processus enfant du processus principal, récupérer et additionner VSS et RSS
     Si le processus ou l'un de ses enfants n'existe plus, enregistrer l'événement et retourner 0
     pour VSS et RSS
       end procedure
5: Lancer le programme à surveiller
   while le programme est en cours d'exécution do
     Appeler measure_ memory avec le PID du programme
     Enregistrer l'utilisation maximale de la mémoire
     Enregistrer l'utilisation actuelle de la mémoire et le temps associé
10:  Vérifier si le programme a terminé
   end while
   Afficher les pics d'utilisation de la mémoire et le nombre d'échantillons pris
   Afficher les données recueillies sous forme de graphiques
   end procedure=0
```

Le script fonctionne comme suit, d'abord, nous importons les modules nécessaires pour l'exécution du script, qui incluent sys, pour récupérer les informations du système d'exploitation, psutil, une bibliothèque pour récupérer les informations de consommation de mémoire, matplotlib.pyplot pour la création des graphiques et pandas pour la manipulation des données.

Nous avons créé la fonction measure_memory qui est l'une des parties essentielles de notre script memTool. Son rôle et d'enregistrer l'utilisation de la mémoire par le programme et ses processus enfants. Pour cela, elle prend un identifiant de processus (PID) comme argument et avec les fonctions memory_info et children de psutil elle récupère les données de consommation VSS et RSS pour le processus et ses enfants.

Nous initialisons ensuite plusieurs variables pour stocker les valeurs maximales d'utilisation de la mémoire et les données pour les deux types de mémoire que nous suivons et lançons le programme à surveiller en tant que sous-processus. Cela nous permet de savoir s'il a terminé ou pas, et nous mesurons ensuite régulièrement son utilisation de la mémoire jusqu'à ce qu'il se termine. Nous stockons également le pic d'utilisation de la mémoire à chaque fois que nous mesurons.

Finalement, nous affichons les informations recueillies sous forme de graphiques en utilisant le module matplotlib.pyplot, puis nous terminons le script avec le même code de retour que le programme surveillé pour nous assurer du bon fonctionnement.

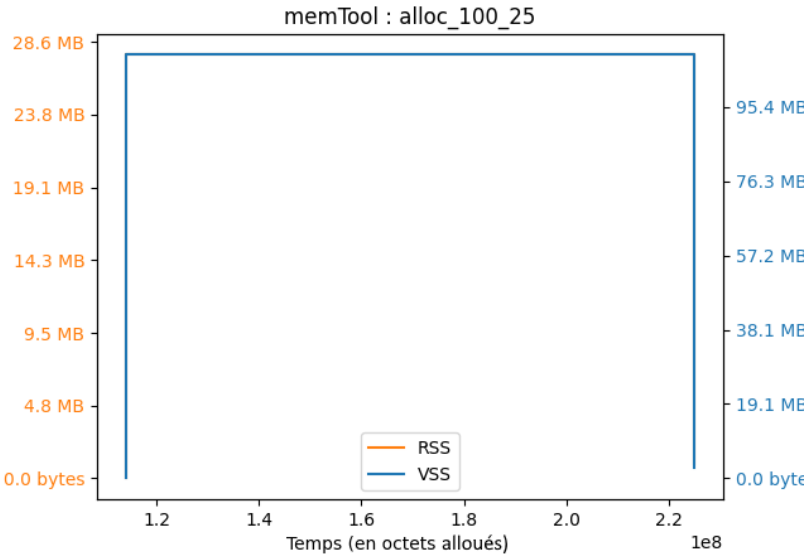


FIGURE 2 – Exemple de graphique généré par memTool

4.1.2 Interception d'appels systèmes : logReader

Principe

Lors de son fonctionnement, un processus va faire des appels systèmes pour diverses raisons, que ce soit pour créer un processus enfant avec fork, lire un fichier avec read ou pour, ce qui nous interresse dans notre cas, des appels liées à la mémoire avec mmap, munmap, brk et sbrk. Ainsi, si on est capable de récupérer tous ces appels systèmes, on pourrait avoir une idée de la consommation mémoire d'un processus après les avoir traités, et il existe justement une commande pour récupérer ces appels sur Linux appelé strace.

De plus, strace possède l'option -ff ou –follow-forks permettant également de prendre en compte les processus enfants crée. Le seul désavantage est que la mémoire observée correspond à la mémoire virtuelle.

