MASTER D'INFORMATIQUE
PARCOURS SIRIS
SCIENCE ET INGÉNIERIE DES RÉSEAUX, DE L'INTERNET ET DES SYSTÈMES

Travail d'Étude et de Recherche

Florian HALM
florian.halm@etu.unistra.fr

ENVIRONNEMENT DE MESURE ET REPRODUCTION D'UNE EXPÉRIMENTATION SUR MOBILE

2 mai 2024

TER encadré par

Pierre DAVID
pda@unistra.fr

Table des matières

Chapitre 1

Introduction

Les téléphones mobiles ont une place importante dans la société actuelle du fait des nombreux services qu'ils offrent à leurs utilisateurs. Selon des statistiques récentes, plus de 60 % de la population mondiale en possède un [1] bien qu'il y ait quelques disparités dans les pays les plus pauvres.

Les deux systèmes d'exploitation qui dominent le marché des mobiles sont Android (71 %) et iOS (27 %) [2]. Les pourcentages restants sont partagés par d'autres systèmes d'exploitation très peu utilisés comme Windows Phone ou Tizen de Samsung. Bien que concurrents, Android et iOS sont liés par le fait qu'ils soient tous les deux basés sur un système de type Unix.

Android est basé sur Linux, un système d'exploitation qui équipe la majorité des serveurs dans le monde [3]. Bien que ces deux systèmes d'exploitation soient très proches, il ne faut cependant pas croire que les mobiles et les ordinateurs ont les mêmes problématiques. En effet, les mobiles sont moins puissants que les ordinateurs, ce qui est encore plus vrai pour les smartphones bas de gamme que beaucoup de personnes choisissent pour leur prix attractif. Cette différence de puissance peut avoir un impact au niveau du système d'exploitation : par exemple, certains algorithmes de gestion de la mémoire ou d'ordonnancement des processus fonctionnant très bien sur un ordinateur Linux seront peut-être moins bien adaptés à un mobile Android.

De par leurs différences, il est important d'étudier et de comprendre les caractéristiques des systèmes mobiles pour mieux les utiliser.

Pour aider les développeurs ou plus généralement les personnes travaillant sur des mobiles, Android dispose de métriques variées donnant diverses informations sur l'état du système comme la mémoire ou le processeur mais aussi des métriques plus générales comme le temps de lancement d'une application ou le pourcentage de batterie restant du mobile. Ces métriques sont mesurables via de nombreux types d'outils divisés en plusieurs catégories selon le type de mesure à réaliser.

En étudiant ces métriques et les outils de mesure, il sera possible d'optimiser les performances des mobiles ou des applications développées pour pouvoir rendre l'expérience utilisateur la meilleure possible.

L'objectif de mon travail a été d'une part de réaliser certaines des mesures de la littérature ([4], [5], [6], [7]) en m'assurant qu'elles soient facilement reproductibles et d'autre part de comparer les résultats de mes mesures avec ceux de la littérature pour vérifier leur cohérence. Pour cela, je vais tout d'abord lister les différentes métriques que possède un système Android et les outils actuellement et précédemment utilisés pour calculer ces métriques. Ensuite, je vais réaliser des scénarios de mesure réalistes issus d'articles de recherche en mettant un accent sur la reproductibilité, pour enfin conclure et faire une mise en perspective.

Chapitre 2

État de l'art

Beaucoup de chercheurs s'intéressent à l'optimisation des performances des systèmes Android. Par exemple, [8] veut réduire le temps de lancement des applications, [9] expose le fait que la gestion de la mémoire réalisée par Linux n'est pas forcément adaptée à un système Android, ou alors [10] qui utilise un modèle d'intelligence artificielle pour déterminer quelle sera la prochaine application utilisée pour optimiser les ressources matérielles.

Pour mesurer l'apport des outils qu'ils développent, ces auteurs utilisent les métriques proposées par Android. Il faut cependant noter que de nombreuses contributions restent vagues sur la manière dont les mesures sont réalisées. Certains conçoivent même leurs propres outils de mesure sans forcément les rendre accessibles : cela empêche la reproductibilité et le lecteur peut remettre en doute la fiabilité des mesures.

Mon état de l'art consiste en les différentes métriques d'un système Android et les outils existant qui permettent de réaliser leur mesure.

2.1 Métriques

Un système Android possède de nombreuses métriques accessibles via plusieurs outils donnant des informations détaillées sur l'état du système. Par exemple, voici certaines métriques des principaux composants matériels d'un mobile :

- CPU : fréquence et utilisation de chaque cœur, pourcentage d'utilisation du CPU par les processus.
- Mémoire : utilisation de la mémoire de chaque processus, obtenir des informations sur la mémoire virtuelle.
- GPU : fréquence du GPU, utilisation de la mémoire du GPU de chaque processus.
- Batterie : batterie restante en pourcentage et en microampère-heures (μAh), courant instantané en microampère (μA).

Il existe également beaucoup d'autres métriques : réseau, événements de toucher, température des composants matériels, rendu d'image... Étant donné l'abondance du nombre de métriques d'un système Android, il est impossible de toutes les lister dans ce texte. Je vais cependant lister précisément les métriques que j'ai utilisées pour réaliser mes mesures :

1. mesure du temps de démarrage d'une application [11]. Celui-ci peut être mesuré selon deux métriques :
   - la métrique *Time To Initial Display* (TTID) mesure le temps pour qu'une application produise sa première image. Cette métrique est présente pour n'importe quelle application et est calculée automatiquement sans avoir besoin de modifier son code

source.
   - la métrique *Time To Full Display* (TTFD) est optionnelle, c'est-à-dire qu'elle n'est pas calculée par défaut. Elle représente le temps qui s'est écoulé entre le démarrage de l'application et un appel à la méthode *reportFullyDrawn ()* [12] qui permet d'indiquer au système que l'application est entièrement affichée. C'est donc au développeur de l'application de décider quand placer cet appel dans le code, c'est-à-dire ni trop tôt et ni trop tard pour ne pas fausser la valeur du TTFD.

   J'ai choisi de mesurer le TTID car cette métrique existe pour toutes les applications et elle ne peut pas être faussée par un mauvais choix du développeur comme avec le TTFD.
2. mesure de l'utilisation du CPU (en pourcentage) selon un nombre donné d'applications en arrière-plan pour savoir si ces applications sont gourmandes en CPU
3. mesure de la quantité de mémoire utilisée et du nombre d'octets reçus via le réseau par une application (YouTube) pour voir l'impact qu'a le contenu et la qualité de la vidéo jouée
4. mesure du *Proportional Set Size* (PSS) maximum de plusieurs applications. Le PSS indique le nombre de pages non partagées plus un partage équitable des pages partagées utilisées par une application. Par exemple, avec 2 processus A et B qui utilisent une bibliothèque partagée de 30 pages et qui possèdent respectivement 2 et 4 pages non partagées, on divise 30 pages par 2 processus et on obtient 15 pages : le PSS indiquera 15 + 2 = 17 pages pour le processus A et 15 + 4 = 19 pages pour le processus B.

