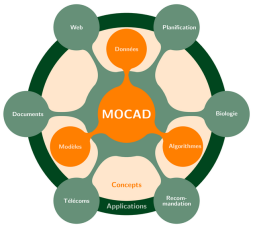

Rapport de Projet de fin d'étude

Réduction de la consommation énergétique des applications mobiles sur Android

Master MoCAD, Université de Lille

Auteur : Antonin Carette
Encadrant : Giuseppe Lipari

Version du 1er mars 2016

Table des matières

Introduction

Le nombre de recherches liées à la réduction de la consommation énergétique a considérablement augmenté depuis l'arrivée des *objets connectés* (smartphones, montres connectées, etc...) sur le marché de l'électronique pour un large public. Ces appareils embarqués, même s'ils utilisent des microprocesseurs à basse consommation ou des algorithmes performants leur permettant de gérer au mieux l'ordonnancement des tâches, dépensent encore trop d'énergie pour certaines tâches récurrentes – par exemple, les échanges réseaux lors d'une géolocalisation.

Nous nous intéressons à la consommation énergétique des applications de type "multimédia" pour le système d'exploitation Android [1], ainsi qu'aux différentes méthodes permettant de diminuer l'énergie dépensée, pour son fonctionnement, du point de vue du système d'exploitation et du système embarqué.

Aujourd'hui, de nombreux constructeurs et designers de processeurs se partagent le marché pour les smartphones [2]. Nous nous focalisons dans cette étude sur les processeurs *big.LITTLE*, conçus par l'entreprise Britannique ARM [3]. Le *big.LITTLE* est utilisé dans beaucoup de smartphones et tablettes, et utilise des techniques intéressantes de réduction de la consommation énergétique.

Dans le chapitre un, nous présenterons le modèle énergétique pris en compte dans cette étude, le processeur *big.LITTLE*, ainsi que les solutions du processeur pour économiser de l'énergie.

Dans le chapitre deux, nous nous pencherons sur le cas des différentes techniques et algorithmes d'ordonnancement pour la gestion des applications dans les systèmes d'exploitation temps-réel [4] (ou RTOS). Un algorithme d'ordonnancement pour des tâches informatique est un algorithme permettant de classifier les tâches selon un ou plusieurs critères. Contrairement au système d'exploitation grand public (*General Purpose*), les applications des systèmes d'exploitation temps-réel verront leur temps de réponse plus réduit. Étant donné la nature périodique des applications de

1. https://www.android.com/intl/fr_fr/
2. http://www.greenbot.com/article/2095485/android-processors-the-past-present-and-future-of-sm
html
3. http://www.arm.com
4. https://en.wikipedia.org/wiki/Real-time_operating_system

type "multimédia", l'étude des méthodes et algorithmes régissant un système d'exploitation temps-réel pourra être l'occasion de comparer la gestion avec un système d'exploitation grand-public comme Android, notamment dans la gestion du côté aléatoire de l'exécution des *threads* d'une application donnée.

Dans le chapitre trois, nous ferons l'état des travaux effectués en *Machine Learning*. Étant donné l'association très forte du logiciel et du matériel dans notre problème, une application pourra se comporter énergétiquement de 2 manières opposées selon le type de matériel supporté. Ainsi, il serait intéressant de pouvoir, non pas établir mathématiquement le modèle énergétique d'une application, mais pouvoir apprendre les paramètres régissant ce modèle mathématique (pour une application et une architecture matérielle donnée).

Enfin, avant de conclure, nous discuterons et nous détaillerons du choix d'une version du système d'exploitation Android, ainsi que des applications à utiliser quant à la validation d'hypothèses pour le problème donné. Nous en profiterons pour présenter également les différentes méthodes et outils permettant de mesurer de façon efficace les performances d'une application, sa consommation énergétique et la consommation du processeur (ou consommation CPU) de l'appareil mobile.

Chapitre 1

Modèle énergétique

Dans cette étude, nous prendrons pour exemple le modèle développé par Chiyoung Seo, Sam Malek et Nenad Medvidovic[1], de l'Université de Californie. L'avantage de ce modèle est qu'il considère les coûts de l'infrastructure pour un logiciel donnée.

