Université de Strasbourg

Mémoire de Travail d'Étude et de Recherche

Sujet : Outils et métriques d'un mobile Android

HALM Florian

Mars 2024

UFR de mathématique et d'informatique
Université de Strasbourg

Sommaire

1 Mobile Android utilisé

Le mobile Android utilisé dans ce TER est :
- Samsung Galaxy S9 2018 (SM-G960F)
- 4GB de RAM
- CPU : Exynos 9 Octa 9810 à 2.70 GHz avec 8 coeurs
- Android 10 (API Level 29)
- 64 GB de stockage
- Noyau Linux 4.9.118

2 Métriques et outils de mesures

2.1 Outil Perfetto

Cet outil fonctionne sur un navigateur web (https://ui.perfetto.dev/) ou en invite de commande avec *Android Debug Bridge* (adb) en entrant la commande **./adb shell Perfetto**. Cependant, l'utilisation de *Perfetto* sur un navigateur web est beaucoup plus simple d'accès. Il permet d'enregistrer des traces pour sonder de nombreuses métriques sur une durée prédéfinie d'un mobile Android connecté en USB sur l'ordinateur.
Par exemple, [1] utilise Perfetto pour réaliser les mesures dans Scénario 2.

2.1.1 Métriques sur Perfetto

Les métriques proposées par *Perfetto* sont divisées en 4 catégories (mesurées sur la durée de la trace) :
- CPU : fréquence de chaque coeur, utilisation de chaque coeur, événements d'ordonnancement, nombre d'appels systèmes effectués
- GPU : fréquence du GPU, utilisation de la mémoire du GPU pour chaque processus
- Batterie : batterie restante en pourcentage et en microampere-heures (μAh), courant instantané en microampere (μA)
- Mémoire :

  obtenir des informations sur chaque processus au niveau de la mémoire :

  - mem.rss : quantité de mémoire utilisée par un processus, elle vaut mem.rss.anon + mem.rss.file + mem.rss.shmem
  - mem.rss.anon : quantité de mémoire utilisée par les pages anonymes utilisées par le processus
  - mem.rss.file : quantité de mémoire utilisée par les file-backed pages utilisées par le processus
  - mem.rss.shmem : quantité de mémoire utilisée par les pages partagées utilisées par le processus
  - mem.rss.watermark : indique la valeur maximale que mem.rss a atteint depuis le démarrage du processus
  - mem.swap : quantité de mémoire de l'espace de swap utilisé par le processus
  - mem.virt : quantité de mémoire virtuelle utilisée par le processus
  - oom_score_adj : affiche la valeur de oom_score_adj associée au processus

  afficher la valeur de chaque métrique du fichier /proc/vmstat, affiche les événements Low Memory Killer, voir les pics d'utilisation de la mémoire,

obtenir le contenu du fichier /proc/meminfo (<=> cat /proc/meminfo), afficher le graphe des objets Java (objets Java qui ont des références entre eux) d'un processus, recenser les allocations/désallocations sur le heap d'un processus

2.2 Outil Android Debug Bridge

Cet outil fonctionne en invite de commande en entrant **./adb** dans le dossier *\sdk_Android_Studio\platform-tools* sur un ordinateur (*\sdk_Android_Studio* étant le dossier où l'Android SDK a été installé sur l'ordinateur). *adb* permet à un ordinateur de communiquer avec un appareil Android par USB (ou Wi-Fi avec Android Studio pour les appareils ayant au moins Android 11) pour réaliser diverses actions comme installer des applications, tuer des processus, récupérer des informations sur le mobile etc.
Par exemple, [1] utilise Android Debug Bridge pour mesurer le temps de démarrage d'applications dans leur Figure 11.

2.2.1 Métriques sur adb

Voyons les métriques proposées par cet outil (les commandes indiquées utilisent la syntaxe de Windows PowerShell, liste des services offrant des métriques : **./adb shell dumpsys -l**) :

— obtenir des informations courtes sur les appareils connectés à l'ordinateur : **./adb devices -l**

Exemple de sortie obtenue :
2c90a56d2c027ece device product :starltexx model :SM_G960F device :starlte transport_id :1

— Premier champ : numéro de série identifiant de manière unique chaque appareil connecté à l'ordinateur.
— Deuxième champ : variante spécifique du modèle du mobile.
— Troisième champ : modèle du mobile. SM = Samsung Mobile, G = Galaxy, 960 = S9, F = attributs spécifiques au modèle.
— Quatrième champ : comme le deuxième champ.
— Cinquième champ : moyen de connexion entre le mobile et l'ordinateur, ici transport_id :1 indique que l'USB est utilisé. Le Wi-Fi aurait été une autre valeur.

— obtenir des informations détaillées sur les appareils connectés à l'ordinateur : **./adb shell getprop**

La sortie renvoyée est très longue. Pour la rendre plus lisible, on peut utiliser la commande :
./adb shell
getprop | grep -e 'model' -e 'version.sdk' -e 'manufacturer' -e 'hardware' -e 'platform' -e 'revision' -e 'serialno' -e 'product.name' -e 'brand'

— commande dumpsys :
  — obtenir des informations sur la batterie (batterie actuelle, technologie de batterie, tension etc.) : **./adb shell dumpsys battery**

  - obtenir des informations sur le Wi-Fi (Wi-Fi activé ou non sur le mobile, SSID, adresse IP etc.) : **./adb shell dumpsys wifi**
  - obtenir des informations sur l'utilisation du CPU par les processus : **./adb shell dumpsys cpuinfo**
  - obtenir des informations sur l'utilisation de la mémoire par les processus au moment de l'appel de la commande : **./adb shell dumpsys meminfo**
  - obtenir des informations sur l'utilisation de la mémoire par les processus sur une période donnée : **./adb shell dumpsys procstats**
  - obtenir une liste des requêtes réseaux effectuées par les applications : **./adb shell dumpsys connectivity**
  - obtenir des informations en temps réel sur le système (taille de la mémoire, quantité d'espace de swap utilisée, liste des processus, mémoire virtuelle utilisée par chaque processus, pourcentage d'utilisation du CPU et de la mémoire par chaque processus, etc.) : **./adb shell top**

- récupérer la métrique *Time To Initial Display* (TTID) qui mesure le temps qu'il faut pour qu'une application soit prête à produire sa première frame (cette métrique est présente pour n'importe quelle application et est calculée automatiquement sans avoir besoin de modifier le code de l'application) :
  ./adb shell
  logcat | grep "Displayed"

  Exemple de sortie obtenue :

  02-16 19 :35 :51.303 913 1037 I ActivityTaskManager : Displayed com.rovio.baba/com.unity3d.player.UnityPlayerActivity : +1s320ms

  Sur cet exemple, on peut voir que le TTID de l'application Angry Birds 2 est de 1s320ms.

- récupérer la métrique *Time To Full Display* (TTFD) : cette métrique est optionnelle => elle n'est donc pas calculée par défaut. Elle représente le temps qu'il s'est écoulé entre le démarrage de l'application et un appel à la méthode reportFullyDrawn (). C'est donc à l'initiative du développeur de l'application de décider quand placer cet appel dans le code de son application : la convention indique que l'appel doit être réalisé quand l'application est entièrement affichée et que l'utilisateur peut interagir avec elle.
  ./adb shell
  logcat | grep "Fully drawn"

  Exemple de sortie obtenue :

  02-16 19 :35 :57.132 913 1037 I ActivityTaskManager : Fully drawn com.rovio.baba/com.unity3d.player.UnityPlayerActivity : +7s221ms

  Sur cet exemple, on peut voir que le TTFD de l'application Angry Birds 2 est de 7s221ms.

- obtenir des informations sur la mémoire virtuelle :
  ./adb shell cat /proc/vmstat

- obtenir des informations sur la mémoire physique et l'espace de swap :
  ./adb shell cat /proc/meminfo

— obtenir des informations sur la température des composants matériels (CPU, batterie, port USB) :
./adb shell dumpsys thermalservice

— obtenir des informations en temps-réel sur les événements de toucher sur l'écran du mobile :
./adb shell getevent

2.3 Métriques/outils supplémentaires

— Pour tracer le processus de rendu d'images, il existe *Systrace* qui peut fonctionner avec *Perfetto*.