2.2 Outils

Pour effectuer les mesures de métriques, de nombreux outils sont disponibles et utilisés dans la littérature. Ceux-ci peuvent être séparés en trois catégories (figure 2.1) :
- mesure directe des métriques via des commandes ou des scripts
- traçage
- profilage

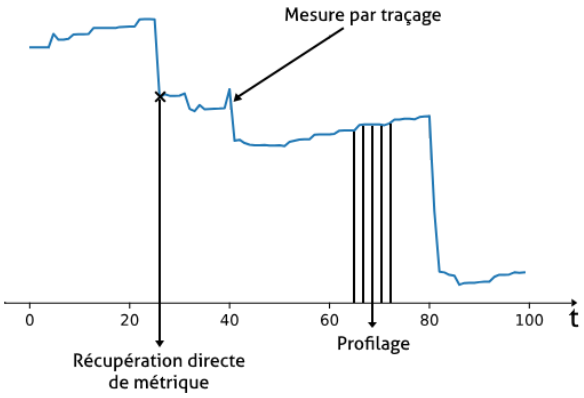

FIGURE 2.1 – Différences entre les trois catégories d'outils

Bien que je n'aie uniquement utilisé la première catégorie d'outil pour réaliser mes mesures, par souci d'exhaustivité, j'ai également décidé de présenter les deux autres catégories car elles sont utilisées dans la littérature.

2.2.1 Récupérer les métriques avec des commandes ou des scripts

Un outil proposé par Android est l'*Android Debug Bridge* (ADB) [13]. Cet outil fonctionne en invite de commandes et permet à un ordinateur de communiquer avec un appareil Android par

USB ou Wi-Fi pour réaliser diverses actions comme installer, lancer ou tuer des applications, ou récupérer des informations sur les métriques du mobile. Par exemple, [4] utilise ADB pour réaliser les mesures de sa figure 14 et [5] de sa figure 11.

Pour récupérer des informations sur les métriques du mobile, ADB propose plusieurs méthodes :

- la première est l'outil *dumpsys* [14] qui permet l'obtention d'informations sur plusieurs catégories de métriques grâce à différents services (*dumpsys cpuinfo*, *dumpsys meminfo*, *dumpsys wifi*, *dumpsys battery*...). 260 services sont proposés en tout par dumpsys.
- la deuxième méthode s'appuie sur le fait qu'Android est basé sur Linux. Ce dernier propose beaucoup de métriques accessibles via des commandes classiques comme *cat /proc/vmstat* pour obtenir des informations sur la mémoire virtuelle, ou *top* pour obtenir des informations sur l'état du système et ses processus. Un système Android intègre également ces métriques et une majorité des commandes fonctionnant dans une invite de commandes Linux sont accessibles via ADB.

Comme ADB fonctionne avec des commandes, il est également possible de récupérer la valeur des métriques via des scripts shell dans lesquels on peut fixer la valeur des paramètres des scénarios de mesure (nombre de répétions des événements, options des commandes...).

2.2.2 Outils de traçage

Les outils de traçage constituent un deuxième type d'outil pour effectuer des mesures de métriques. Un outil de traçage réalise une trace, c'est-à-dire génère un fichier au format *.json*, *.systrace*, *.html* ou *.txt* contenant l'évolution des valeurs de différentes métriques du système au cours du temps sur une période donnée. Cette trace est ensuite visualisée par un visualiseur de traces qui rendra une représentation visuelle très détaillée de l'évolution de chacune des métriques. L'utilisateur choisit sur quelles métriques il souhaite capturer la trace et la durée de celle-ci.

Beaucoup d'outils permettent la capture et/ou la visualisation de traces d'un système Android.

Il est possible de capturer une trace de trois façons différentes :

- via une invite de commandes sur un ordinateur
- via le mobile
- via une interface web

Systrace [15] est un outil maintenant obsolète qui permettait la capture de traces d'un système Android branché sur un ordinateur. Cet outil fonctionnait en invite de commandes grâce à laquelle on renseignait les métriques que l'on souhaitait mesurer. Par exemple, [5] utilise Systrace pour réaliser les mesures de ses figures 1 et 8 et [7] pour sa figure 3.

Si l'on souhaite capturer une trace directement sur le mobile, il suffit d'activer le mode développeur dans l'application « Settings » et de débuter la capture via l'onglet *System Tracing* [16]. Le fonctionnement et les capacités de cet outil sont extrêmement proches de Systrace.

Le dernier moyen pour capturer une trace est via une interface web. C'est ce que propose l'outil *Perfetto* [17], une interface web permettant à la fois la capture et la visualisation de traces. Cet outil a été développé par Google [18] et est l'outil le plus récent qu'il est conseillé d'utiliser. Par exemple, [5] utilise Perfetto pour réaliser ses mesures que j'ai reproduites dans mon scénario 2.

Avant que Perfetto ne soit développé, il existait deux façons de visualiser des traces :

- si la trace est au format *.html*, il est possible de la visualiser avec un navigateur Web.
- si la trace est au format *.systrace* ou *.json*, le projet Catapult [19] développé par Google était accessible via le navigateur Google Chrome : *chrome://tracing/*. Cet outil pouvait

enregistrer des traces de Google Chrome et visualiser des traces dans les formats énoncés [20].

2.2.3 Outils de profilage

Après la récupération de métriques par commandes et les outils de traçage, les outils de profilage permettent de mesurer par échantillonage régulier l'usage d'une ressource par une application donnée.

Parmi les outils de profilage que propose un système Android, on peut citer : *CPU profiler*, *Memory profiler*, *Energy profiler*, *Power profiler* [21]. Ces outils de profilage sont tous issus d'*Android Studio*, un environnement de développement permettant le développement d'applications Android [22]. Comme leur nom l'indique, chacun de ces outils permet de mesurer l'usage qu'a notre application de l'un des composants matériels choisis. Pour réaliser un profilage du GPU, il existe l'outil *Android GPU Inspector* [23] qui n'appartient pas à Android Studio.

Une restriction importante de ce type d'outil est le fait qu'il faut posséder le code source de l'application dont on réalise les mesures. Cela est principalement dû à deux raisons :

— ces outils sont accessibles via Android Studio qui requiert le code source de l'application pour être utilisé.

— il est possible d'utiliser ce type d'outil en instrumentant le code source de l'application [24] pour y indiquer exactement ce que l'on souhaite mesurer. Pour cela, il faut utiliser la classe *Debug* fournie par Android [25] et y indiquer exactement les moments de début et de fin du profilage : il est donc nécessaire de disposer du code source de l'application si on veut utiliser ces fonctionnalités. Dans le cas où on réalise le profilage d'une application en instrumentant son code, il existait un ancien outil du nom d'*Android Traceview* [26] qui est maintenant obsolète. Il permettait de visualiser les résultats du profilage de l'application. Par exemple, [4] a utilisé cet outil dans sa figure 3.

Ces outils sont donc plus limités par rapport aux outils de traçage qui peuvent mesurer toutes les métriques voulues du système dans son intégralité. Cependant, les outils de profilage ont un avantage majeur : comme ils opèrent par échantillonnage par rapport aux outils de traçage qui analysent un système à tout instant sur la période de la trace, les outils de profilage stockent beaucoup moins de données, donc leurs résultats sont moins volumineux [27].

2.2.4 Autres outils spécifiques

Bien qu'il existe déjà de nombreux outils pour réaliser des mesures, certains chercheurs développent leurs propres outils pour qu'ils soient exactement adaptés à leurs besoins. Malheureusement, beaucoup de ces outils ne sont pas rendus publics ce qui limite grandement la reproductibilité des mesures. Par exemple, [5] a modifié le code source Android pour récupérer les informations nécessaires pour les mesures, ou bien [4] et [6] utilisent leurs propres outils de traçage pour réaliser certaines de leurs mesures : aucun de ces articles n'a donné d'informations sur comment se procurer leur outil ou voir les modifications du code source.