1.1 Définition du modèle énergétique

Ce modèle énergétique définie le coût énergétique d'un logiciel ($E_{software}$) comme l'énergie nécessaire pour le calcul et l'accès aux données ($E_{comp-access}$), l'énergie nécessaire pour envoyer des données sur le réseau (E_{com}) et le coût de l'infrastructure (E_{infra}). Le coût de l'infrastructure pourra être considéré comme le coût d'utilisation de la machine virtuelle Java (ou JVM) pour une application Java. .

$$E_{\mathsf{software}} = E_{\mathsf{comp-access}} + E_{\mathsf{com}} + E_{\mathsf{infra}} \tag{1.1}$$

Ce modèle a été repris en 2015[2] et simplifié, afin d'inclure le coût énergétique de l'infrastructure logicielle dans l'énergie nécessaire pour le calcul et l'accès aux données :

$$E_{\mathsf{software}} = E_{\mathsf{comp-access-infra}} + E_{\mathsf{com}} \tag{1.2}$$

Nous nous intéressons spécifiquement au coût de calcul et d'accès aux données, afin de pouvoir le minimiser et ainsi réduire la consommation d'énergie globale d'une application.

Pour l'accès aux données, il pourra s'agir de minimiser les défauts de cache. De même, nous définissons la consommation énergétique des processeurs (E_{CPU}) mo-

dernes comme le produit de la capacitance (c), de la fréquence du processeur (f) et du voltage (V) à la puissance 2[2].

$$E_{CPU} = c * f * V^2 \tag{1.3}$$

Enfin, nous pouvons définir l'énergie consommée par une tâche ou un programme comme le produit de la puissance (P_{task}) requise pour faire fonctionner la tâche et du temps d'exécution de la tâche (T_{task}) [3] :

$$E_{\mathsf{task}} = P_{\mathsf{task}} \cdot T_{\mathsf{task}} \tag{1.4}$$

Or, la puissance d'une tâche dépend de la fréquence, selon un polynôme du 3ème degré. De plus, chaque coefficient du polynôme sera différent pour chaque tâche. Ainsi, afin de pouvoir calculer la consommation énergétique d'une tâche donnée, il faut pouvoir la mesurer[3].

Nous allons maintenant discuter du processeur *big.LITTLE*, ainsi que des techniques générales disponibles pour réduire la consommation énergétique.

1.2 Le *big.LITTLE*

Le processeur *big.LITTLE* (voir figure 1.1) consiste à gérer le ratio Puissance/Économie d'énergie pour un système et l'ensemble des applications installées sur ce dernier, via l'utilisation simultanée de processeurs à très faible (les *LITTLE*) et à grande consommation (les *big*). Les processeurs *LITTLE* détiennent une fréquence comprise entre 200MHz et 1.4GHz, tandis que la fréquence des processeurs *big* est comprise entre 200MHz et 2GHz, afin d'adapter la consommation énergétique aux besoins des applications. Ainsi, les applications peu gourmandes en ressources iront utiliser les processeurs *LITTLE* pour leurs calculs, tandis que les calculs des applications nécessitant plus de ressources préféreront les processeurs *big*[4].

La figure 1.2 illustre bien le rendu Énergie/Performance pour un coeur *big* (Cortex-A15) et un coeur *LITTLE* (Cortex-A7), issus d'un même processeur *big.LITTLE*. La figure 1.3 illustre quant à elle la consommation d'énergie d'une multiplication de matrices de taille 150 sur les coeurs *big* et *LITTLE*, pour un même processeur et à des fréquences différentes. Sur cette figure, nous pouvons déjà remarquer qu'à une même fréquence, un coeur *LITTLE* consomme moins qu'un coeur *big*.

Ces processeurs, ou coeurs, ont pour avantage d'être activé et désactivé en *run-time*, lorsqu'il en est nécessaire. Une application utilisant peu de ressources à un instant