— L'outil *UI/Application Exerciser Monkey* permet de simuler le comportement d'un humain sur une application. La commande est **./adb shell monkey**

— La métrique *Refresh Rate* (FPS) est affichable via les options du mobile (à partir de Android 11) ou via une application installée sur le Google Play Store comme *Real-Time FPS Meter* ou en activant GPUWatch dans les options développeur pour une application.

— La métrique *Ratio of Interaction Alert* (RIA) peut être mesurée avec les *interaction alert* détectées par *Systrace* : elle est incrémentée si une frame n'a pas réussi à être affichée dans les 16.6ms <=> 60FPS.

— Resident Set Size (RSS) : cette métrique indique le nombre de pages partagées et non-partagées utilisées par une application => indique toute la mémoire utilisée par un processus. Elle est accessible avec perfetto ou **cat /proc/<pid>/status**.

— Proportional Set Size (PSS) : cette métrique indique le nombre de pages non partagées utilisées par une application et un partage équitable des pages partagées. Elle est accessible avec **dumpsys meminfo <nom de l'application>** (Exemple : soit 3 processus qui utilisent une bibliothèque partagée de 30 pages. On a : 30/3 = 10 => le PSS indiquera 10 pages partagées pour chacun des 3 processus).

— Unique Set Size (USS) : cette métrique indique le nombre de pages non partagées utilisées par une application. Elle est accessible avec **dumpsys meminfo <nom de l'application>**

— Minor page fault : cette métrique est accessible pour un processus ayant le pid <pid> en entrant la commande **cat /proc/<pid>/stat | awk '{print $10}'**. Cette métrique est également accessible pour le système en entrant la commande :
cat /proc/vmstat | awk 'NR==59 {sum=$2} NR==60 {sum-=$2} END{printf("Number of minor page faults : %d\n", sum);}'
On l'obtient en soustrayant *pgfault* (défauts de page mineurs + majeurs) par *pgmajfault* (défauts de page majeurs).

— Major page fault : cette métrique est accessible pour un processus ayant le pid <pid> en entrant la commande **cat /proc/<pid>/stat | awk '{print $12}'**. Cette métrique (*pgmajfault*) est également accessible pour le système en entrant la commande **cat /proc/vmstat | awk 'NR == 60 {print}'**.

— En activant le mode développeur et en allant dans l'onglet Developper Options de l'application Settings, beaucoup de métriques sont accessibles : enregistrer une trace (System Tracing), voir l'usage de la mémoire physique par le système et les applications en temps réel (Running Services), GPUWatch pour obtenir des informations sur l'utilisation du CPU par une application, etc.

2.4 Outils utilisés par les autres papiers

Papier [1] :
— Pour la réalisation des mesures des Figure. 1 et Figure. 8, [1] utilise *systrace*, un outil maintenant déprécié qui permettait de réaliser des traces sur de nombreuses métriques systèmes. Il est maintenant conseillé d'utiliser *Perfetto* à la place de *systrace*.
— Pour la réalisation des mesures des Figure. 2, Figure. 3, Figure. 4 et Figure. 10, [1] a modifié le code source Android pour récupérer les informations nécessaires aux mesures réalisées dans ces figures.

Papier [2] :
— Pour la réalisation des mesures de la Figure. 2, [2] utilise son propre mécanisme de log développé par ses auteurs.

Papier [3] :
— Pour la réalisation des mesures de la Figure. 3, [3] utilise *FTrace*, un sous-ensemble de *systrace* permettant de réaliser des traces au niveau du noyau. *Ftrace* provient de Linux.
— Pour la réalisation des mesures de Table. 1 et Table. 2, [3] utilise *procstats*, une commande de adb : *adb shell dumpsys procstats*

Papier [4] :
— Pour la réalisation des mesures de la Figure. 1, [4] utilise *Astat*, une application de tracage développée par ses auteurs qui permet de compter le nombre d'interactions (nombre de toucher, nombre d'événements réalisés, ...) d'un utilisateur sur les applications de son mobile.
— Pour la réalisation des mesures de la Figure. 3, [4] utilise *Android traceview*, un outil maintenant déprécié qui permettait d'afficher de manière graphique les log des traces des applications que l'on a développé. Il est maintenant conseillé d'utiliser *CPU Profiler* d'Android Studio à la place d'*Android traceview*.
— Pour la réalisation des mesures de la Figure. 9, [4] utilise *blktrace* et *blkparse*. *blktrace* est un outil qui génère des traces donnant des informations sur les entrées-sorties au niveau de la couche bloc dans un format peu lisible pour l'humain, et *blkparse* permet de rendre les résultats de *blktrace* plus simple à lire pour un humain. Ces deux outils proviennent de Linux.
— Pour la réalisation des mesures de la Figure. 14, [4] utilise *procrank*, une commande de adb : *adb shell procrank*

3 Mesures réalisées

3.1 Scénario 1 : mesurer le temps de démarrage d'applications

Ces mesures ont été réalisées dans [4] dans la Fig.2. Les auteurs ont développé un outil nommé MARS permettant d'accélérer le temps de lancement d'applications Android. Ils ont réalisé cette mesure pour montrer le temps de lancement d'applications sans leur outil. Dans la Fig.11, les auteurs montrent l'amélioration du temps de lancement des applications avec leur outil.

Je vais mesurer le temps de démarrage (cold start et hot start) de 6 applications parmi les 20 applications indiquées dans la Table 3 de [1] sans qu'il n'y ait aucune application en arrière-plan.

Une application est lancée en cold start/launching si elle n'est pas déjà en mémoire à son lancement. C'est le cas quand l'application est lancée pour la première fois depuis le démarrage du système, ou bien l'application avait été tuée par le système précédemment. Le système va devoir réaliser de nombreuses tâches comme la création du processus de l'application, de l'activité et du thread principal.
Une application est lancée en hot start/launching si toutes ses données sont déjà en mémoire à son lancement. C'est le cas quand l'utilisateur d'un mobile met une application en arrière-plan sans la fermer. Le système aura juste à ramener au premier plan l'application résidant en arrière-plan.

Pour réaliser ces mesures, deux scripts ont été créés : `script_startup_cold.sh` et `script_startup_hot.sh` pour respectivement mesurer le temps d'un cold start et d'un hot start d'une application (champ TotalTime de la commande adb shell am start-activity -W -S packageName/.activityName pour le cold start, l'option -S est retiré pour le hot start car celle-ci tue l'application à chaque lancement. A la place, on simule un appuie sur la touche home pour garder l'application en mémoire). A l'inverse de [4] qui lance chaque application 10 fois en cold start et en hot start, je lance chaque application 30 fois en cold start et en hot start pour pouvoir calculer un intervalle de confiance à 95% : une moyenne des 30 lancements sera réalisée pour obtenir la durée moyenne d'un cold start et d'un hot start pour l'application.

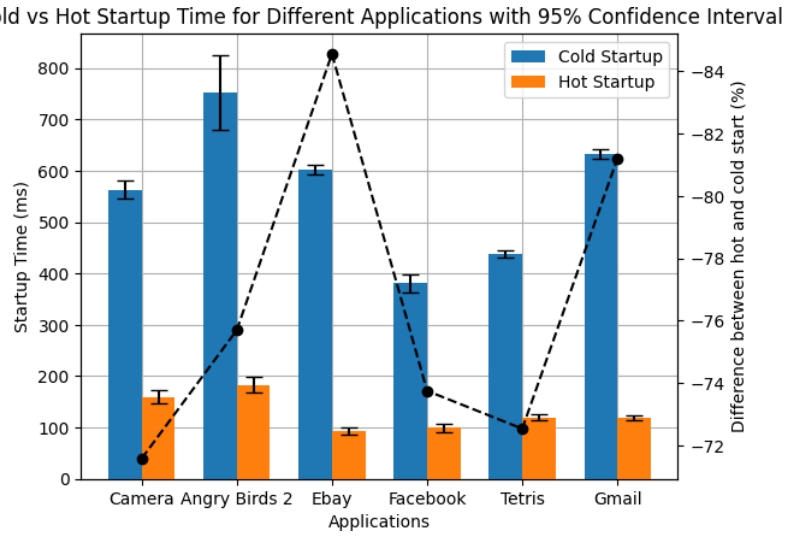


FIGURE 1 – Cold start et Hot start de différentes applications

Avec la figure 1, voici ce qu'on observe au niveau des pourcentages de différence entre le cold startup et le hot startup pour les 6 applications :

- Pour l'application Camera : le hot startup est environ 71.5% plus rapide que le cold startup.
- Pour l'application Angry Birds 2 : le hot startup est environ 75% plus rapide que le cold startup.
- Pour l'application Facebook : le hot startup est environ 73% plus rapide que le cold startup.
- Pour l'application Ebay : le hot startup est environ 84.5% plus rapide que le cold startup.
- Pour l'application Tetris : le hot startup est environ 72.5% plus rapide que le cold startup.
- Pour l'application Gmail : le hot startup est environ 81.2% plus rapide que le cold startup.