De plus, bien que les outils utilisés soient clairement énoncés, si la méthode de mesure d'une métrique n'est pas bien expliquée, cela peut également réduire la reproductibilité. Par exemple, [5] a effectué la mesure des Frames Per Second (FPS) d'applications en utilisant l'outil Systrace. Cependant, la méthode utilisée n'a pas été clairement expliquée : je voulais réaliser ces mesures de FPS dans le cadre de mon TER mais, n'ayant pas trouvé la méthode qu'a utilisé [5], cela m'a malheureusement été impossible.

Chapitre 3

Mesures

Ce chapitre se concentre sur la réalisation de scénarios de mesure issus de différents articles publiés dans la littérature. La réalisation de mes mesures a plusieurs objectifs :

1. vérifier si j'obtiens des résultats similaires aux articles du fait des méthodes de mesure et du matériel différents ;
2. réaliser des mesures supplémentaires sur des cas que je juge intéressants et qui n'ont pas été traités par les articles ;
3. utiliser et documenter une méthode de mesure facilement reproductible.

Pour le dernier point, j'ai choisi l'outil ADB (version 1.0.39 sur Ubuntu 20.04.6 LTS) en utilisant selon les cas des scripts (disponibles sur https://git.unistra.fr/fhalm/ter-416-scripts) ou directement des commandes dont des explications détaillées sur leur fonctionnement se trouve en annexe B. Comme je souhaite obtenir les valeurs de certaines métriques spécifiques, l'outil ADB est le plus adapté. En effet, un outil de traçage donnerait trop d'informations et ne permettrait pas d'isoler une métrique précise, et un outil de profilage ne fonctionnerait pas car les mesures peuvent être réalisées uniquement sur des applications dont on possède le code source, or je veux réaliser mes mesures avec des applications du Google Play Store dont je n'ai pas le code source.

Pour représenter les figures de mes mesures réalisées, j'ai choisi d'utiliser la bibliothèque *matplotlib* du langage de programmation Python du fait de sa polyvalence et de sa documentation fournie.
Les figures provenant d'articles scientifiques que j'ai utilisées dans mon mémoire sont encadrées d'un cadre gris pour les différencier de mes figures.

Pour réaliser mes mesures, j'ai utilisé le mobile suivant :
— *Samsung Galaxy S9* 2018 (SM-G960F)
— 4 Go de RAM
— CPU : *Exynos 9 Octa 9810* à 2.70 GHz avec 8 cœurs
— Android 10 (API Level 29)
— 64 Go de stockage
— Noyau Linux 4.9.118

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

Je réalise ce scénario dans le but de reproduire les mesures qui ont été réalisées dans [4] dans leur figure 2. Les auteurs ont développé un outil nommé MARS pour accélérer le temps de lancement d'applications Android. Ils ont réalisé une mesure dans leur figure 2 pour montrer le temps de lancement d'applications sans leur outil.

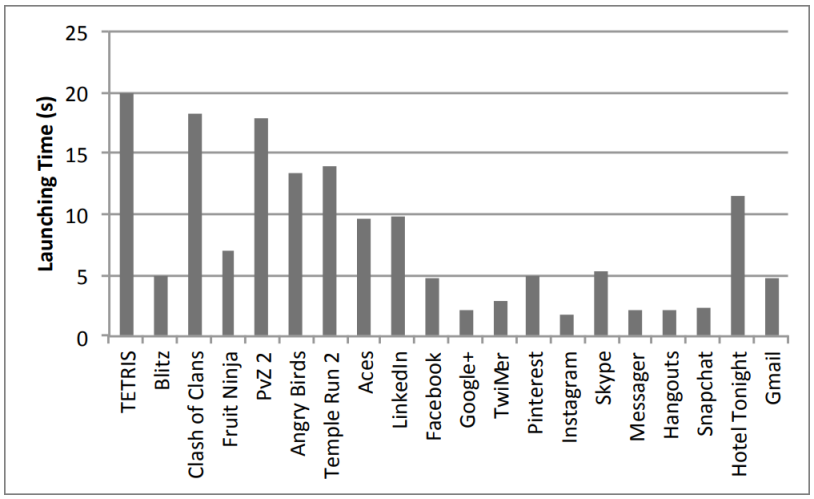

FIGURE 3.1 – Launching time of some popular applications on Android
(Figure 2 extraite de [4])

Dans leur figure 11, les auteurs montrent l'amélioration du temps de lancement des applications avec leur outil.

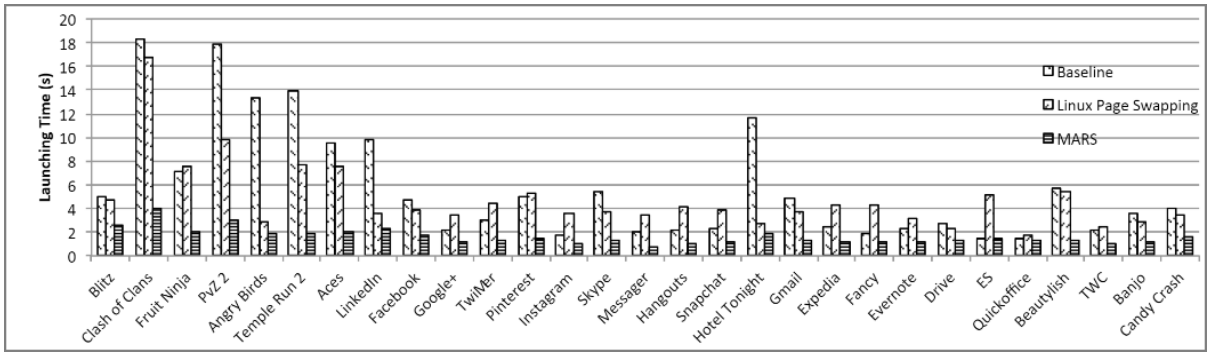

FIGURE 3.2 – Relaunching time under Linux page swapping and MARS
(Figure 11 extraite de [4])

Je vais mesurer le temps de démarrage (cold start et hot start) de 6 applications parmi les 20 indiquées dans la table 3 de [5] sans qu'il n'y ait aucune application en arrière-plan.

Une application est lancée en *cold start* 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 quand elle avait été terminée précédemment par le système. Ce dernier va devoir réaliser de nombreuses tâches pour lancer une application en *cold start* comme créer le processus de l'application, de l'activité et du thread principal.

Une application est lancée en *hot start* 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 la ramener au premier plan.

Pour réaliser mes mesures, j'ai créé deux scripts : `script_startup_cold.sh` et `script_startup_hot.sh` pour mesurer respectivement le temps d'un *cold start* et d'un *hot start* d'une application.

À l'inverse de [4] qui lance chaque application 10 fois en *cold start* et en *hot start*, mes scripts lancent chaque application 100 fois pour améliorer la fiabilité des résultats et calculer des intervalles de confiance à 95 %. Une moyenne des 100 lancements est réalisée pour obtenir la durée moyenne.