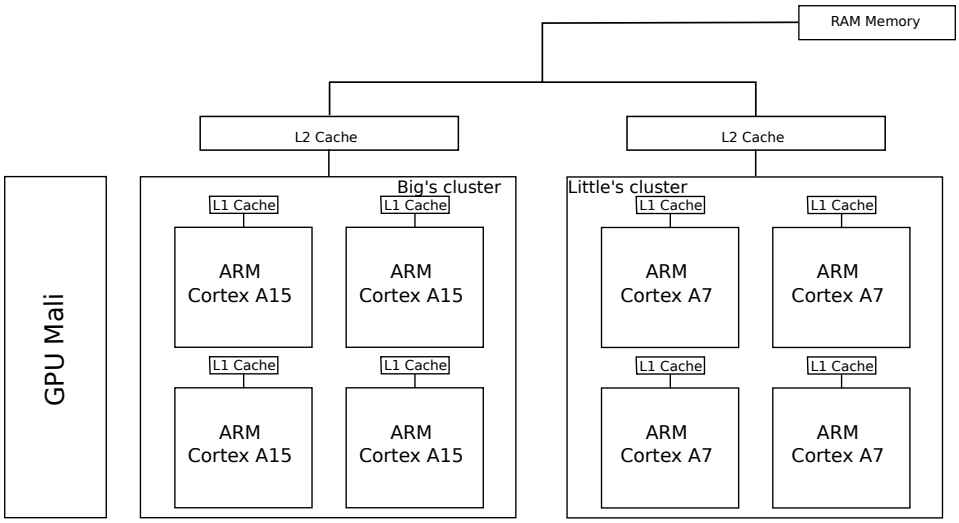

FIGURE 1.1 – Un modèle octocore *big.LITTLE*, composé de 4 coeurs *big* (Cortex-A15) et 4 coeurs *LITTLE* (Cortex-A7).

t pourra donc "demander" de n'utiliser que les processeurs à très faible consommation, jusqu'au moment où un plus grand nombre de ressources est demandé. Malheureusement, l'activation et la désactivation de ces processeurs coûte en ressource, et pourra aussi agir sur le temps de réponse de l'application, dû au temps d'activation d'un processeur. Aujourd'hui, on retrouve sur un processeur Exynos du constructeur SAMSUNG la technologie *big.LITTLE*, équipée de 4 processeurs *big* et 4 processeurs *LITTLE*[1].

1.3 Les solutions actuelles de réduction énergétique

Un processeur pour un matériel embarqué, comme le *big.LITTLE*, a plusieurs moyens de réduire, par lui-même, sa consommation d'énergie[5] de façon efficace et sans réduire drastiquement les performances.

Une première solution logique consiste à diminuer la fréquence des processeurs *big*, ce qui influera directement sur le temps de réponse mais aussi sur la qualité de service. De ce fait, une baisse trop importante de la fréquence des processeurs *big* pourra influer négativement sur l'expérience utilisateur (ou "qualité de service"), tandis qu'une baisse très faible de la fréquence influera peu sur l'économie d'énergie. Il est ainsi nécessaire de mesurer dynamiquement la fréquence nécessaire au fonctionnement d'une application, sans détériorer l'expérience utilisateur.

Une technique efficace permettant d'ajuster dynamiquement la fréquence du processeur est l'algorithme DVFS (*Dynamic Voltage & Frequency Scaling*), qui permet

1. `http://www.samsung.com/semiconductor/minisite/Exynos/w/solution.html#?v=modap_8octa`

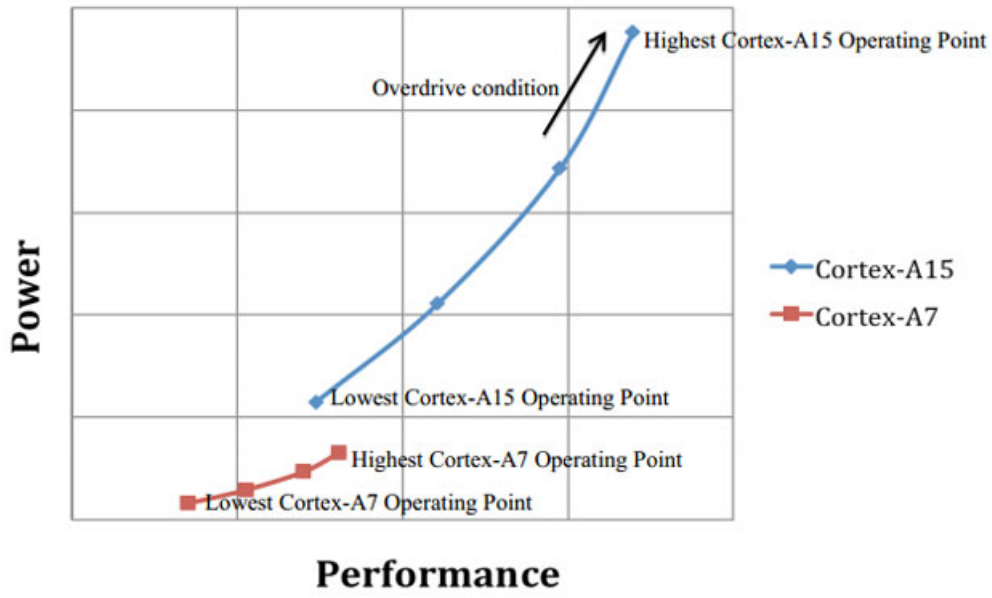

FIGURE 1.2 – Exemple d'un rendu Énergie/Performance pour un Cortex-A15 (*big*) et un Cortex-A7 (*LITTLE*), issus d'ARM.