Cette mesure permet de conclure qu'une application lancée en cold startup met beaucoup plus de temps à s'afficher qu'en hot startup. En effet, une application lancée en hot startup se lancera environ plus de 76% plus vite qu'en cold startup. Cela provient du fait qu'une application lancée en cold startup doit avoir son processus de créé et celui-ci doit être placé en mémoire alors qu'en hot startup, toutes les données de l'application sont déjà en mémoire.

En comparant mes mesures avec celles de [4] pour les applications Tetris et Gmail :

- pour Tetris, on est passé de 20 secondes de temps de lancement en cold startup à 437ms.
- pour Gmail, on est passé de 5 secondes de temps de lancement en cold startup à 633ms.

Les différences énormes entre mes mesures et celles de [4] peuvent être expliquées par deux choses : en 10 ans ([4] date de 2014), les applications ont sûrement été optimisées donc leurs temps de lancement ont peut-être été réduits. Cependant, la cause principale de ces différences est sûrement l'évolution de la puissance des mobiles entre 2014 et aujourd'hui.

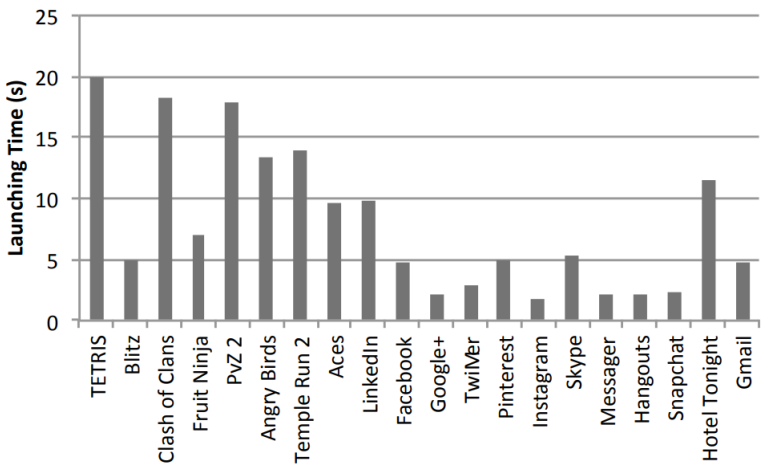


Fig. 2. Launching time of some popular applications on Android

3.2 Scénario 2 : utilisation du CPU

Cette mesure a été réalisée dans [1]. Les auteurs voulaient montrer que les applications en arrière-plan ne sont pas grandes consommatrices de CPU en général.

Comme indiqué dans la Table 1 de [1], je vais mesurer l'utilisation du CPU selon le nombre d'applications qu'il y a en arrière-plan : ces applications seront choisies au hasard parmi les 20 indiquées dans la Table 3 de [1]. 5 cas seront mesurés : 0, 2, 4, 6 et 8 applications en arrière-plan.

Pour réaliser ces mesures, un script a été créé : `script_cpu_all.sh`. Ce script va réaliser les opérations suivantes :

- il va lancer les n applications en arrière-plan (commande adb shell am start-activity -W -S packageName/.activityName)
- il va laisser le temps aux applications de se lancer
- il va attendre 2 secondes pour simuler une utilisation de l'application
- il simule un appuie sur la touche Home du mobile et lance l'application suivante
- il va additionner le pourcentage d'utilisation du CPU de chaque processus obtenu avec la commande `./adb shell top` (l'utilisation du CPU par le shell produit par adb n'est pas compté car il ne fait pas réellement parti du mobile)
- il va calculer la moyenne et la valeur maximale d'utilisation du CPU trouvées. Ces deux dernières valeurs sont divisées par 8 car comme le CPU de mon mobile possède 8 cœurs, les valeurs d'utilisation du CPU retournées par `./adb shell top` peuvent dépasser les 100% jusqu'à atteindre les 800%.

Par rapport à la section 2.2.3 (1) de [1] qui effectue 10 mesures pour chaque cas, les mesures seront réalisées 100 fois pour chaque cas.

On obtient le tableau suivant :

Num. of BG apps	CPU Utilization	
(No FG app)	Average	Peak
0	6.3%	36.3%
2	25.9%	48.7%
4	28.8%	51%
6	32.3%	53.3%
8	36.9%	50.7%

TABLE 1 – Utilisation moyenne et maximale du CPU pour chaque cas

Avec la table 1, on remarque que la valeur du pic d'utilisation du CPU n'évolue pas conjointement avec le nombre d'applications en arrière-plan. En effet, à partir de 2 applications en arrière-plan, le pic d'utilisation du CPU reste stable.
On observe :

- une différence d'environ 30% entre 0 et 2 applications en arrière-plan
- une différence d'environ 4.5% entre 2 et 4 applications en arrière-plan
- une différence d'environ 4.5% entre 4 et 6 applications en arrière-plan
- une différence d'environ 5% entre 6 et 8 applications en arrière-plan

Par rapport à l'utilisation moyenne du CPU pour chaque cas, on obtient le graphique suivant :

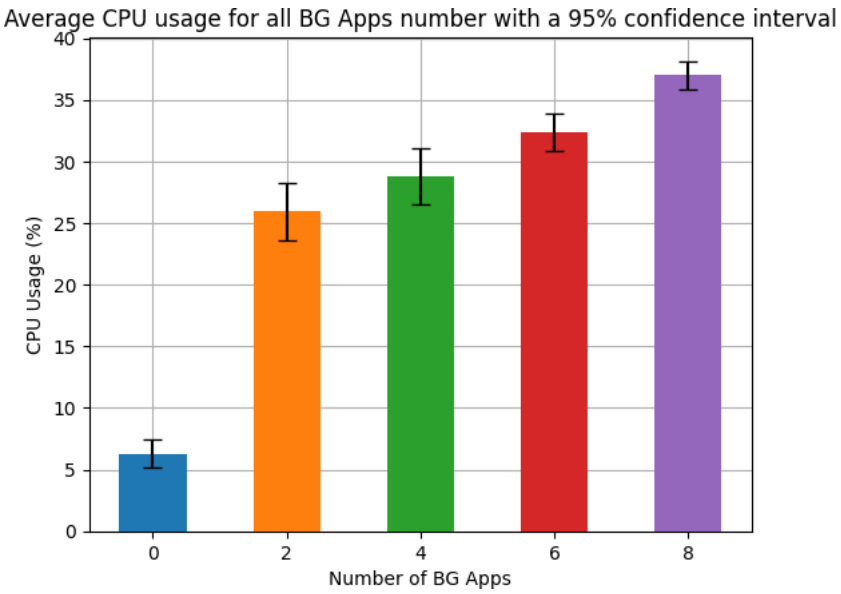


FIGURE 2 – Utilisation moyenne du CPU pour chaque cas

Avec la table 1 et la figure 2, on observe la plus grande différence d'utilisation moyenne du CPU quand on passe de 0 à 2 applications en arrière-plan (augmentation de 311% de l'utilisation moyenne du CPU). De plus, on remarque qu'avec les cas suivants, l'évolution de l'utilisation moyenne du CPU reste stable :

- Augmentation de 11% de l'utilisation moyenne du CPU entre le cas 2 et 4 applications en arrière-plan
- Augmentation de 12% de l'utilisation moyenne du CPU entre le cas 4 et 6 applications en arrière-plan
- Augmentation de 14% de l'utilisation moyenne du CPU entre le cas 6 et 8 applications en arrière-plan

On peut voir dans la table 1 et la figure 2 qu'augmenter le nombre d'applications en arrière-plan entraîne une augmentation presque linéaire de l'utilisation moyenne du CPU. Le pic d'utilisation du CPU reste quant à lui stable avec l'augmentation du nombre d'applications en arrière-plan. Ces mesures permettent de conclure que les applications en arrière-plan ne sont pas très gourmandes en consommation CPU. J'obtiens la même conclusion que [1] dans la section 2.2.3 (1).