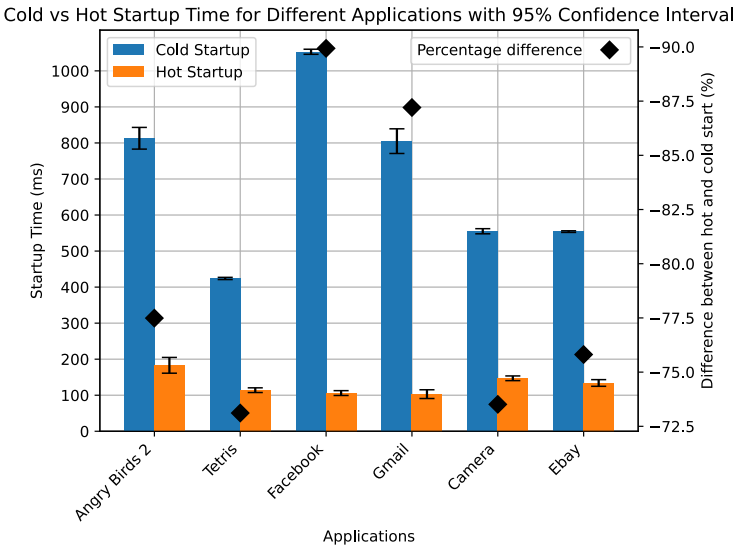

FIGURE 3.3 – Cold start et Hot start de différentes applications

Avec la figure 3.3, j'observe que selon les applications, leurs temps de lancement (*cold start* ou *hot start*) ne sont pas les mêmes : par exemple l'application Angry Birds 2 a un *cold start* deux fois plus long que l'application Tetris alors même que les deux applications utilisent toutes les deux Unity [28], un moteur de jeu : cela montre que la technologie utilisée par une application n'est donc pas le seul facteur qui agit sur son temps de lancement. Un élément surprenant est que l'application Facebook est celle qui possède le plus long *cold start* bien qu'elle ne soit pas un jeu vidéo gourmand en ressources. La même observation peut être faite pour l'application Gmail.

En comparant mes mesures avec la figure 3.1 (figure 2 de [4]) pour les applications Tetris, Facebook et Gmail, j'observe que :

— pour Tetris, on passe de 20 secondes de temps de lancement en cold start à 424 ms ;
— pour Facebook, on passe de 5 secondes de temps de lancement en cold start à 1053 ms ;
— pour Gmail, on passe de 5 secondes de temps de lancement en cold start à 805 ms.

Les grandes différences entre mes mesures et la figure 3.1 peuvent être expliquées par deux facteurs : principalement par l'évolution de la puissance des mobiles entre 2014 et aujourd'hui, mais aussi qu'en 10 ans ([4] date de 2014), les applications ont sûrement été optimisées donc leurs temps de lancement ont peut-être été réduits.

Comme on pouvait s'en douter, une application lancée en *cold start* met beaucoup plus de temps à s'afficher qu'en *hot start*. En effet, une application lancée en *hot start* se lancera entre 72.5 et 90 % (indiqué par le losange sur la figure 3.3) plus vite qu'en *cold start*. Cela provient du fait qu'une application lancée en *cold start* doit créer son processus, lui allouer de la mémoire et le charger, alors qu'en *hot start*, toutes les données de l'application sont déjà en mémoire. J'observe également que le pourcentage de différence entre *cold start* et *hot start* est assez homogène pour toutes les applications mesurées.

3.2 Scénario 2 : utilisation du CPU

Dans l'optique de montrer que les applications en arrière-plan ne sont pas grandes consommatrices du CPU en général, la table 3.1 (table 1 de [5]) expose les résultats de la mesure de l'utilisation du CPU :

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%

TABLE 3.1 – CPU utilization with N ($0 \sim 8$) apps in the BG.
(Table 1 extraite de [5])

Pour vérifier si j'obtiens la même conclusion que [5] sur la consommation CPU des applications en arrière-plan, je réalise ce scénario dans lequel je mesure avec mon script `script_cpu_all.sh` l'utilisation du CPU selon le nombre d'applications en arrière-plan : ces applications sont choisies au hasard parmi les 20 indiquées dans la table 3 de [5]. Comme dans l'article original, 5 cas sont mesurés : 0, 2, 4, 6 et 8 applications en arrière-plan.

Contrairement à [5] qui effectue 10 mesures pour chaque cas, je réalise les mesures 100 fois pour chaque cas pour améliorer la fiabilité des résultats et calculer des intervalles de confiance à 95 %. J'obtiens ainsi 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 %	51.0 %
6	32.3 %	53.3 %
8	36.9 %	50.7 %

TABLE 3.2 – Utilisation moyenne et maximale du CPU pour chaque cas

Avec la table 3.2, je 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 (autour de 50 %). Cela montre qu'à partir de 2 applications en arrière-plan, augmenter le nombre d'applications en arrière-plan n'impacte pas le pic d'utilisation du CPU.

Je remarque également qu'à partir de 2 applications en arrière-plan, l'utilisation moyenne du CPU augmente d'environ 12 % à chaque fois que 2 applications sont ajoutées en plus en arrière-plan : cette augmentation est donc linéaire sur les cas étudiés.

En représentant les valeurs des tables 3.1 et 3.2 sur les figures 3.4 et 3.5 respectivement, il est possible de voir une différence majeure pour le cas avec 0 application en arrière-plan :

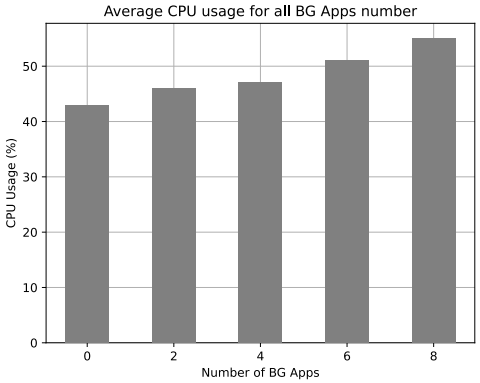

FIGURE 3.4 – Utilisation moyenne du CPU pour chaque cas (données de la table 3.1 de [5])

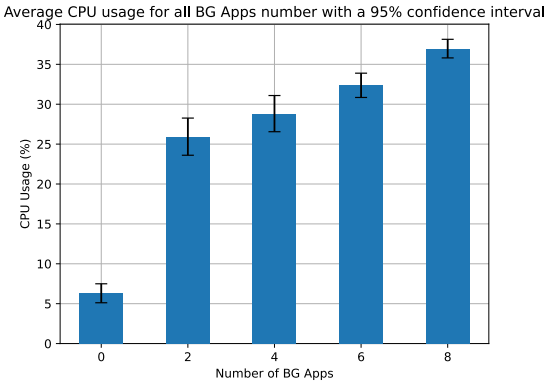

FIGURE 3.5 – Utilisation moyenne du CPU pour chaque cas

Sur la figure 3.5, j'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). Cela est différent de la figure 3.4 qui possède une augmentation de seulement 7 % : par manque d'élément, cette différence ne peut pas être expliquée.

On peut remarquer que toutes mes mesures sont inférieures à celles de la table 3.1 (table 1 de [5]). Cela peut venir du fait que mon mobile n'est pas le même que celui utilisé dans [5] (HUAWEI P20) et que les performances des deux mobiles sont différentes : il est donc compliqué de comparer directement mes résultats avec ceux de la table 3.1.

Cependant, une comparaison des ordres de grandeur est possible. Mes valeurs mesurées semblent cohérentes pour plusieurs raisons :

— les mesures dans le cas Average sont strictement croissantes comme dans la table 3.1
— 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 la table 3.1, l'augmentation moyenne de la consommation CPU est de 6.1 %. Cela représente une différence de 6 points de pourcentage ce qui ne semble pas être un écart significatif.

La plus grande différence apparaît dans le cas avec 0 application en arrière-plan. En effet, le CPU de mon mobile est utilisé à 6.3 % en moyenne alors que dans la table 3.1, le CPU du mobile est utilisé à 43 %. Bien que les opérations réalisées par les deux mobiles soient identiques, il y a une différence de 36.7 points de pourcentage : cela illustre bien les différences de performance du CPU entre les deux mobiles.