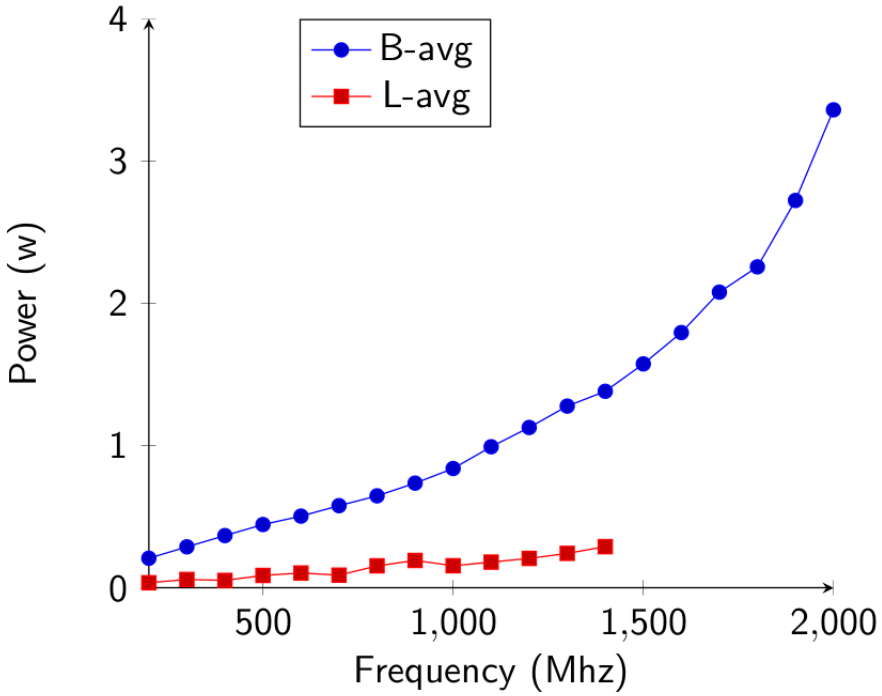

FIGURE 1.3 – Tracé moyen de la consommation d'énergie pour les coeurs *big* (B-avg) et *LITTLE* (L-avg), en fonction de la fréquence, sur un programme de multiplication de matrices de taille 150.

de gérer l'énergie dynamiquement en augmentant ou diminuant le voltage, selon la consommation du ou des processeur(s) au temps courant. Malheureusement, le problème de DVFS, pour les processeurs hétérogènes, est l'attribution des tâches biaisée pour ces derniers [6][7].

Une deuxième solution consiste à "éteindre" les processeurs non-utilisés ou non-voulus. Ainsi, le système fera tourner l'ensemble des programmes dynamiquement sur une plage limitée de processeurs par la demande de la consommation d'énergie à un temps t. Une technique efficace qui permet de faire ceci est le DPM (*Dynamic Power Management*). Le problème de cette technique, utilisée par de nombreux algorithmes, est que la désactivation et l'activation d'un processeur a un coût en temps et en énergie non-négligeable. De même, l'activation et la désactivation d'un processeur se fera dynamiquement si l'application requière ou non un processeur *big* à un temps t. Dans le pire des cas, une application consommant à la limite d'un processeur $LITTLE$ se verra attribué un processeur *big* et un processeur $LITTLE$ à chaque pas de temps, désactivant ou réactivant à la volée et à chaque pas de temps un processeur *big* pour utilisation.

Chapitre 2

Le système d'exploitation Android

2.1 Généralités

Android est un système d'exploitation Open-Source [1], pour objets connectés comme les téléphones portable, les montres connectées, le secteur de l'automobile ou encore celui de la télévision. Basé sur le noyau Linux [2], il est développé par Google [3] depuis 2005. Son architecture est composé de 4 couches principales (voir figure 2.1). Ces couches, de la plus basse à la plus haute, sont :

- le noyau Linux (*Linux kernel*),
- les bibliothèques C/C++ et la machine virtuelle Android (*Native libraries*),
- le kit de développement d'applications (*Android Framework*),
- les applications et widgets installés par défaut (pour la base du système d'exploitation) (*Android Applications*).

Actuellement, la version du noyau Linux prise en compte pour la version 6.0 d'Android (nom de code "Marshmallow" [4]) est la version 3.4. Le noyau Linux utilisé pour Android a été modifié par Google afin de faciliter son intégration sur les appareils mobiles. Ainsi, le système d'exploitation n'embarque pas la *glibc*, les outils standards et un système d'affichage natif tel que le *X Window System*, intégrés par défaut dans un noyau Linux. C'est pour cela qu'Android n'est pas considéré comme un système d'exploitation GNU/Linux.