```
6256 brk(NULL) = 0x5635234e8000
6256 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
    ↪ -1, 0) = 0x7fcf37ffd000
6256 mmap(NULL, 83723, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fcf37fe8000
6256 mmap(NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
    ↪ x7fcf37c00000
6256 mmap(0x7fcf37c28000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|
    ↪ MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7fcf37c28000
6256 mmap(0x7fcf37dbd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|
    ↪ MAP_DENYWRITE, 3, 0x1bd000) = 0x7fcf37dbd000
6256 mmap(0x7fcf37e15000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|
    ↪ MAP_FIXED|MAP_DENYWRITE, 3, 0x214000) = 0x7fcf37e15000
6256 mmap(0x7fcf37e1b000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|
    ↪ MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fcf37e1b000
```

```
6256 mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
    ↪ -1, 0) = 0x7fcf37fe5000
6256 mprotect(0x7fcf37e15000, 16384, PROT_READ) = 0
6256 mprotect(0x563523062000, 4096, PROT_READ) = 0
6256 mprotect(0x7fcf38037000, 8192, PROT_READ) = 0
6256 munmap(0x7fcf37fe8000, 83723) = 0
6256 brk(NULL) = 0x5635234e8000
6256 brk(0x563523509000) = 0x563523509000
6256 +++ exited with 0 +++
```

FIGURE 3 – Exemple de fichier trace en filtant les appels liés à la mémoire

Implémentation

Algorithm 2 Pseudo Code de logReader

```
    Ouvrir fichier log
    for chaque ligne du fichier do
      if "mmap" dans ligne then
        Augmenter memoire actuelle
5:    else if "munmap" dans ligne then
        Diminuer memoire actuelle
      else if "brk" dans ligne then
        Ajuster memoire actuelle
      end if
10:   if memoire actuelle > memoire max then
        Mise a jour memoire max
      end if
      Enregistrer memoire actuelle, taille alloc
    end for
15: Afficher graphique =0
```

En s'appuyant sur strace nous avons donc écrit en Python le script logReader pour analyser et crée la courbe de l'évolution de la mémoire RSS à partir du fichier de log généré. Pour ce faire, le script exploite également les bibliothèques Matplotlib et Pandas pour les mêmes raisons que le programme d'échantillonnage.

Au début du script, nous avons importé les modules nécessaires. Nous avons besoin de 're' pour les expressions régulières qui sont utilisés pour la capture des valeurs dans les appels système, de sys, de Pandas et Matplotlib. De plus, nous avons importé un sous-module de Matplotlib, 'ticker', qui nous aide à formater les valeurs de l'axe des ordonnées de notre graphique.

Le script utilise un fichier de trace comme entrée, passé en argument lors de l'exécution du script. Nous extrayons le nom du programme à partir du chemin du fichier de trace, puis nous ouvrons ce dernier pour lire chaque ligne, chacune d'elles étant analysée pour déterminer l'utilisation de la mémoire.
Dans l'analyse de ces lignes, nous recherchons des appels spécifiques tels que **mmap**, **munmap**, **sbrk**, et **brk** qui indiquent une allocation ou une désallocation de mémoire. À chaque fois que

l'on rencontre un de ces appels, on calcule la valeur de l'allocation/désallocation, on met à jour l'utilisation actuelle de la mémoire (current_vss), et on vérifie si elle est supérieure à l'utilisation maximale de la mémoire (max_vss) enregistrée jusqu'à présent.

Chaque fois qu'une allocation ou désallocation de mémoire est détectée, le script fait avancer le temps de la taille de l'allocation/désallocation. Ces données seront ensuite utilisées pour générer le graphique montrant l'évolution de l'utilisation de la mémoire dans le temps.

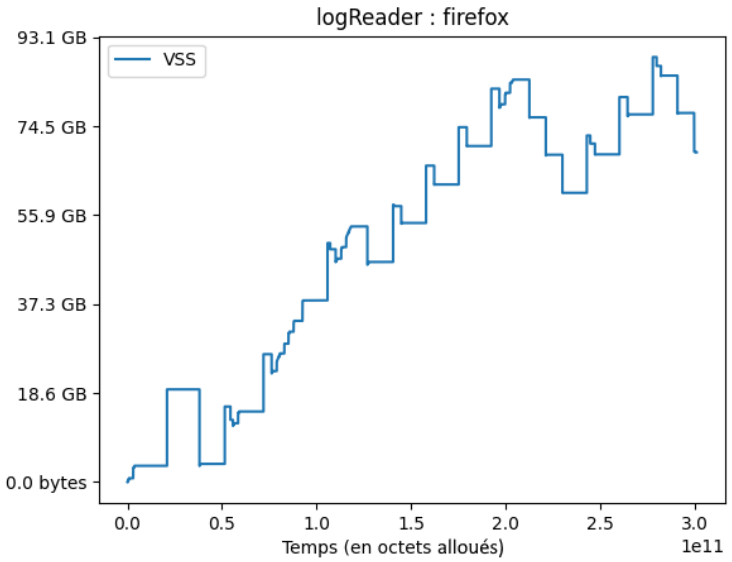


FIGURE 4 – Exemple de graphique généré par logReader

4.1.3 Conteneurisation : cgroup

Principe

Certains logiciels de conteneurisation offrent la possibilité de suivre la quantité de ressources allouées durant l'exécution, notamment la RSS. En exécutant alors une application dans un conteneur doté de cette capacité, on peut lire le pic de mémoire consommée. C'est ce que cgroup (Control Group V2), une fonctionnalité de Linux, permet de réaliser.