On peut remarquer que toutes mes mesures sont inférieures à celles de [1]. Cela peut venir du fait que mon mobile est différent de celui utilisé dans [1] (HUAWEI P20) et que les performances entre les deux mobiles sont différentes : il est donc compliqué de comparer directement les valeurs de mes résultats avec ceux de [1].
Cependant, une comparaison des ordres de grandeurs est possible. Mes valeurs mesurées semble cohérentes pour plusieurs raisons :

- les mesures dans le cas Average sont strictement croissantes comme dans [1].
- dans le cas Average pour 2, 4, 6 et 8 applications en arrière-plan : j'ai observé dans mes mesures une augmentation moyenne de la consommation CPU de 12.3% alors que dans [1] l'augmentation moyenne de la consommation CPU est de 6.1%. Cela représente une différence de 6 points de pourcentage ce qui ne me semble pas une différence significative.

La plus grande différence apparaît dans le cas 0 application en arrière-plan. En effet, le CPU de mon mobile est utilisé à 6.3% en moyenne alors que dans [1], le CPU du

mobile est utilisé à 43% en moyenne. Cela représente une différence de 36.7 points de pourcentage : cela montre bien les différences de performance du CPU entre les deux mobiles.

Table 1. CPU utilization with $N(0 \sim 8)$ apps in the BG.

Num. of BG apps	**CPU Utilization**	
(No FG app)	**Average**	**Peak**
0	43%	52%
2	46%	58%
4	47%	63%
6	51%	67%
8	55%	69%

3.3 Scénario 3 : mesure de l'utilisation de la mémoire et du CPU par une application

Ces mesures ont été réalisées dans [2] dans la Fig.2. Les auteurs voulaient montrer l'usage du CPU, de la mémoire et du réseau par une application (YouTube). Les auteurs ont développé un outil nommé DVFS permettant de réduire la consommation en énergie d'un mobile en ajustant la fréquence du CPU selon l'application en premier plan.

Je vais mesurer l'utilisation de la mémoire et du CPU pour une application donnée : ici YouTube.

Pour réaliser ces mesures, deux scripts ont été créés : `script_cpu_one_ps.sh` et `script_memory_RSS.sh` qui vont respectivement mesurer l'utilisation du CPU (champ %CPU de la commande top qui est divisé par 8 comme le CPU du mobile possède 8 coeurs) et de la mémoire (champ VmRSS du fichier /proc/[pid YouTube]/status) chaque seconde par le processus, ici YouTube, dans 2 cas différents :

- différence d'utilisation de la mémoire et du CPU entre une vidéo en qualité 360p qui se joue en premier plan sur l'application YouTube et une vidéo (identique, vidéo de course de F1) en qualité 1080p
- différence d'utilisation de la mémoire et du CPU entre une vidéo où l'image est toujours noire, une vidéo "qui bouge beaucoup" (vidéo de course de F1) et une vidéo d'un dessin animé, toutes en 1080p

100 mesures de chaque ont été réalisées (100 secondes).

3.3.1 Cas vidéo 1080p Vs vidéo 360p

3.3.1.1 Mémoire

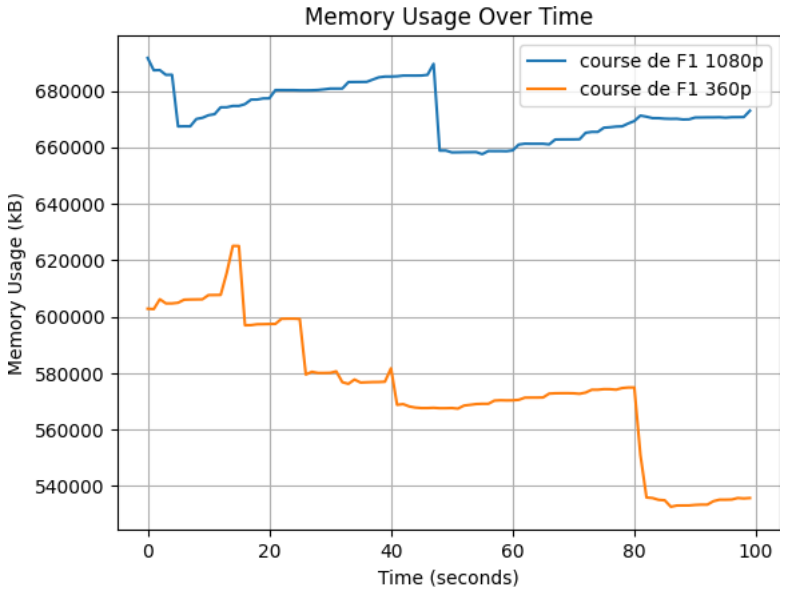


FIGURE 3 – Évolution de l'utilisation de la mémoire pour chaque cas

Avec la figure 3, on peut voir que la vidéo en 1080p consomme plus de mémoire que la vidéo en 360p.

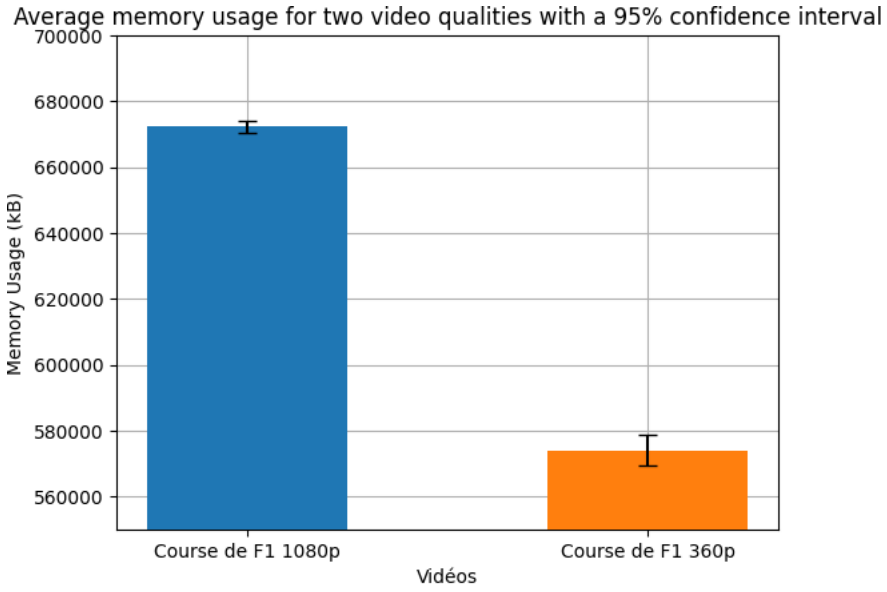


FIGURE 4 – Utilisation moyenne de la mémoire pour chaque cas

Avec la figure 4, on observe une augmentation de 17.1% de l'utilisation moyenne de la mémoire entre le cas 1080p et 360p.

Avec la figure 3 et la figure 4, on remarque que la qualité de la vidéo a un impact sur sa consommation mémoire : une vidéo de meilleur qualité consommera plus de mémoire que la même vidéo dans une qualité inférieure.