Les applications Android sont exclusivement écrites en Java, utilisant l'API fournie par Google [5]. Chacune de ces applications est lancée sur une instance de la machine virtuelle Android.

1. `https://source.android.com/source/downloading.html`
2. `https://en.wikipedia.org/wiki/Linux_kernel`
3. `https://www.google.com/intl/fr/about/`
4. `https://www.android.com/intl/fr_fr/versions/marshmallow-6-0/`
5. `http://developer.android.com/reference/packages.html`

FIGURE 2.1 – Schéma de l'architecture d'Android

Une machine virtuelle est un logiciel qui se comporte comme une machine physique, permettant d'exécuter des applications écrites dans un langage spécifique. Une machine virtuelle Java exécutera spécifiquement du *bytecode* Java. Android utilise sa propre machine virtuelle (Dalvik, ou ART). L'utilisation d'une machine virtuelle propre pour Android est justifiée pour optimiser le lancement de plusieurs instances de cette dernière, étant donné que chaque application sera isolée des autres par son lancement dans une machine virtuelle. L'utilisation de Dalvik permet à Android de réduire considérablement son "empreinte mémoire" [6] (*memory footprint* dans la littérature scientifique) [7]. Jusqu'à la version 5.0 d'Android (nom de code "Lollipop" [8]), Dalvik était utilisée par défaut par ce système d'exploitation. Depuis la version 5.0, ART a remplacé Dalvik, afin de pouvoir utiliser la compilation anticipée pour une application donnée (*Ahead-of-Time* dans la littérature scientifique), au lieu de la compilation *Just-in-Time* qui était offerte par Dalvik [9].

Android permet l'utilisation des JNI [10] (*Java Native Interface*), afin de pouvoir utiliser du code natif (C ou C++) dans une classe Java. Cette méthode, bien que permettant d'utiliser certaines fonctionnalités bas-niveau (comme l'utilisation des fonctions système `clock_gettime()` ou `nanosleep()`) a le défaut de ne pas permettre la portabilité logicielle sur des systèmes d'exploitation différentes.

6. Une "empreinte mémoire" est la quantité de mémoire utilisée par le programme en cours d'exécution.
7. `http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf`
8. `https://www.android.com/intl/fr_fr/versions/lollipop-5-0/`
9. `https://source.android.com/devices/tech/dalvik/`
10. `https://en.wikipedia.org/wiki/Java_Native_Interface`

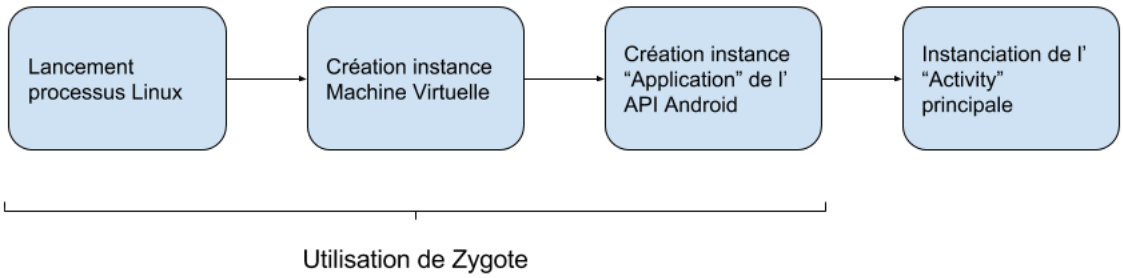

FIGURE 2.2 – Schéma du lancement d'une application

2.2 Gestion des applications

Un processus est lancé pour chaque application, et chaque processus contient sa propre instance, ou exécution, de la machine virtuelle Android : la Machine virtuelle Dalvik (DVM) ou l'Android RunTime (ART).

La figure 2.2 présente les étapes quant au lancement d'une application. Les trois premières étapes du schéma ne sont pas instantanées en temps, car elles nécessitent de charger un certain nombre de librairies afin de pouvoir instancier une application. Ce décalage en temps dégrade l'expérience utilisateur. Pour y remédier, la technologie *Zygote*[11] a été lancé, permettant de précharger des librairies utiles pour chaque application Android, et ainsi de les partager un nombre illimité de fois. Cela évite donc de charger en mémoire ces librairies pour chaque application.

2.2.1 Mapping des threads

Dans un processus, l'on retrouve un thread principal (le thread "main" ou "UI"), responsable de mettre-à-jour l'interface utilisateur – d'autres threads peuvent être créer par l'utilisateur, au besoin. Il est recommandé d'exécuter toute longue tâche dans un thread, afin de ne pas entraîner de problèmes de délai pour la réponse de l'application. Il est important de savoir que toute décision prise pour l'application (la synchronisation des données par exemple), se fait dans le thread UI. C'est le thread UI qui est responsable de lancer les fonctionnalités d'Android comme les "Activity" ou les "Service".