Implémentation

L'algorithme de calcul du pic est le suivant :

Algorithm 3 Utilisation de cgroup V2

```
0: procedure CgroupV2(commande)
   Créer un conteneur cgroup
   valeur de mémoire swap max utilisable <- 0
   Exécuter la commande dans le conteneur
   pic <- lire la valeur du pic de mémoire
5: Supprimer le conteneur
   Retourner pic
   end procedure=0
```

Deux variantes de cet algorithme, ont été développées. La première renvoie le pic pour commande unique (voir ci-dessus). La deuxième réalise la même chose pour une commande comportant des paramètres entiers constants ou variants, en demandant un min, un max et un pas, et renvoit le tableau des pics de mémoire en fonction des arguments.

Dans le programme principal, différentes options ont été ajoutées : celle d'afficher l'output de chaque commande, celle de renvoyer la moyenne de plusieurs mesures au lieu d'une unique, et celle d'exécuter préalablement une commande avant celle à mesurer.

4.1.4 Validation de nos outils

Afin d'obtenir des résultats valides, il faut que nos outils fonctionnent pour sûr, et donc réaliser des programmes qui allouent une quantité de mémoire connue afin de nous assurer que nos valeurs soient correctes. Le programme en C++ suivant a été utilisé pour ces tests :

```
#include <iostream>
int main() {
    char* alloc = (char*) malloc(100 * 1024 * 1024);
    for(int i = 0; i < 25 * 1024 * 1024 ; i++) {
        alloc[i] = 'a';
    }
    free(alloc);
    return 0;
}
```

Ce programme réalise en premier lieu une allocation 100 Mo, puis écrit sur 25 Mo, puis libère la mémoire.

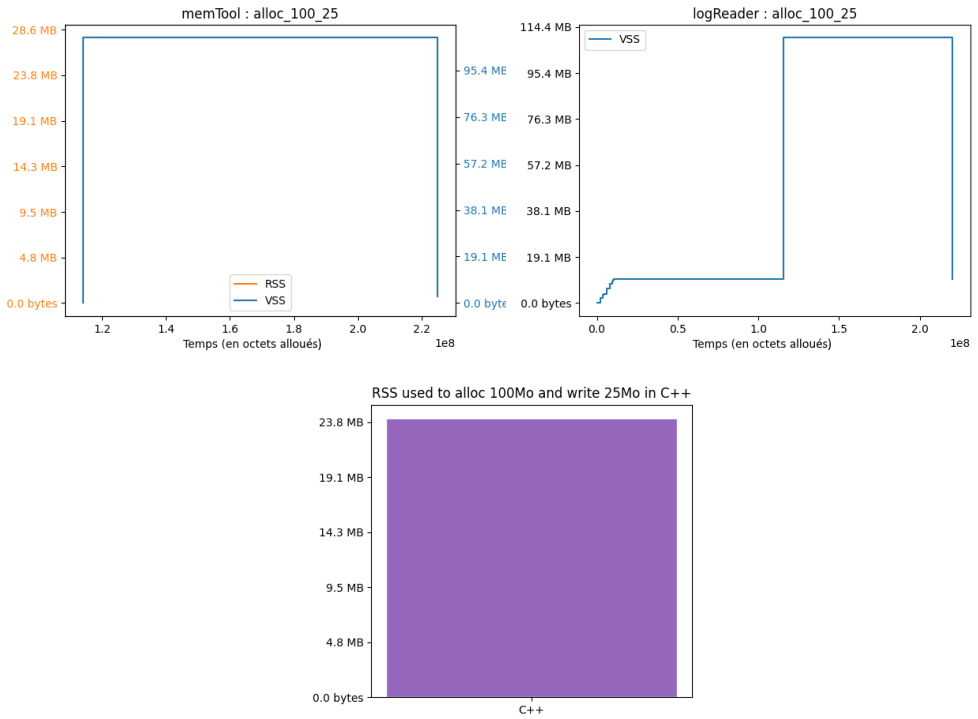


FIGURE 5 – Mesures de la mémoire avec memTool (à gauche), logReader (à droite), cgroup (en bas)

On remarque que memTool et cgroup ont une même estimation de la RSS, autour de 25 Mo. Aussi, memTool et logReader ont tous deux une estimation de la VSS autour de 100 Mo. Comme ces valeurs sont très proches de celles connues, compte tenu d'un coût d'exécution de base. Nous pouvons déduire que les estimations renvoyées sont les quantités connues rentrées en paramètre, et donc que nos outils de mesure sont fiables.