3.3.1.2 CPU

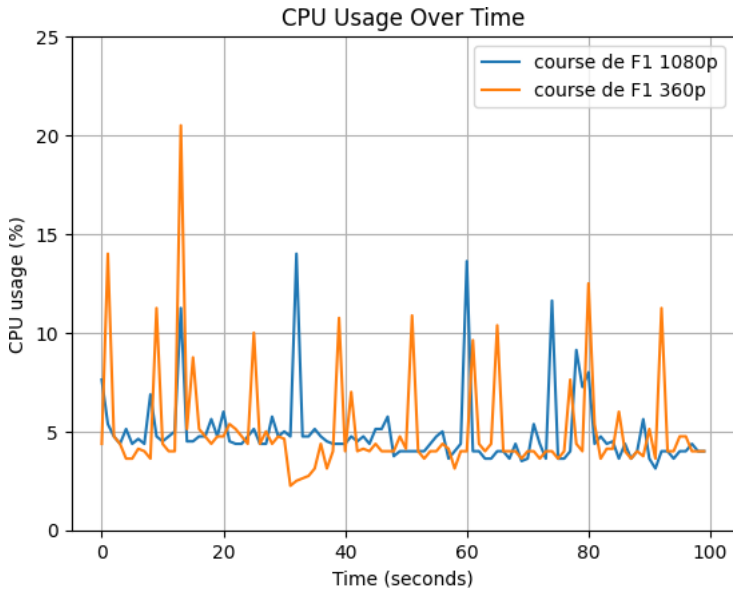


FIGURE 5 – Évolution de l'utilisation du CPU pour chaque cas

Avec la figure 5, on peut voir qu'il n'y a pas de différences significatives dans l'évolution de la consommation du CPU par les deux vidéos.

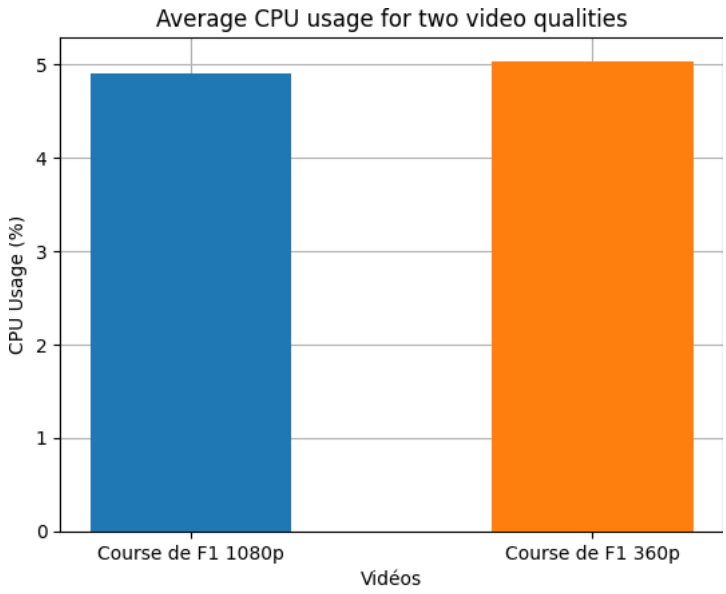


FIGURE 6 – Utilisation moyenne du CPU pour chaque cas

Avec la figure 6, on observe une différence de 2.5% de l'utilisation moyenne du CPU entre le cas 1080p et 360p.

Avec la figure 5 et la figure 6, on remarque que la qualité de la vidéo ne joue pas de rôle significatif sur sa consommation du CPU : elle reste proche pour les deux qualités de vidéo.

3.3.2 Cas trois types de vidéos de même qualité

3.3.2.1 Mémoire

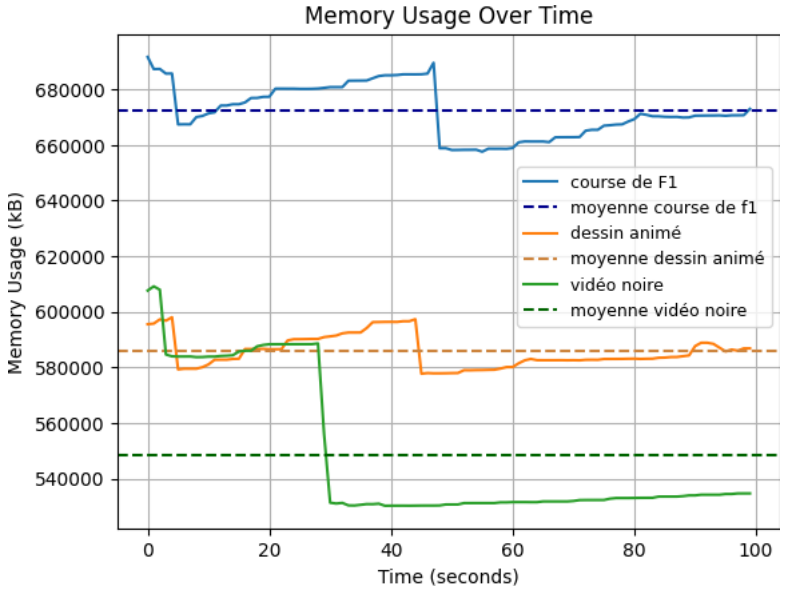


FIGURE 7 – Évolution de l'utilisation de la mémoire pour chaque cas

Avec la figure 7, on remarque que la consommation mémoire des trois types de vidéos évolue d'une manière similaire. En effet, on remarque dans les 3 cas :

— une baisse de la consommation mémoire au début des mesures, donc au lancement de la vidéo
— une augmentation de la consommation mémoire
— une chute brutale de la consommation mémoire
— une augmentation de la consommation mémoire jusqu'à la fin des mesures

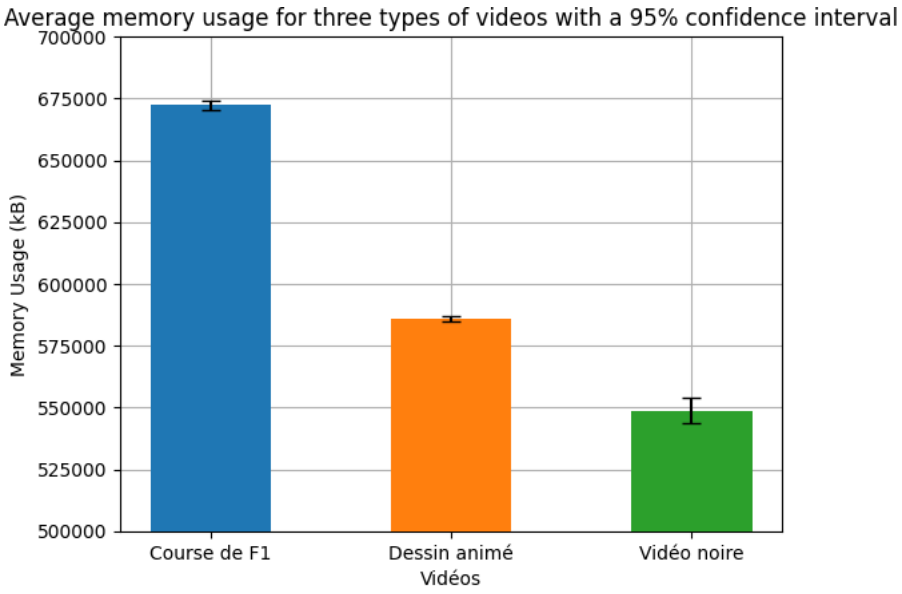


FIGURE 8 – Utilisation moyenne de la mémoire pour chaque cas

Avec la figure 8, on observe que :
— la vidéo de course de F1 consomme en moyenne 14.7% en plus de mémoire que la vidéo de dessin animé.
— la vidéo de course de F1 consomme en moyenne 22.5% en plus de mémoire que la vidéo noire.
— la vidéo de dessin animé consomme en moyenne 6.7% en plus de mémoire que la vidéo noire.

Avec la figure 7 et la figure 8, on remarque également (comme dans le cas précédent) que le contenu de la vidéo a un impact sur sa consommation mémoire : si le contenu de la vidéo est plus gourmand graphiquement, sa consommation mémoire sera plus importante.