2.2.2 Algorithmes d'ordonnancement

Les tâches sont ordonnancées par leur priorité, priorité donnée par le *Process Ranking System*[12]. Ce système permet notamment de déterminer quel processus en cours

11. `http://developer.android.com/training/articles/memory.html`
12. `http://developer.android.com/guide/components/processes-and-threads.html`

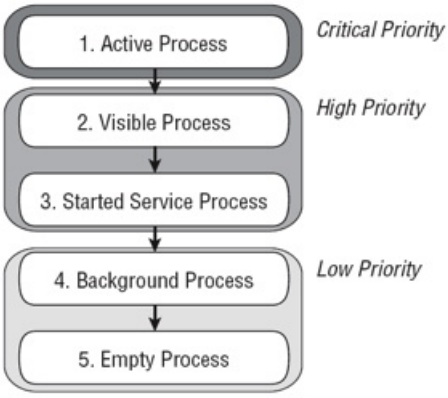

FIGURE 2.3 – Schéma de la priorité accordée aux processus pour Android

d'exécution de le "tuer" en cas de manque de ressources (la RAM par exemple), et est dépendant de la visibilité de l'application et des composants en cours d'exécution (par exemple, les "Services" associés à l'application).

La détermination de la priorité sur les tâches est donnée par la figure 2.3. Une application peut être classée comme en "Premier plan" (*Active process*) ou en "Arrière plan"(*Background process*). Les applications en *Active process* pourront consommer jusqu'à 95% du CPU, et les application en *Background process* ne pourront consommer, quant à eux, que jusque 5% du CPU. À noter qu'il est possible de changer la priorité, et de lui attribuer une priorité statique (entre 1, minimale, et 10, maximale) via la méthode Java standard *Thread.setPriority()*.

L'ordonnanceur Android utilise par défaut l'algorithme *Completely Fair Scheduler*, présent depuis la version 2.6.23 du noyau Linux (`SCHED_OTHER`). Cet algorithme trie les processus en fonction du temps d'exécution demandé. Chaque thread d'un processus détiendra une valeur représentant l'équité par thread. Cette valeur sera comprise entre -19 (plus de temps CPU sera alloué pour ce thread) et +20 (moins de temps sera alloué pour ce thread). Les processus ayant la même valeur seront ordonnancé en mode "*Round robin*". Il est possible de choisir un autre algorithme que CFS pour la gestion des processus, notamment un ordonnancement prévu pour les systèmes temps-réel comme *FIFO* (`SCHED_FIFO`) ou *RR* (`SCHED_RR`) depuis la version 4.1 d'Android [13].

L'algorithme CFS manipule l'allocation des ressources CPU pour l'exécution des processus. Il vise à maximiser l'utilisation du CPU, pour maximiser les performances interactives. Par exemple, pour deux tâches en cours, l'algorithme CFS va mimer le comportement simple visant à partager équitablement la consommation CPU à chaque tâche. La structure de données utilisée pour cet algorithme est un arbre de recherche permettant d'équilibrer ses charges par lui-même, comme l'arbre bicolore,

13. `https://source.android.com/devices/audio/latency_contrib.html`

ou arbre rouge et noir (*red-black tree* dans la littérature scientifique) [14]. Il y a ainsi une file d'attente par processeur, et un noeud (processus) peut être placé dans la file d'attente d'un des processeurs si ce dernier n'est pas rempli.

L'algorithme CFS fonctionne de la façon suivante, lors de son invocation pour le lancement d'un nouveau processus :

1. le processus ayant eu le moins de temps d'exécution jusque là (le noeud le plus à gauche de l'arbre bicolore) est envoyé pour être exécuté par le processeur,
2. si le processus complète son exécution, il est supprimé du système et de l'arbre bicolore,
3. sinon, il est réinséré dans l'arbre bicolore, et un nouveau temps d'exécution lui est donné.

La complexité de la décision de l'ordonnanceur est en $0(1)$, et la réinsertion ou la délétion d'un élément de l'arbre est en $0(\log n)$, où n est le nombre d'éléments présents dans l'arbre. Cet algorithme ne nécessite pas d'heuristique pour déterminer l'interactivité d'un processus, étant donné que l'ordonnanceur utilise une granularité temporelle à la nanoseconde [15].