4.2 Premières expérimentations et résultats préliminaires

Dans les graphes suivant, "MB" signifie "mega bytes", et non "mega bits". Nous parlerons donc de Mo et MB en désignant la même chose.

4.2.1 HelloWorld

Il est important de connaître la quantité de mémoire nécessaire pour exécuter un programme "vide" pour ajuster les futures mesures. Pour cela, le programme universel "Helloworld" s'impose. Le profil mémoire pour certains langages est disponible en annexe.

4.2.1.1 Compilation générale

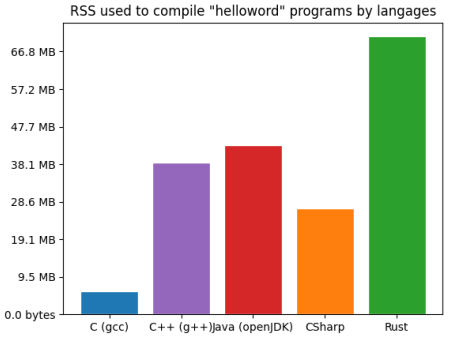


FIGURE 6 – Compilation de Helloworld

Ces résultats donnent les bornes minimales de compilation pour ces 5 langages. Elles varient de 6 Mo (C) à 74 Mo (Rust). Pour l'exécution, elles vont de 0,4 Ko (C) à 9,5 Mo (Java). Cela représente déjà un facteur 10 d'utilisation de la RSS.

4.2.1.2 Exécution générale

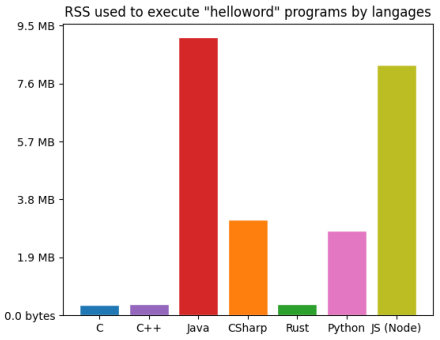


FIGURE 7 – Exécution de Helloworld

Ces résultats donnent les bornes minimales d'exécution pour ces 7 langages. Elles vont de 0,4 Ko (C) à 9,5 Mo (Java). Il y a ici une différence d'ordres de grandeur remarquable.

4.2.1.3 GCC / Clang

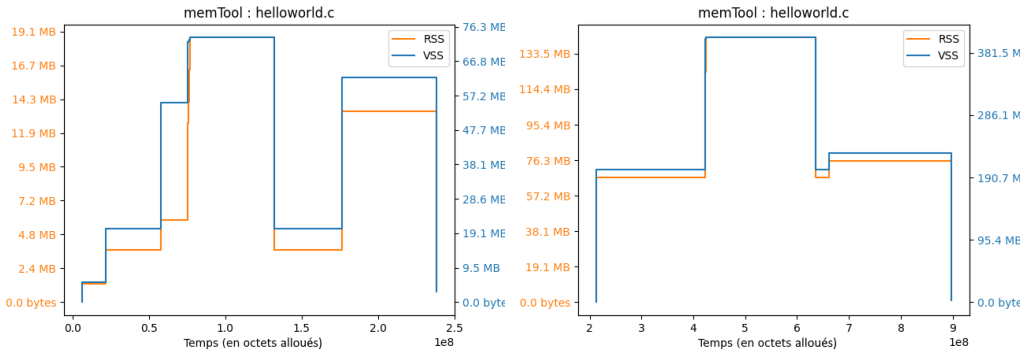


FIGURE 8 – Compilation de helloworld avec gcc (à gauche) et clang (à droite)

4.2.1.4 G++ / Clang++