Pour conclure, on peut voir dans la table 3.2 et la figure 3.5 qu'augmenter le nombre d'applications en arrière-plan entraîne une augmentation presque linéaire de l'utilisation moyenne du CPU. Le pic d'utilisation 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 donc la même conclusion que [5].

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

Dans [6], les auteurs voulaient comprendre le modèle d'utilisation des ressources des applications mobiles. Pour cela, ils ont développé un outil nommé DVFS permettant de réduire la consommation en énergie en ajustant la fréquence du CPU selon l'application en premier plan. La figure 3.6 (figure 2 de [6]) a pour objectif d'analyser l'usage des ressources d'une application (YouTube) en montrant son usage du CPU, de la mémoire et du réseau :

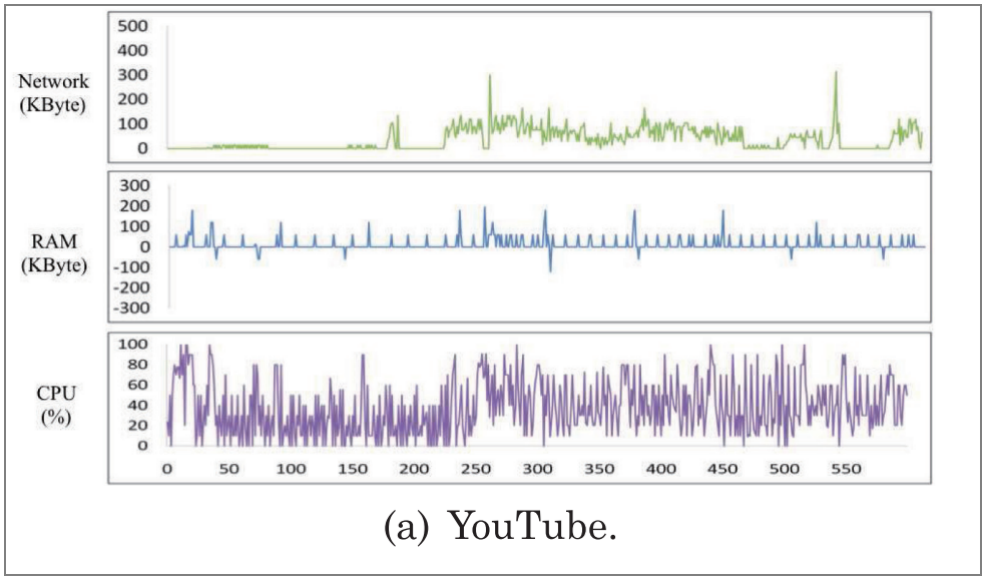

FIGURE 3.6 – Resource usage patterns.
(Figure 2 (a) extraite de [6])

Les méthodes de mesure de la figure 3.6 ne sont pas claires. Par exemple, que signifie une quantité négative de RAM ? De plus, les auteurs n'ont pas indiqué quels types de vidéos ont été utilisées pour leurs mesures.

Ce scénario de mesure a pour objectif de reproduire et d'approfondir les mesures de la figure 3.6 en y ajoutant des cas intéressants pour conforter ma méthode de mesure comme des types ou des qualités de vidéos différentes. Je mesure l'utilisation de la mémoire, du CPU et le nombre d'octets reçus par l'interface réseau *wlan0* de mon mobile (communications Wi-Fi) pour une application donnée (YouTube). Cette dernière est en premier plan avec une vidéo qui se joue sans aucune application en arrière-plan.

Pour réaliser ces mesures, j'ai créé trois scripts : `script_memory_RSS.sh`, `script_cpu_one_ps.sh` et `script_network.sh` qui mesurent respectivement l'utilisation de la mémoire, du CPU et du réseau chaque seconde par le processus pendant 100 secondes dans 2 cas différents :

- différence d'utilisation de la mémoire, du CPU et du réseau entre une vidéo d'une course de formule 1 (F1) en qualité 360p et la même vidéo en qualité 1080p ;
- différence d'utilisation de la mémoire, du CPU et du réseau entre une vidéo où l'image est toujours noire (image fixe peu coûteuse à afficher), une vidéo "qui bouge beaucoup" (vidéo de course de F1 comme dans le premier cas car elle possède beaucoup de changements de plans de caméra) et un dessin animé (couleurs uniformes moins gourmandes à afficher que des images réalistes).
  Toutes ces vidéos sont en 1080p.

3.3.1 Cas d'une vidéo en 360p par rapport à la même vidéo en 1080p

3.3.1.1 Mémoire

Voyons si l'usage de la mémoire est plus élevé si la vidéo est de meilleure qualité.

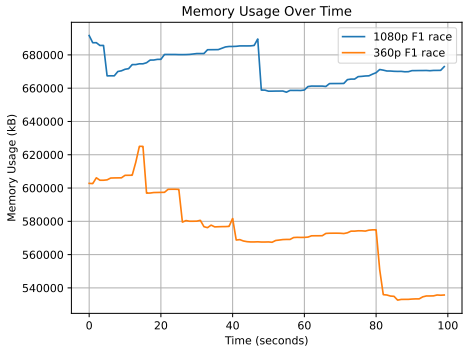

FIGURE 3.7 – Évolution de l'utilisation de la mémoire pour chaque cas

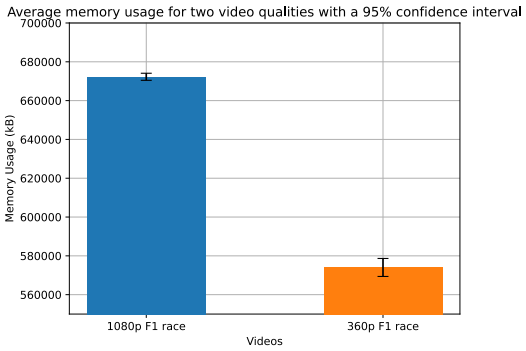

FIGURE 3.8 – Utilisation moyenne de la mémoire pour chaque cas

On peut voir sans surprise sur la figure 3.7 que la vidéo en 1080p consomme plus de mémoire que la même vidéo en 360p. On remarque également que pour les deux vidéos, l'usage de la mémoire baisse brusquement pour continuer à remonter par la suite. Cela est peut-être dû à la mémoire tampon de la vidéo qui est pleine, donc que le système a suffisamment stocké de données et qu'il doit attendre que le tampon se vide pour le remplir à nouveau.

Avec la figure 3.8, j'observe une augmentation de 17.1 % de l'utilisation moyenne de la mémoire entre le cas 1080p et 360p. Cette augmentation ne semble pas être corrélée à la différence du nombre de pixels entre les deux vidéos. En effet, bien qu'une vidéo en 1080p possède 9 fois plus de pixels qu'une vidéo en 360p, l'augmentation de l'utilisation de la mémoire entre les deux cas ne suit pas cet ordre de grandeur.

On peut conclure que la qualité de la vidéo a un impact sur sa consommation mémoire : une vidéo de meilleure qualité (ici 1080p) consomme plus de mémoire que la même vidéo dans une qualité inférieure (ici 360p).

3.3.1.2 CPU

Voyons si l'utilisation du CPU varie selon la qualité de la vidéo.