2.3 Gestion de la mémoire

Android laisse le soin au noyau Linux de gérer les processus et les threads. Le système d'exploitation n'offre pas d'espace *swap* pour la mémoire, mais utilise le stockage secondaire [16] et le *memory mapping* [17], une technologie permettant de donner l'illusion à un programme qu'il peut utiliser la mémoire contiguë, alors qu'elle est en fait fragmentée.

2.3.1 Machine virtuelle Android

Dalvik

Dalvik est une implémentation de machine virtuelle basée sur les registres (et non pas sur une pile comme la JVM), utilisant trois opérandes dans ses instructions.

Ainsi, un programme aussi simple que le calcul du carré de la somme de deux entiers (figure 2.4) se convertit de façon beaucoup plus réduite (figure 2.5) que la

14. `https://en.wikipedia.org/wiki/Red%E2%80%93black_tree`
15. `http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fkerneltrap.org%2Fnode%2F8059`
16. `https://en.wikipedia.org/wiki/Paging`
17. `https://en.wikipedia.org/wiki/Virtual_memory`

```
public int m(int i1, int i2) {
int c = a + b;
    return c * c;
}
```

FIGURE 2.4 – Exemple d'un programme Java permettant de calculer le carré de la somme de 2 entiers

```
.method public m(II)I
add-int v0, v2, v3
    mul-int/lit-8 v0, v0, v0
    return v0
.end method
```

FIGURE 2.5 – *Bytecode* Android (Dalvik), du programme en figure 2.4

transformation effectuée par la machine virtuelle Java (figure 2.6). On remarque que le nombre d'instructions de la DVM (pour Dalvik Virtual Machine) est plus réduite que celles de la JVM ; par contre, elles sont beaucoup plus larges (ce qui est le défaut des instructions pour les machines à registre, comparé aux machines à pile).

Avec Dalvik, Android laisse le soin de compiler une application par un compilateur intermédiaire, qui renverra du *bytecode*[18] Dex (Dalvik Executable) en sortie, lors de l'installation de l'application par l'utilisateur. Ainsi, à chaque lancement de l'application, le *bytecode* de l'application sera compilé vers un langage machine spécifique au processeur physique, afin de pouvoir être lancé sur le matériel (la compilation *Just-in-Time*[19]). Cette méthode perd ainsi en performance, à chaque lancement d'une application.

18. https://en.wikipedia.org/wiki/Bytecode
19. https://en.wikipedia.org/wiki/Just-in-time_compilation

```
.method public m(II)I
iload_1
    iload_2
    iadd
    istore_3
    iload_3
    iload_3
    imul
    ireturn
.end method
```

FIGURE 2.6 – *Bytecode* Java, du programme en figure 2.4

ART

ART (pour Android RunTime) a été conçu dans le but de pallier au défaut principal mis en avant par Dalvik, en compilant une fois pour toute l'application. ART compilera une seule fois l'application en la traduisant directement en langage machine, et stockera cette version de l'application compilée. Cette méthode s'appelle la compilation *Ahead-of-Time*[20]. Cette technologie permettra ainsi de ne plus compiler l'application à chaque demande de lancement de cette dernière. Elle fait alors profiter la réponse du lancement du logiciel et de l'économie d'énergie, au détriment de l'espace de stockage. De plus, les applications compatibles avec Dalvik sont compatibles avec ART, vu qu'elles se basent tous deux sur le *bytecode* Dex[21].

2.3.2 *Garbage collector*

Android dispose d'un *garbage collector*[22], simple, à une génération. Cette technologie, présente dans la machine virtuelle Android, permet au développeur de ne pas se soucier de la gestion de la mémoire de son application. Le désavantage est que, en plus de la perte de contrôle sur la gestion mémoire par le développeur, le *garbage collector* influera négativement sur le temps d'exécution du processus. Pour la DVM, cette lenteur peut ainsi interférer négativement sur le temps de réponse des applications utilisées - un ralentissement sur l'application ou une "pause" pouvant aller jusqu'à $50ms$.

Une bonne méthode de développement, outre de ne pas instancier des objets inutiles au bon établissement de l'application, consiste à ne pas instancier les objets durant les "phases critiques" de l'application (par exemple, pendant les phases requérant une interaction utilisateur). Ainsi, pour un jeu-vidéo, il s'agira de créer les objets nécessaires au bon déroulement du niveau **avant** que le niveau ne soit lancé. Le *garbage collector* pourra ainsi détruire les objets instanciés à la fin du niveau, afin d'initialiser ceux intervenant dans le prochain.