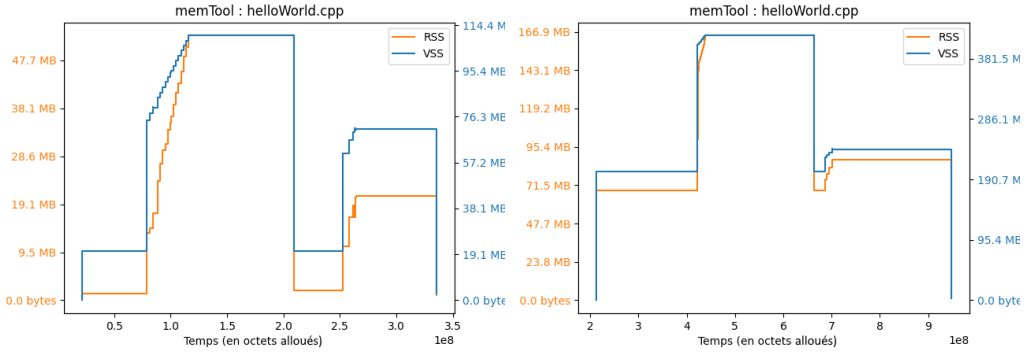
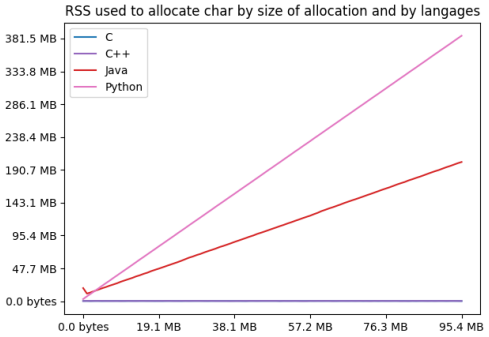


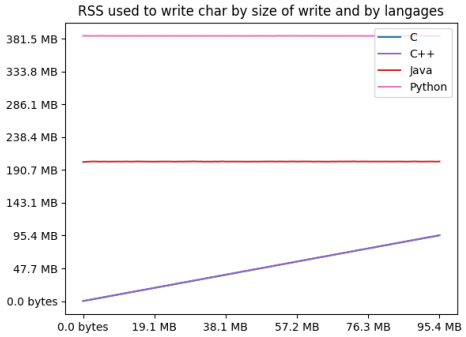
FIGURE 9 – Compilation de helloworld avec g++ (à gauche) et clang++ (à droite)

4.2.2 Allocations et écritures multiples

Nous pouvons maintenant voir avec précision comment se comporte le pic de mémoire lorsque des allocations et des écritures de tailles différentes sont réalisées. Cela va nous permettre de définir dans les grandes lignes de comportement de chaque application mesurée.



Cette première figure a été réalisée en faisant pour chaque langage une allocation allant de 0 à 100 Mo, sans écriture. On remarque que le C et le C++ ont un pic constant, d'environ 0,4 Mo. Tandis que Java et Python ont un pic qui varie linéairement en fonction de l'allocation, avec une pente d'environ 2 pour Java et d'environ 4 pour Python.



Cette deuxième figure a été réalisée en faisant une allocation de 100 Mo et une écriture allant de 0 à 100 Mo. Les pics de Python sont constants et culminent à environ 400 Mo, et ceux de Java sont aussi constant et d'environ 200 Mo. Cela représente respectivement 4 et 2 fois la quantité de mémoire écrire par le programme, ce qui est en accord avec nos mesures. Les pics du C et du C++ sont quant à eux linéaires et de pente environ 1.