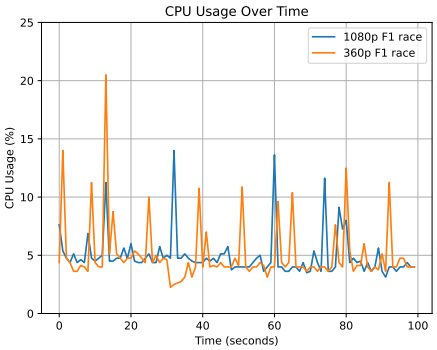

FIGURE 3.9 – Évolution de l'utilisation du CPU pour chaque cas

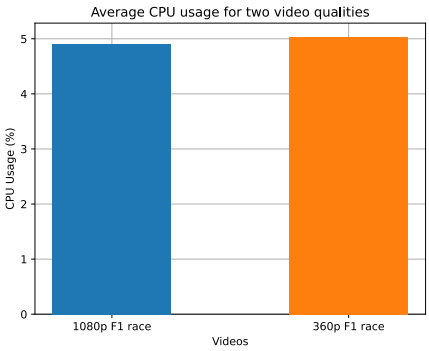

FIGURE 3.10 – Utilisation moyenne du CPU pour chaque cas

Avec la figure 3.9, on peut voir qu'il n'y a pas de différence significative dans l'évolution de la consommation du CPU pour les deux vidéos. En effet, j'observe qu'en moyenne, il y a une différence de 2.5 % de l'utilisation du CPU (figure 3.10) ce qui n'est pas significatif. On remarque également que la consommation du CPU est très variable dans les deux cas.

Les figures 3.9 et 3.10 permettent de conclure que la qualité de la vidéo ne joue pas de rôle significatif sur sa consommation du CPU : cette dernière reste proche pour les deux qualités de vidéo bien qu'elle soit très variable.

3.3.1.3 Consommation réseau

Voyons si la consommation réseau est plus élevée si la vidéo est de meilleure qualité.

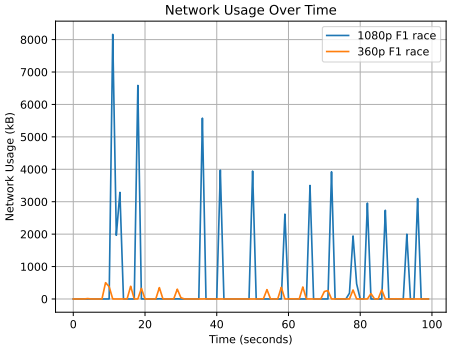

FIGURE 3.11 – Évolution de la consommation réseau pour chaque cas

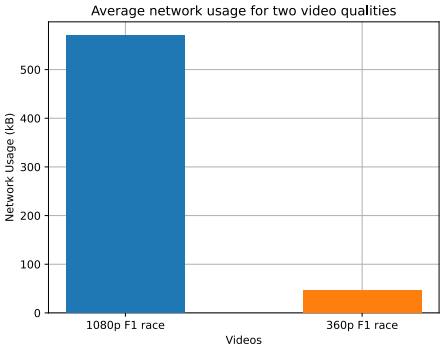

FIGURE 3.12 – Consommation réseau moyenne pour chaque cas

J'observe avec la figure 3.11 que la consommation réseau de la vidéo en 1080p au cours du temps est largement supérieure à la même vidéo en 360p, ce qui n'est là encore pas une surprise. Il y a une augmentation de 1136.4 % de la consommation réseau entre le cas 1080p et 360p (figure 3.12). Celle-ci est sûrement liée au fait que l'augmentation du nombre de pixels entre une vidéo 1080p et une vidéo 360p est de 800 % : cette valeur est proche de l'augmentation de la consommation réseau mesurée (1136.4 %).

On remarque également que la consommation réseau pour les deux cas est très variable. En effet, la figure 3.11 est principalement composée de pics d'utilisation réseau. Cela est confirmé par les écarts types qui sont supérieurs aux valeurs des moyennes (écart type = 1500.94 Ko pour le cas 1080p, écart type = 115.64 Ko pour le cas 360p).

On peut conclure que la qualité de la vidéo joue un rôle majeur dans sa consommation réseau : une vidéo d'une meilleure qualité aura une plus grande consommation réseau que la même vidéo dans une qualité plus basse ce qui est un résultat attendu. Cela peut être expliqué par la différence du nombre de pixels entre les deux vidéos qui est en corrélation avec l'augmentation de la consommation réseau mesurée.

3.3.2 Cas pour trois types de vidéos de même qualité (1080p)

3.3.2.1 Mémoire

Voyons si l'usage de la mémoire est plus élevé si la vidéo possède un contenu plus coûteux à afficher graphiquement.

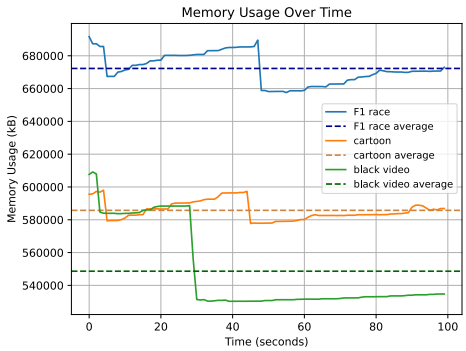

FIGURE 3.13 – Évolution de l'utilisation de la mémoire pour chaque cas

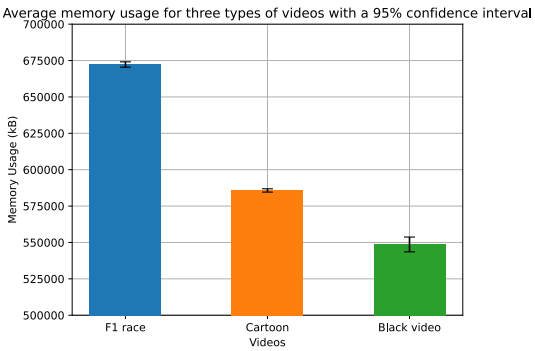

FIGURE 3.14 – Utilisation moyenne de la mémoire pour chaque cas

La figure 3.13 montre que la consommation mémoire des trois types de vidéos évolue d'une manière similaire. En effet, on observe dans les 3 cas :

- une baisse au début des mesures, donc au lancement de la vidéo ;
- puis une augmentation ;
- et une chute brutale ;
- suivie d'une augmentation jusqu'à la fin des mesures.

Cette chute brutale de la consommation mémoire peut sûrement venir de la mémoire tampon qui est pleine, comme souligné dans le cas précédent avec les vidéos de même qualité.

Avec la figure 3.14, on observe que plus une vidéo a de variation dans ses images, plus elle consomme de la mémoire (la course de F1 consomme 22.5 % de plus que la vidéo noire).

On peut conclure que le contenu de la vidéo a également un impact sur sa consommation mémoire : si le contenu de la vidéo est plus gourmand graphiquement, sa consommation mémoire est plus importante.

3.3.2.2 CPU

Voyons si l'utilisation du CPU varie selon le contenu de la vidéo.

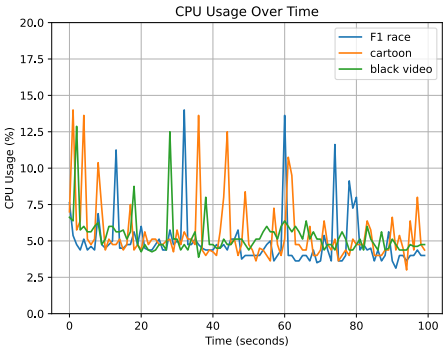

FIGURE 3.15 – Évolution de l'utilisation du CPU pour chaque cas

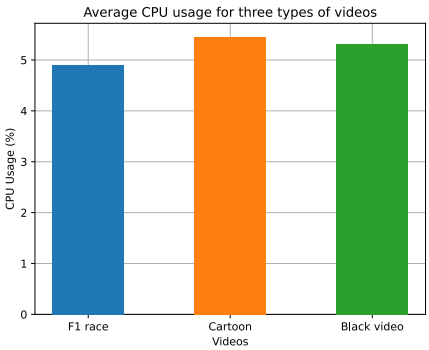

FIGURE 3.16 – Utilisation moyenne du CPU pour chaque cas

On peut voir avec la figure 3.15 qu'il n'y a pas de différence significative dans l'évolution de la consommation du CPU pour les trois types de vidéos. On peut également remarquer que la consommation CPU est très variable dans les trois cas.