3.3.2.2 CPU

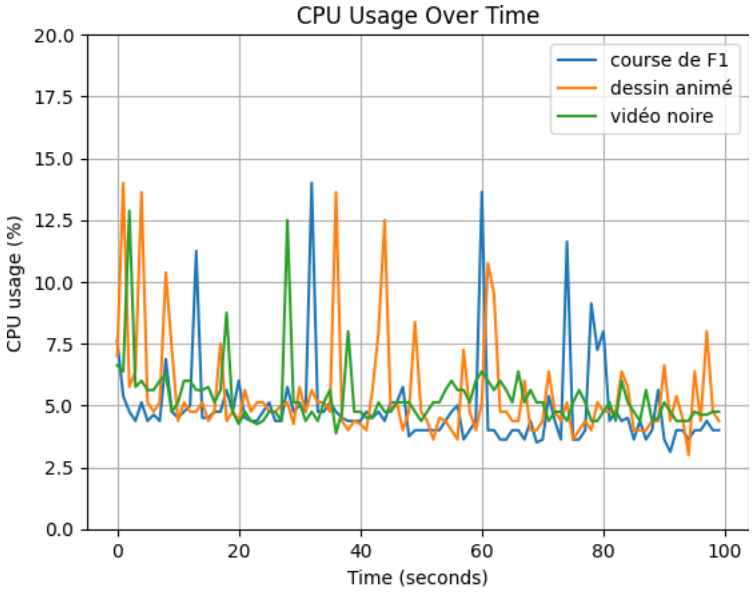


FIGURE 9 – Évolution de l'utilisation du CPU pour chaque cas

Avec la figure 9, on peut voir qu'il n'y a pas de différences significatives dans l'évolution de la consommation du CPU par les trois types de vidéos.

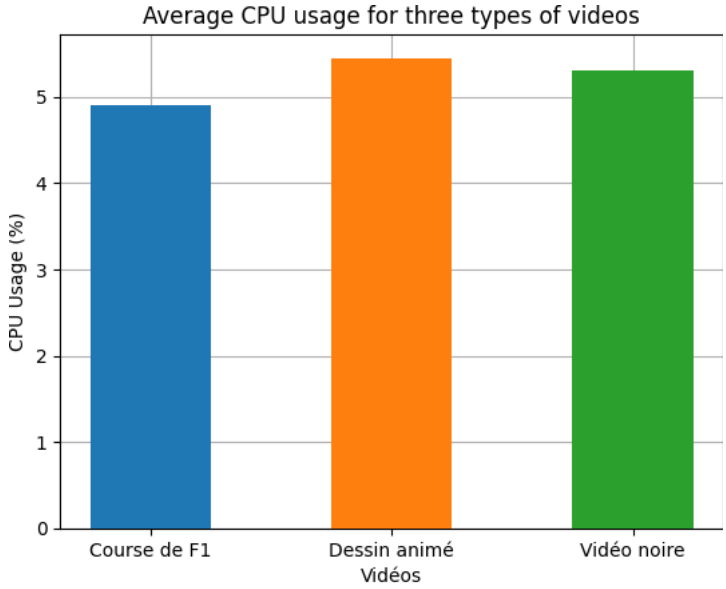


FIGURE 10 – Utilisation moyenne du CPU pour chaque cas

Avec la figure 10, on observe que :

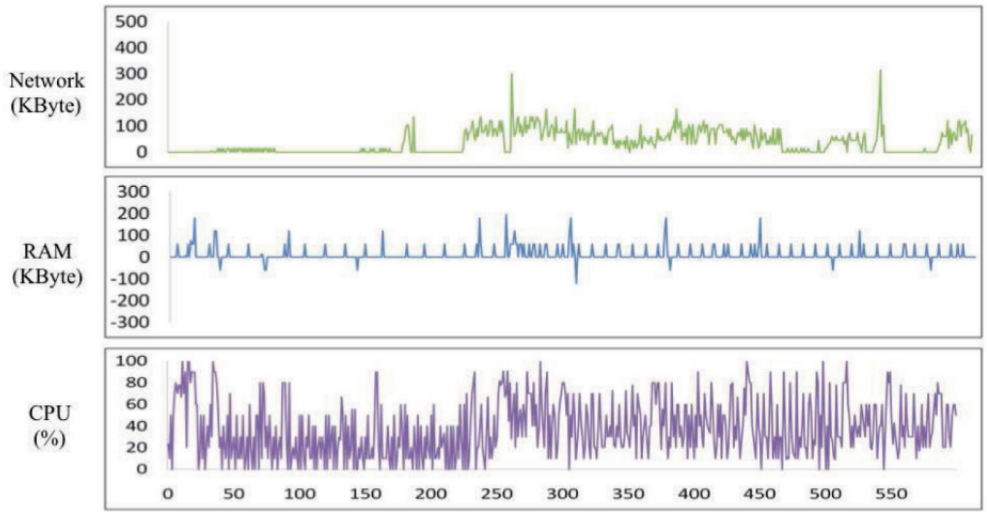
- la course de F1 consomme en moyenne 4.9% le CPU. Il y a une différence de 11% entre la course de F1 et le dessin animé pour la consommation moyenne du CPU.
- le dessin animé consomme en moyenne 5.4% le CPU. Il y a une différence de 2.6% entre le dessin animé et la vidéo noire pour la consommation moyenne du CPU.

— la vidéo noire consomme en moyenne 5.3% le CPU. Il y a une différence de 8.2% entre la vidéo noire et la course de F1 pour la consommation moyenne du CPU.

Comme dans le premier cas, on ne remarque pas de différences significatives d'utilisation du CPU selon le contenu de la vidéo.

3.3.3 Conclusion

Grâce à ces deux cas (premier cas et deuxième cas), on peut conclure que la qualité de la vidéo et son contenu ont un rôle significatif sur sa consommation mémoire. En effet, une vidéo d'une qualité plus élevée consommera plus de mémoire, de même pour une vidéo coûteuse graphiquement. Cependant, la qualité ou le contenu de la vidéo ne jouent pas de rôle significatif sur sa consommation CPU.



(a) YouTube.

3.4 Scénario 4 : mesure du PSS maximum d'applications

Ces mesures ont été réalisées dans [3] dans la Table 1. Les auteurs ont réalisé cette mesure pour montrer que toutes les applications Android ne demandent pas la même quantité de mémoire au maximum pour fonctionner. Dans le cas où il reste pas assez de mémoire libre dans le mobile et que l'utilisateur lance une application qui possède un PSS supérieur à cette quantité de mémoire, de la réclamation de mémoire sera réalisée, ralentissant le temps de lancement de cette application et augmentant le nombre d'instructions noyau réalisées.

Je vais mesurer le PSS maximum de certaines applications les plus connues du Google Play Store.
Proportional Set Size (PSS) : cette métrique indique le nombre de pages non partagées utilisées par une application et un partage équitable des pages partagées. Elle est accessible avec **dumpsys meminfo <nom de l'application>** (Exemple : soit 3 processus qui utilisent une bibliothèque partagée de 30 pages. On a : 30/3 = 10 => le PSS indiquera 10 pages partagées pour chacun des 3 processus).

Pour réaliser ces mesures, la commande suivante sera utilisée :
./adb shell
dumpsys procstats <nom application> –hours 24 | grep -A 5 "Process summary :" | grep -m 1 "TOTAL"

Les données retournées par cette commande sont de la forme :
TOTAL : 21% (495MB-822MB-0.94GB/331MB-444MB-623MB/450MB-586MB-816MB over 10)

- 21% : pourcentage de temps pendant lequel l'application était en train de s'exécuter ces dernières 24 heures.
- 495MB-822MB-0.94GB : PSS de l'application sous la forme minPSS-avgPSS-maxPSS
- 331MB-444MB-623MB : USS de l'application sous la forme minUSS-avgUSS-maxUSS
- 450MB-586MB-816MB : RSS de l'application sous la forme minRSS-avgRSS-maxRSS
- over 9 : nombre d'échantillons prélevés sur la période

Pour obtenir le PSS maximum d'une application, on a juste à récupérer le maxPSS indiqué par la commande ci-dessus.