4.2.3 Comparaison VS code

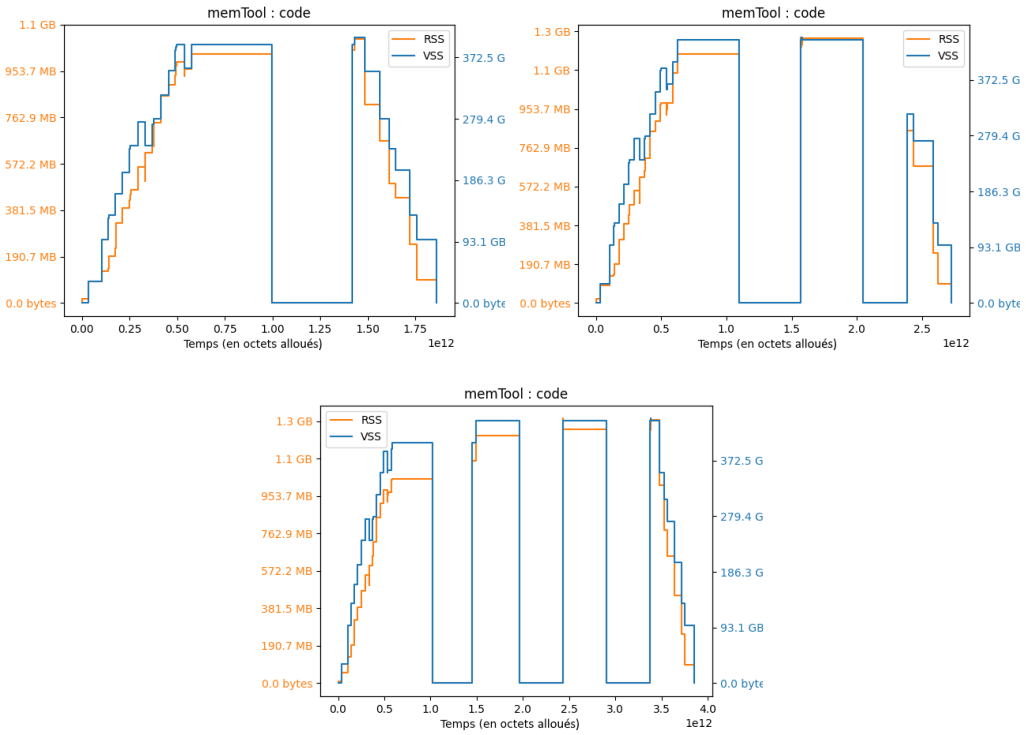


FIGURE 10 – Mesure de VS code avec 0, 15 et 25 plugins

4.3 Pistes explorées

Nous avons dans un premier temps effectué quelques recherches afin de voir les différents outils déjà existants. Sur github nous avons trouvé plusieurs dépôts comme ps_mem et bytehound qui se sont avérés assez limités leurs mesures ne donnant pas les informations exactes. Nous avons quand même trouvé un petit dépôt nommé memUsage qui, bien qu'imparfait car les mesures étaient inexactes, la logique de code inutilement complexe et faisait des fork non nécessaires, a servi de base pour notre outil appelé memTool appliquant le principe de l'échantillonnage.

Valgrind était une option envisagée, mais n'a pas été retenue car trop complexe et pas assez adapté à notre cas.

Linux possédant également des commandes de base nous nous sommes demandés si certaines d'entre elles pouvaient nous être utiles dans les mesures. Nous avons donc exploré les commandes top et time, mais ces dernières n'était pas assez précises et ne permettait pas de suivre les forks. Notre choix s'est donc porté sur strace pour les raisons détaillées dans la partie la concernant.

5 Gestion de projet

5.1 Outils

En termes de communication, nous avons utilisé principalement nos messageries électroniques, permettant d'échanger en dehors des réunions, ainsi que les salons de visioconférence fournis par nos encadrants. Nous disposions aussi d'un petit serveur Discord pour échanger certains documents et discuter du projet entre étudiants.
En termes de développement, nous nous sommes tournés vers GitHub, qui permet très facilement de développer et de partager du code entre 2-4 personnes, et vers Visual Studio Code, qui permet d'avoir facilement un environnement de développement adapté à plusieurs langages.
Nous n'avons cependant pas utilisé d'outil de gestion de projet (type diagramme de gantt, UML, etc) car notre projet n'étant pas un développement d'application et notre sujet étant trop large pour prévoir des jalons, cela ne nous a pas paru pertinent.

5.2 Rythme de travail

Dès le début du projet, nous nous sommes mis d'accord, encadrants et étudiants, pour réaliser une réunion par semaine, modulo les vacances et les emplois du temps chargés. Compte tenu du volume horaire de cette UE (à 3 crédits au début de l'année puis à 6 crédits à partir du 2nd semestre), nos réunions hebdomadaires duraient environ 1h. Cela nous a permis de pouvoir beaucoup discuter des chemins à prendre et de tenir régulièrement au courant nos encadrants sur l'avancée du projet.

5.3 Prévisions et avancement réel

Dans le cahier des charges rendu début novembre 2022, nous avions prévu d'avancer sur le projet en découpant notre temps en 2 parties de 3 mois : la première concernait le fait de développer des scripts de mesure du pic de mémoire et de les tester sur différents programmes (petits et moyens), et la deuxième concernait l'acquisition de mesure pour des programmes plus importants (du type toolchain), la rédaction du rapport, la réalisation d'une vidéo de vulgarisation et

l'approfondissement du sujet. Le tout devant être soutenu par une lecture d'article et de documentation constante.