Avec la figure 3.16, on observe que le contenu d'une vidéo n'influe pas sur la consommation moyenne du CPU. En effet, pour les trois types de vidéos mesurées, la consommation du CPU

est proche de 5%.

Comme dans le premier cas avec la qualité, le contenu d'une vidéo ne provoque pas de différence significative sur l'utilisation du CPU.

3.3.2.3 Consommation réseau

Voyons si la consommation réseau est plus élevée si la vidéo possède un contenu plus coûteux à afficher graphiquement.

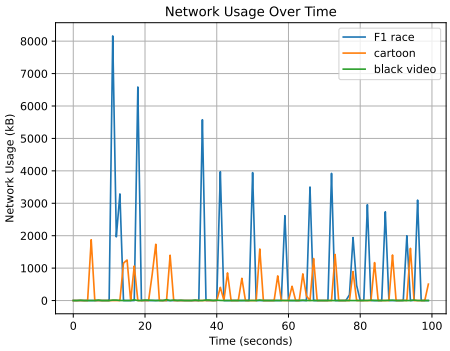

FIGURE 3.17 – Évolution de la consommation réseau pour chaque cas

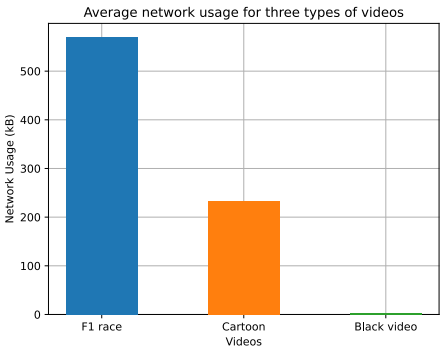

FIGURE 3.18 – Consommation réseau moyenne pour chaque cas

Avec la figure 3.18, j'observe que la consommation réseau de la vidéo de course de F1 est largement supérieure au dessin animé ou à la vidéo noire. Comme avec la mesure de la mémoire, cela vient du fait que la course de F1 est plus gourmande graphiquement. On remarque également que la vidéo noire a une consommation réseau presque nulle (3 Ko en moyenne) car son image est toujours fixe et qu'elle ne possède aucun son : aucune donnée ne doit donc être récupérée via le réseau pour visionner cette vidéo.

Comme pour les vidéos de F1 de qualité différente en 3.3.1.3, on constate que la consommation réseau est très variable. En effet, la figure 3.17 est principalement composée de pics d'utilisation réseau (hormis la vidéo noire qui a une consommation réseau presque nulle). La variabilité de la consommation réseau de la course de F1 et du dessin animé est confirmée par leurs écarts types qui sont supérieurs aux valeurs de leurs moyennes (écart type = 1500.94 Ko pour la vidéo de F1, écart type = 396.04 Ko pour le dessin animé).

Comme pour la mémoire, et à l'opposé de la consommation CPU, on peut conclure 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 possède des images qui changent souvent et qui sont coûteuses à afficher par le GPU, sa consommation réseau sera plus importante.

3.3.3 Conclusion

Grâce au premier cas (qualités différentes) et au deuxième cas (contenus différents), on peut conclure que la qualité de la vidéo et son contenu jouent un rôle significatif sur la consommation mémoire et la consommation réseau. En effet, une vidéo d'une qualité plus élevée consommera plus de mémoire et aura une plus grande consommation réseau, de même pour une vidéo plus coûteuse graphiquement. Cependant, la qualité ou le contenu de la vidéo ne jouent pas de

rôle significatif sur la consommation CPU : cette dernière reste stable en moyenne peu importe la qualité ou le contenu de la vidéo.

3.4 Scénario 4 : mesure du PSS maximum d'applications

Dans [7], les auteurs cherchent à montrer que toutes les applications Android ne demandent pas la même quantité de mémoire au maximum pour fonctionner. La table 3.3 (table 1 de [7]) expose les résultats de la mesure du PSS de certaines applications :

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

TABLE 3.3 – The maximum memory (PSS) of famous applications of Google Play on Nexus 5 (Table 1 extraite de [7])

Grâce aux PSS mesurés dans la table 3.3, les auteurs ont réalisé un scénario de mesure supplémentaire qui consiste à lancer des applications alors qu'il reste moins de 100 Mo de mémoire disponible (100 Mo est inférieur au PSS des applications lancées). Cela a été réalisé pour montrer que lorsque le noyau réclame de la mémoire, le temps de lancement des applications est ralenti car de nombreux appels à des fonctions de réclamation mémoire sont réalisés [29].

Je mesure le PSS maximum de certaines applications les plus connues du Google Play Store pour observer l'éventuelle évolution du PSS des applications entre 2015 (année de [7]) et 2024. J'obtiens ainsi le tableau suivant :

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

TABLE 3.4 – PSS maximum pour chaque application

On peut remarquer que tous les PSS que j'ai mesurés dans la table 3.4 sont plus grands que ceux de la table 3.3 (table 1 de [7]). Par exemple, j'ai mesuré un PSS plus grand de 41.0 % pour l'application Facebook et un PSS plus grand de 100.2 % pour l'application Amazon Shopping.

Ces différences de PSS entre mes mesures et celles de la table 3.3 (table 1 de [7]) peuvent venir du fait que mes mesures ont été réalisées en 2024 alors que celles de [7] ont été réalisées en 2015 : selon [30], 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. J'observe également cette augmentation avec les applications du Google Play Store, tout particulièrement avec Cooking Fever et Amazon Shopping qui ont triplé leur consommation mémoire. N'ayant pas accès aux applications de 2015, je formule l'hypothèse qu'en 9 ans, les applications ont gagné en fonctionnalités ce qui les font donc utiliser plus de mémoire.

Avec ces mesures, on peut conclure que les applications mobiles ne sont pas toutes également gourmandes en mémoire. En effet, selon les mesures réalisées dans la table 3.4, on remarque que parmi les applications choisies, les applications de catégorie Game sont les plus gourmandes (en moyenne : 628 Mo de PSS). Les applications de Shopping et Social ont une consommation mémoire très proche (respectivement 388 Mo et 370 Mo 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). On observe également qu'en 9 ans, le PSS de toutes les applications mesurées a augmenté, ce qui peut être sans doute corrélé avec un ajout de fonctionnalités dans les applications.

Chapitre 4

Conclusion

Nous avons pu constater dans ce mémoire la diversité des métriques des systèmes Android (processeur, mémoire, réseau...), les différents moyens pour les mesurer (récupération directe de métrique, outils de traçage ou de profilage) et leur importance dans l'évolution de ces systèmes. En effet, elles permettent de davantage comprendre ces systèmes pour mieux les utiliser et éventuellement les améliorer. Par exemple, les chercheurs en optimisation des performances des systèmes Android utilisent leurs métriques pour mesurer l'efficacité des outils qu'ils développent.

J'ai pu mettre en œuvre des scénarios de mesure réalistes et reproductibles de certaines métriques en m'inspirant de mesures réalisées dans la littérature. La réalisation de mes mesures m'a apporté beaucoup d'informations sur le fonctionnement interne d'un mobile Android. De plus, j'ai pu obtenir des conclusions intéressantes en comparant les résultats de mes mesures avec ceux des articles.