On obtient le tableau suivant :

Application name	Package name	Category	Max PSS
Facebook	com.facebook.katana	Social	464 MB
Angry Birds 2	com.rovio.baba	Game	940 MB
Amazon Shopping	com.amazon.mShop.android.shopping	Shopping	373 MB
eBay	com.ebay.mobile	Shopping	263 MB
Cooking Fever	com.nordcurrent.canteenhd	Game	718 MB
Game of War	com.machinezone.gow	Game	500 MB
Pinterest	com.pinterest	Social	298 MB
Instagram	com.instagram.android	Social	348 MB
AliExpress	com.alibaba.aliexpresshd	Shopping	529 MB
Empire War : Age of Hero	com.feelingtouch.empirewaronline	Game	355 MB

TABLE 2 – PSS maximum pour chaque application

Avec ces mesures, on peut conclure que toutes les applications mobiles ne sont pas toutes autant gourmandes en ressources mémoire. En effet, selon les mesures réalisées dans la table 2, on remarque que parmi les applications choisies, les applications de catégorie Game sont les plus gourmandes en mémoire (en moyenne : 628MB de PSS). Les applications de Shopping et Social possèdent elles une consommation de la mémoire très proche (respectivement 388MB de PSS et 370MB de PSS en moyenne donc une différence d'environ 4.7% entre Shopping et Social, mais une différence d'environ 49% pour ces deux catégories avec Game).

Comparons mes résultats avec ceux de [3] :

- Facebook : 329 MB vs 464 MB => différence de 34%
- eBay : 135 MB vs 263 MB => différence de 64.3%
- Cooking Fever : 237 MB vs 718 MB => différence de 100.7%
- Pinterest : 276 MB vs 298 MB => différence de 7.6%
- Game of War : 401 MB vs 500 MB => différence de 21.9%
- Instagram : 248 MB vs 348 MB => différence de 33.5%
- Amazon Shopping : 124 MB vs 373 MB => différence de 100.2%
- Empire War : Age of Hero : 248 MB vs 355 MB => différence de 35%

On remarque trois tendances entre mes mesures et celles de [3] :

- Pinterest possède une différence de PSS d'environ 7.6%. Pour cette application, on peut parler d'une "petite" différence dans l'ordre de grandeur des 10%.
- Facebook, Instagram et Empire War : Age of Hero ont toutes une différence de PSS d'environ 34%. eBay possède une une différence de PSS d'environ 64.3%. Pour ces applications, on peut parler d'une différence "moyenne" dans l'ordre de grandeur des 50%.
- Cooking Fever et Amazon Shopping ont toutes une différence de PSS d'environ 100%. Pour ces applications, on peut parler d'une "grande" différence dans l'ordre de grandeur des 100%.

Ces différences de PSS entre mes mesures et celles de [3] sont parfaitement normales. Mes mesures ont été réalisées en 2024 alors que celles de [3] ont été réalisées en 2015. Cela nous montre qu'en 9 ans, comme les applications deviennent de plus en plus fournies en fonctionnalités, elles utilisent donc de plus en plus de mémoire. Bien que certaines ne changent pas trop (ex : Pinterest), certaines utilisent dans les 50% de mémoire en plus (ex : Facebook, Instagram et Empire War : Age of Hero) et certaines doublent même leur usage de la mémoire (ex : Cooking Fever et Amazon Shopping).

Selon https ://sensortower.com/blog/ios-app-size-growth-2021, la taille des applications mobiles sur iOS ne fait qu'augmenter au fil des années, jusqu'à presque quadrupler pour certaines applications entre 2016 et 2021. Sur Android, on remarque également cette augmentation : Amazon Shopping et Cooking Fever ont triplé en taille.

TABLE I. THE MAXIMUM MEMORY (PSS) OF FAMOUS APPLICATIONS OF GOOGLE PLAY ON NEXUS 5

Name	Category	Max PSS	Name	Category	Max PSS
Game of War	Game	401 MB	Cooking Fever	Game	237 MB
Facebook	Social	329 MB	The Weather Channel	Weather	198 MB
Candy Crush Saga	Game	328 MB	Waze	Travel and Location	193 MB
Clash of Clans	Game	314 MB	Kik	Communication	165 MB
Pinterest	Social	276 MB	SoundCloud	Music and Audio	153 MB
Instagram	Social	248 MB	Skype	Communication	129 MB
Empire War: Age of Hero	Game	248 MB	eBay	Shopping	135 MB
Dubsmash	Media and Movie	246 MB	Amazon Shopping	Shopping	124 MB
Angry Bird	Game	240 MB	Twiter	Social	122 MB

3.5 Scénario 5 : mesure de la consommation réseau

Ces mesures sont inspirées des mesures réalisées dans [2] dans la Fig.2.

Je vais mesurer le nombre d'octets reçus par l'interface réseau wlan0 du mobile (celle s'occupant des communications Wi-Fi) lorsque l'application YouTube est en premier plan avec une vidéo qui se joue sans aucune application en arrière-plan.

Pour réaliser ces mesures, un script a été créé : `script_network.sh`. Ce script va mesurer le nombre d'octets reçus par l'interface wlan0 chaque seconde pendant 100 secondes (champ wlan0 du fichier /proc/net/dev) dans 2 cas différents :

— mesure de la différence de consommation réseau entre une vidéo en qualité 360p qui se joue en premier plan sur l'application YouTube et une vidéo (identique, vidéo de course de F1) en qualité 1080p

— mesure de la différence de consommation réseau entre une vidéo où l'image est toujours noire, une vidéo "qui bouge beaucoup" (vidéo de course de F1) et une vidéo d'un dessin animé, toutes en 1080p

3.5.1 Cas vidéo 1080p Vs vidéo 360p

3.5.1.1 Consommation réseau

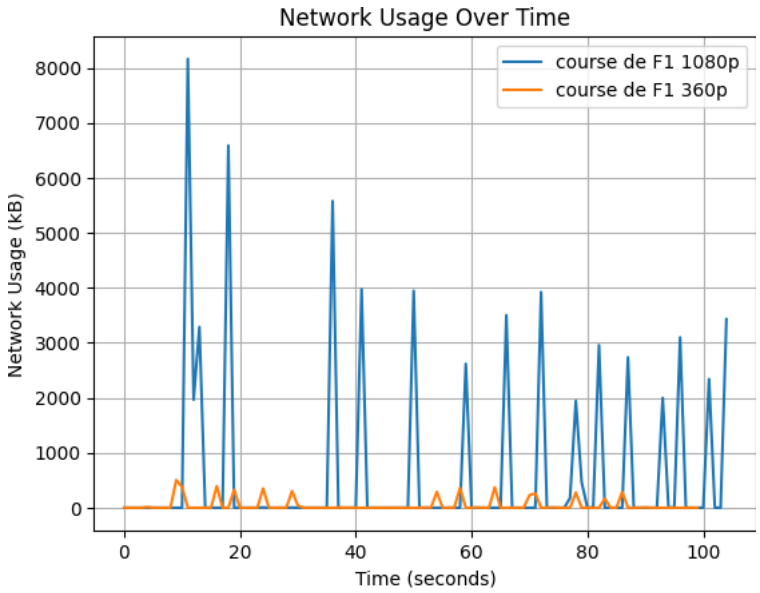


FIGURE 11 – Évolution de la consommation réseau pour chaque cas

Avec la figure 11, on observe que la consommation réseau de la vidéo en 1080p au cours du temps est largement supérieure à la vidéo en 360p, ce qui n'est clairement pas une surprise.

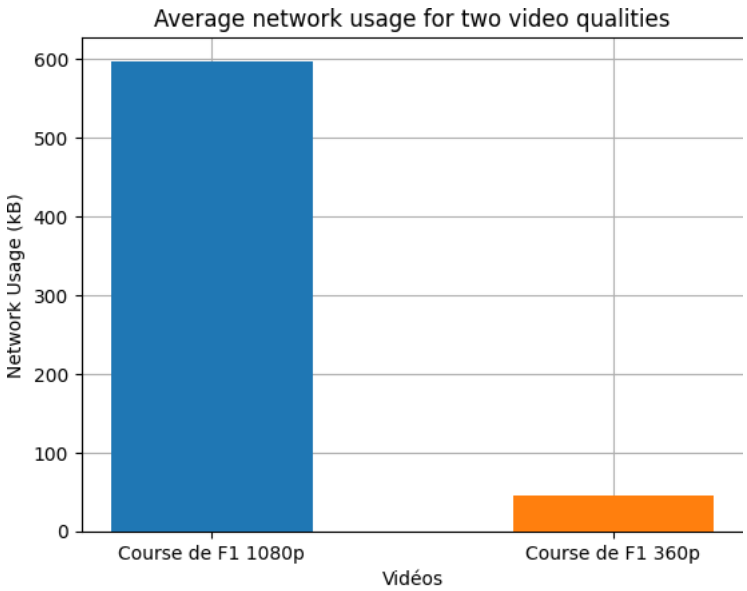


FIGURE 12 – Consommation réseau moyenne pour chaque cas

Il y a une augmentation de 1196.7% de la consommation réseau entre le cas 1080p et 360p (figure 12).