En réalité, nous avions découpé notre temps différemment (les durées présentées sont approximatives). En premier lieu, nous avons, durant un mois et demi, étudier la littérature pour avoir une meilleure idée de ce que nous devons mesurer et pouvoir orienter nos choix concernant quelles techniques de mesures nous allions utiliser. Ensuite, durant un mois et demi, nous avons cherché des outils pour nous aider à développer nos scripts de mesure et nous avons commencé à les développer, ce qui nous amène à fin janvier. S'en est suivi un mois de développement, de tests, et de récolte des premiers résultats, puis de deux mois où nous avons continué à faire cel tout en faisant la vidéo et en mettant au propre notre code pour le rendre. Et enfin, nous finissons avec un mois d'écriture du rapport et 2 semaines de réalisation de la soutenance.

6 Conclusion

6.1 Résultats

Dans l'optique d'une société future frugale en ressources, le choix du langage de programmation est crucial. Au vu de nos résultats, le langage C, économe en mémoire, apparaît comme une option attrayante pour minimiser l'empreinte énergétique tout en assurant de bonnes performances, il possède en effet la plus petite consommation de mémoire que ce soit pour la compilation ou l'exécution. Viennent ensuite les langages C++ et Rust, qui bien que plus consommateurs pour la compilation, restent proche en termes d'exécution. Cependant, d'autres facteurs, comme l'expressivité et la facilité d'utilisation, doivent être pris en compte. Java et Python, malgré une consommation mémoire plus importante, offrent une grande flexibilité. En outre, une utilisation judicieuse des outils de codage, en limitant le nombre de plugins par exemple, permet d'optimiser la consommation de mémoire.

6.2 Travail futur

Il serait intéressant dans la suite de ce projet de réaliser une vérification poussée de nos outils, pour pouvoir ensuite comparer une quantité plus importante de programmes, plus représentatifs du quotidien d'un utilisateur. Ces programmes peuvent être des éditeurs de texte, des applications de mails, des lecteurs vidéo, des jeux vidéos, des navigateurs, etc. La piste du parallélisme dans les applications est aussi intéressante, que ce soit le multithreading, le multiprocessus, en réseau ou non, pour définir des tendances.

Pour la partie développeurs, mesurer la toolchain d'un langage serait un point fort de l'étude, car cela caractériserait à quel point un langage est maintenable et évoluable, suivant une certaine quantité de RAM donnée. Comparer différentes versions de langages en fonction de leurs dates de parution, leurs fonctionnalités, leur flags, servirait à définir ce qui consomme le plus de RAM pour chaque langage.

Annexes

Divers

cgroup

Les premières tentatives d'utilisation de cgroup ne furent pas fructueuses, car nous utilisions la 1ère version de cgroup (cgroup V1), qui ne permet pas de gérer les ressources ainsi que d'avoir accès à certaines informations de manière simple. Nous observions cependant un échec du conteneur lorsqu'une application dépassait une limite de mémoire préalablement définie. Ainsi, nous avions basé notre méthode de recherche du pic sur de la dichotomie : en rentrant une commande, une valeur de mémoire maximum, une valeur de mémoire minimum et une précision, nous pouvions accéder à l'ordre de grandeur du pic de mémoire de la commande. L'algorithme se passait de la manière suivante :

Algorithm 4 Utilisation de cgroup V1

```
0: procedure CgroupV1(commande, max, min, precision)
   if lancerCommande(commande, min) est un échec then
     retourner erreur
   end if
   while max - min > precision do
5:   current = (max - min) / 2
     if lancerCommande(commande, current) est un échec then
       max = current
     else
       min = current
10:  end if
   end while
   retourner min, max
   end procedure=0
```

Seulement, même si cela fonctionnait, cette manière de faire n'était pas du tout optimale, car il fallait relancer plusieurs fois la commande, ce qui peut prendre beaucoup de temps. De plus, nous ne pouvions pas savoir pourquoi le conteneur cessait vraiment de fonctionner, et la fonctionnalité de désactivation de l'utilisation de la mémoire swap (technique qui consiste à déplacer une partie de la RAM non-utile dans le disque dur pour avoir la place de réaliser certaines opérations, puis de la déplacer à nouveau lorsque les opérations ont fini d'être réalisées ou que cette partie de la RAM devient à nouveau utile) ne marchait pas.

C'est alors que nous sommes passés à la V2 de cgroup. Celle-ci est beaucoup plus facile à prendre en main, permet de facilement désactiver la mémoire swap, et surtout fournit de fichier comportant la valeur du pic de mémoire utilisé. De plus, elle contient aussi des fichiers décrivant quels mécanismes d'arrêts d'un processus ont été utilisés, ce qui a pu confirmer que le pic affiché était bien la mémoire résidente maximum utilisée durant toute la durée d'exécution de la commande. Ainsi, en plus du fait que l'ordre de grandeur du pic de mémoire acquis avec la méthode utilisant la V2 de cgroup était le même que celui acquis en utilisant la V1, l'utilisation de la conteneurisation a pu grandement être simplifiée.