Mes recherches m'ont montré que de nombreux chercheurs utilisent leurs propres outils pour réaliser leurs mesures ou restent vagues sur la méthodologie utilisée. Cela pose problème car il n'y a aucune garantie sur la justesse, la fiabilité ou l'honnêteté des mesures. Cependant, l'élément le plus problématique est le manque de reproductibilité des mesures. En effet, j'ai été à plusieurs reprises dans l'incapacité de reproduire certains scénarios de mesure réalisés dans des publications car celles-ci n'étaient pas précises sur les métriques ou les outils utilisés.

Du fait du manque de reproductibilité des mesures, j'ai pu apprendre qu'il faut toujours garder un œil critique sur les résultats que les auteurs nous exposent et ne pas aveuglement les accepter. De plus, améliorer la reproductibilité permettrait également de faire avancer la recherche dans ce domaine. En effet, les chercheurs pourraient plus facilement comparer leurs résultats avec d'autres si la même méthodologie est utilisée.

Je tiens à remercier mon encadrant Pierre DAVID pour son aide et ses conseils qui m'ont été d'une grande aide dans la réalisation de mon TER.

Annexe A

Bibliographie

[1] A. Turner, "How many smartphones are in the world ?." https://www.bankmycell.com/blog/how-many-phones-are-in-the-world, 2024.

[2] statcounter, "Mobile operating system market share worldwide." https://gs.statcounter.com/os-market-share/mobile/worldwide, 2024.

[3] Fortune Business Insights, "Server operating system market." https://www.fortunebusinessinsights.com/server-operating-system-market-106601, 2022.

[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.

[5] 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.

[6] 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.

[7] 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.

[8] S. S. Hahn, S. Lee, I. Yee, D. Ryu, and J. Kim, "{FastTrack} : Foreground {App-Aware}{I/O} management for improving user experience of android smartphones," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 15–28, 2018.

[9] Y. Liang, Q. Li, and C. J. Xue, "Mismatched memory management of android smartphones," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[10] S. Zhao, Z. Luo, Z. Jiang, H. Wang, F. Xu, S. Li, J. Yin, and G. Pan, "Appusage2vec : Modeling smartphone app usage for prediction," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1322–1333, IEEE, 2019.

[11] Android, "App startup time." https://developer.android.com/topic/performance/vitals/launch-time.

[12] Android, "Android reportfullydrawn()." https://developer.android.com/reference/android/app/Activity#reportFullyDrawn().

[13] Android, "Android debug bridge (adb)." https://developer.android.com/tools/adb.

[14] Android, "dumpsys." https://developer.android.com/tools/dumpsys.

[15] Android, "Capture a system trace on the command line." https://developer.android.com/topic/performance/tracing/command-line.

[16] Android, "Capture a system trace on a device." https://developer.android.com/topic/performance/tracing/on-device.
[17] "perfetto : System profiling, app tracing and trace analysis." https://ui.perfetto.dev/.
[18] Google, "Github perfetto." https://github.com/google/perfetto.
[19] Google, "Github catapult." https://github.com/catapult-project/catapult.
[20] Google, "Catapult." https://chromium.googlesource.com/catapult/+/refs/heads/main/tracing/docs/perfetto.md.
[21] Android, "Profile your app performance." https://developer.android.com/studio/profile.
[22] Android, "Android studio." https://developer.android.com/studio.
[23] Android, "Android gpu inspector." https://gpuinspector.dev/.
[24] Android, "Generate trace logs by instrumenting your app." https://developer.android.com/studio/profile/generate-trace-logs.
[25] Android, "Debug class." https://developer.android.com/reference/android/os/Debug.
[26] Android, "Inspect trace logs with traceview." https://developer.android.com/studio/profile/traceview.
[27] Android, "Tracing 101." https://perfetto.dev/docs/tracing-101.
[28] "Angry birds wiki : Unity." https://angrybirds.fandom.com/wiki/Unity.
[29] "Chapter 10 page frame reclamation." https://www.kernel.org/doc/gorman/html/understand/understand013.html.
[30] S. Chan, "The iphone's top apps are nearly 4x larger than five years ago." https://sensortower.com/blog/ios-app-size-growth-2021, 2021.

Annexe B

Explications de mes scripts

B.1 Scripts du scénario 1

Les scripts `script_startup_cold.sh` et `script_startup_hot.sh` utilisent le champ *TotalTime* de la commande *adb shell am start-activity -W -S packageName/.activityName* pour retourner respectivement le temps de lancement d'une application en cold start et en hot start. L'option *-S* est retirée pour le hot start car celle-ci tue l'application à chaque lancement. À la place, un appui sur la touche Home du mobile (*adb shell input keyevent KEYCODE_HOME*) est réalisé pour garder l'application en mémoire à chaque lancement en hot start.

B.2 Script du scénario 2

Le script `script_cpu_all.sh` réalise les opérations suivantes :

- il lance une application avec la commande :
  adb shell am start-activity -W -S packageName/.activityName
- il laisse le temps à l'application de se lancer
- il attend 2 secondes pour simuler une utilisation de l'application
- il appuie sur la touche Home du mobile pour mettre l'application en arrière-plan et lance l'application suivante. Cela sera réalisé autant de fois qu'on veut d'applications en arrière-plan (0, 2, 4, 6 ou 8).
- une fois les applications en arrière-plan, il additionne le pourcentage d'utilisation du CPU de chaque processus obtenu avec la commande *adb shell top* toutes les secondes pendant 100 secondes. L'utilisation du CPU par le shell produit par ADB n'est pas comptée car il ne fait pas réellement partie du mobile.
- il calcule la moyenne et la valeur maximale d'utilisation du CPU trouvées. Ces deux dernières valeurs sont divisées par 8 car le CPU de mon mobile possède 8 cœurs, donc les valeurs d'utilisation du CPU retournées par *adb shell top* peuvent dépasser les 100 % jusqu'à atteindre les 800 %.

B.3 Scripts du scénario 3

Le script `script_cpu_one_ps.sh` utilise le champ *%CPU* de la commande *top* qui est divisé par 8 comme le CPU de mon mobile possède 8 cœurs pour retourner le pourcentage de l'utilisation du processeur.

Le script `script_memory_RSS.sh` utilise le champ *VmRSS* du fichier */proc/[pid YouTube]/status* pour retourner l'utilisation de la mémoire.

Le script `script_network.sh` calcule le nombre d'octets reçus par l'interface wlan0 en utilisant le champ *wlan0* du fichier */proc/net/dev*.

B.4 Commande du scénario 4

Pour réaliser les mesures du scénario 4, la commande suivante est 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
- 495 MB-822 MB-0.94 GB : PSS de l'application sous la forme *minPSS-avgPSS-maxPSS*
- 331 MB-444 MB-623 MB : USS de l'application sous la forme *minUSS-avgUSS-maxUSS*
- 450 MB-586 MB-816 MB : RSS de l'application sous la forme *minRSS-avgRSS-maxRSS*
- over 10 : nombre d'échantillons prélevés sur la période (ici 24 heures)

La métrique *Resident Set Size* (RSS) indique le nombre de pages partagées et non partagées utilisées par une application, donc toute la mémoire utilisée.

La métrique *Unique Set Size* (USS) indique le nombre de pages non partagées utilisées par une application.

Pour obtenir le PSS maximum d'une application, il faut juste récupérer le *maxPSS* retourné par la commande ci-dessus.