Voyons la valeur des écarts types et les intervalles de confiances à 95% :
- F1 1080p : 1503.95 kB, [302.711, 892.249] en kB
- F1 360p : 115.64 kB, [23.408, 68.739] en kB

Avec la figure 12, on remarque que pour la vidéo à 1080p et la vidéo à 360p, la moyenne calculée est inférieure à l'écart type. De plus, les intervalles de confiances sont tous les deux grands. Cela indique que les valeurs mesurées sont espacées les unes des autres et sont loin de la moyenne. Cela montre que la consommation réseau d'une vidéo varie énormément au cours du temps.

On remarque que la qualité de la vidéo joue un rôle majeur dans sa consommation réseau : une vidéo d'une meilleur qualité aura une plus grande consommation réseau que la même vidéo dans une qualité plus basse (figure 11 et figure 12)

3.5.2 Cas trois types de vidéos de même qualité

3.5.2.1 Consommation réseau

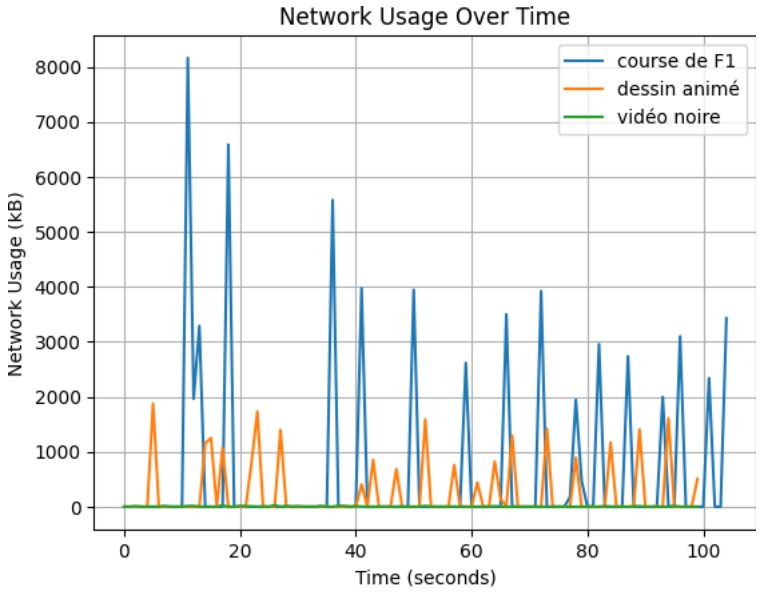


FIGURE 13 – Évolution de la consommation réseau pour chaque cas

Avec la figure 13, on observe que :

- la consommation réseau de la vidéo noire est extrêmement faible par rapport aux deux autres (Moyenne vidéo noire : 3 kB)
- la vidéo du dessin animé est deuxième en consommation réseau parmi les trois types de vidéos (Moyenne dessin animé : 233.3 kB)
- la vidéo de la course de F1 est celle qui possède la plus grande consommation réseau (Moyenne course de F1 : 597.4 kB)

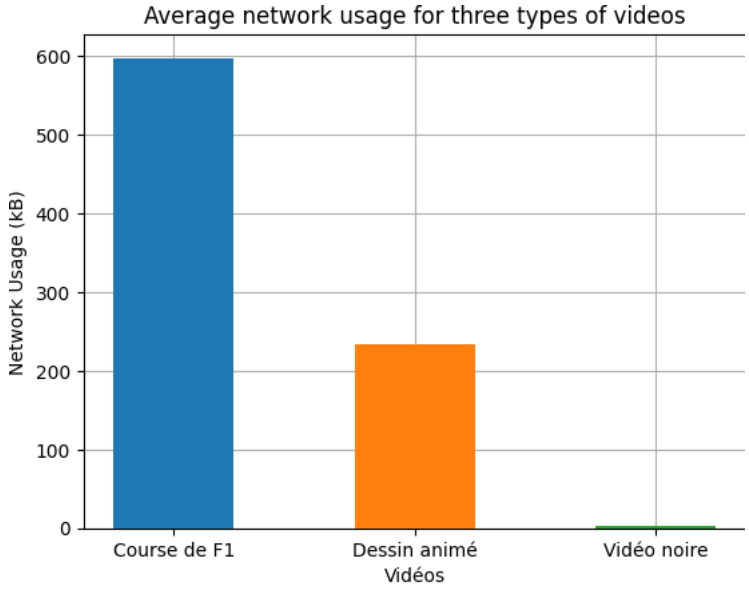


FIGURE 14 – Consommation réseau moyenne pour chaque cas

Avec la figure 14, on observe que :

- la vidéo de course de F1 consomme en moyenne 156% en plus de mémoire que la vidéo de dessin animé.
- la vidéo de course de F1 consomme en moyenne 19561% en plus de mémoire que la vidéo noire.
- la vidéo de dessin animé consomme en moyenne 7578.8% en plus de mémoire que la vidéo noire.

Voyons la valeur des écarts types et les intervalles de confiances à 95% :

- F1 : 1503.95 kB, [302.711, 892.249] en kB
- Dessin animé : 396.04 kB, [155.730, 310.977] en kB
- Vidéo noire : 5.72 kB, [1.918, 4.160] en kB

On peut remarquer que pour les 3 types de vidéos, la moyenne calculée est inférieure à l'écart type. De plus, les intervalles de confiances sont tous les trois grands. Cela indique que les valeurs mesurées sont espacées les unes des autres et sont loin de la moyenne. Comme dans le cas précédent, cela montre que la consommation réseau d'une vidéo varie énormément au cours du temps.

Avec la figure 13 et la figure 14, on remarque que le contenu de la vidéo joue également un rôle majeur dans sa consommation réseau : si le contenu de la vidéo est plus gourmand graphiquement, c'est-à-dire affiche des images qui changent souvent et qui sont coûteuses à afficher par le GPU, sa consommation réseau sera plus importante.

3.5.3 Conclusion

Grâce à ces deux cas (premier cas et deuxième cas), on peut conclure que la qualité de la vidéo jouée possède un grand impact sur sa consommation réseau : en effet, une vidéo de meilleure qualité aura une plus grande consommation réseau que la même vidéo dans une qualité moindre. On peut également conclure que le contenu de la vidéo joue un rôle dans sa consommation réseau : une vidéo dont le contenu est plus coûteux graphiquement aura une plus grande consommation réseau.

3.6 Scénario 6 : mesure du nombre de page reclaim et page refault

BG Null (100 secondes) : Page reclaimed : 5051, Page refault : 2185

Commandes pour obtenir ces valeurs :

- Page reclaimed : **adb shell cat /proc/vmstat | grep -w pgsteal_direct**
- Page refault : **adb shell cat /proc/vmstat | grep -w pgmajfault**

Références

[1] C. Li, Y. Liang, R. Ausavarungnirun, Z. Zhu, L. Shi, and C. J. Xue, "Ice : Collaborating memory and process management for user experience on resource-limited mobile devices," in *Proceedings of the Eighteenth European Conference on Computer Systems*, pp. 79–93, 2023.

[2] Y.-M. Chang, P.-C. Hsiu, Y.-H. Chang, and C.-W. Chang, "A resource-driven dvfs scheme for smart handheld devices," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3, pp. 1–22, 2013.

[3] M. Ju, H. Kim, M. Kang, and S. Kim, "Efficient memory reclaiming for mitigating sluggish response in mobile devices," in *2015 IEEE 5th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*, pp. 232–236, IEEE, 2015.

[4] W. Guo, K. Chen, H. Feng, Y. Wu, R. Zhang, and W. Zheng, "$mars$: Mobile application relaunching speed-up through flash-aware page swapping," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 916–928, 2015.