Scripts

Lors de l'implémentation de nos outils en Python, nous avons utilisé plusieurs librairies. Voici les plus importantes avec leur utilité :

— matplotlib : permet de réaliser des graphs de données

— pandas et cvs : permettent respectivement de traiter des données et et de les formater

— subprocess et sys : permettent de créer des sous-processus et d'accéder à des opérations systèmes

Figures

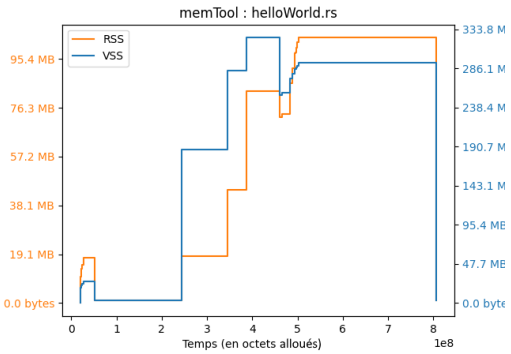


FIGURE 11 – Compilation de helloworld en Rust

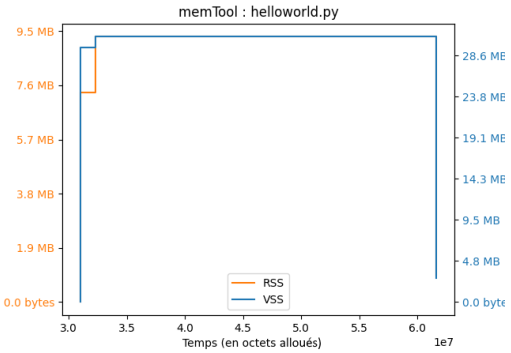


FIGURE 12 – Execution de hellowrold en Python

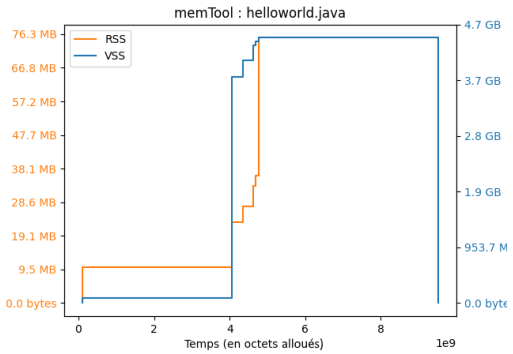


FIGURE 13 – Execution de hellowrold en Java

Références

Prechelt, L. (2000). An empirical comparison of seven programming languages. IEEE Computer, 33(10), 23-29. https://doi.org/10.1109/2.876288

Nanz, S., Furia, C. A. (2015). A comparative study of programming languages in Rosetta code. Dans arXiv (Cornell University) (Vol. 1, p. 778-788). Cornell University. https://doi.org/10.5555/2818754.2818848

Marquet, K., Combaz, J., Berthoud, F. (2019). Introduction aux impacts environnementaux du numérique. Bulletin 1024, 13, 85-97. https://doi.org/10.48556/sif.1024.13.85

Contributeurs aux projets Wikimedia. (2023). Équivalent CO2. fr.wikipedia.org. https://fr.wikipedia.org/wiki/%C3%89quivalent_CO2

Contributeurs aux projets Wikimedia. (2023b). Réchauffement climatique. fr.wikipedia.org. https://fr.wikipedia.org/wiki/R%C3%A9chauffement_climatique

The Shift Project. (2022, 19 juin). Accueil - The Shift Project. https://theshiftproject.org/

Renouard, L. (2022, 19 avril). Smartphone : 85 kg de CO2 émis en moyenne durant la première année d'usage. Les Numériques. https://www.lesnumeriques.com/telephone-portable/smartphone-85-kg-html

Contributeurs aux projets Wikimedia. (2022). Effet rebond (économie). fr.wikipedia.org. https://fr.wikipedia.org/wiki/Effet_rebond_(%C3%A9conomie)

Contributeurs aux projets Wikimedia. (2023a). Fork (programmation). fr.wikipedia.org. https://fr.wikipedia.org/wiki/Fork_(programmation)

Linux man pages online. (s.d.). https://www.man7.org/linux/man-pages/