Même si le changement de machine virtuelle dans la version d'Android Lollipop permet de diminuer la pause prise dans l'exécution du processus lié au *garbage collector*, le temps de pause des applications est diminué (dans la dizaine de millisecondes aujourd'hui). Cependant, ce temps n'atteint pas encore les objectifs entrepris par Google[23], qui est de réduire la pause induite par l'exécution du *garbage collector* à $3ms$.

20. https://en.wikipedia.org/wiki/Ahead-of-time_compilation
21. https://source.android.com/devices/tech/dalvik/
22. https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
23. http://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/2

Chapitre 3

Gestion de l'énergie

3.1 Généralités

Les applications multimédia, applications pour lesquelles nous nous intéressons particulièrement dans ce projet, consomment beaucoup d'énergie ; notamment dû au fait qu'elles communiquent avec le réseau, l'utilisateur, et qu'elles font tourner de nombreuses tâches très gourmandes en énergie.

Une définition d'une "application multimédia" peut être donnée par une application usant d'une collection de sources média tel que du texte, des graphiques et/ou images, du son, de l'animation et/ou de la vidéo [1].

3.2 Système Android

3.2.1 Aide de l'utilisateur

En 2013, Pietro Mercati, Andrea Bartolini, Francesco Paterna, Tajana Simunic Rosing et Luca Benini ont proposé de classifier les applications installées par défaut[8] et par l'utilisateur en 2 catégories :

— critique,

— non-critique.

Les applications critiques sont toujours exécutées avec la fréquence CPU la plus grande, tandis que les applications non-critiques peuvent être utilisées sur les processeurs à faible consommation, et à fréquence réduite. Dans une mise-à-jour des recherches en 2014, ils proposent de gérer la classification de ces applications par l'utilisateur, via une interface[9]. Un des points négatifs que l'on peut souligner dans

1. `http://www.cs.cf.ac.uk/Dave/ISE_Multimedia/node10.html`

cette méthode est qu'un utilisateur pourra se tromper dans la gestion des applications installées, et ainsi allouer accidentellement un coeur *big* pour une application peu consommatrice en énergie en la classant comme application phare.

3.2.2 Sensitivité de l'application

D'autre travaux menés en 2014 par Pi-Cheng Hsiu, Po-Hsien Tseng, Wei-Ming Chen, Chin-Chiang Pan et Tei-Wei Kuo ont été effectués sur la recherche d'un meilleur ordonnancement des threads pour le système d'exploitation Android[10]. Cet ordonnancement travaillera au travers d'une approche utilisant un concept nommé "sensitivité". La sensitivité, pour une application donnée, définie l'attention que porte l'utilisateur à cette dernière. C'est ce qui permettra, pour ces travaux, de ne pas détériorer la qualité de service. La sensitivité est déterminée sur le statut courant de l'application (en premier plan, en arrière plan, etc...), et non sur sa catégorie. De même, elle pourra variée au cours de l'exécution de l'application.

Trois niveaux de sensitivité peuvent être alloués :

- *high*, l'application est en premier plan et intéragit actuellement avec l'utilisateur,
- *medium*, l'application est en premier plan mais n'intéragit pas actuellement avec l'utilisateur,
- *low*, l'application est en arrière-plan.

À travers ce concept et ces différents degrés de sensitivité, l'ordonnanceur exploitera la sensitivité des applications à un temps t donné afin de pouvoir accorder des priorités aux threads contenus dans ces applications, l'allocation et la migration des threads. [2] Le gouverneur CPU (*kernel governor*) [3] utilisera aussi la sensitivité des threads, afin de diriger les ressources de calcul avec l'aide des algorithmes DVFS et DPM.

3.3 Systèmes temps-réel

Nous pouvons voir une application multimédia comme une application contenant des threads périodiques, avec une période stochastique.

2. L'allocation des threads va pouvoir assigner un nouveau thread au coeur qui contient le moins de threads, afin de pouvoir équilibré la charge des threads sur un processeur (par la migration des threads, via CFS). Enfin, la priorisation des threads se fera en assignant une priorité spécifique à chaque thread, et allouera le temps CPU (par périodes) aux threads lancés en fonction de leur sensitivité.

3. Un gouverneur CPU est un module du noyau qui a pour rôle de gérer la fréquence du processeur en fonction de la demande en ressources du système et des applications.

Les systèmes temps-réel [4] prennent en compte des contraintes temporelles[11], contrairement aux systèmes grand-public (*general-purpose*), au détriment d'un code plus complexe et d'une plus grande complexité à interagir physiquement avec ses applications. Ces systèmes temps-réels utilisent donc une borne temporelle spécifique à chaque tâche, que l'on appellera *deadline*. L'objectif d'un ordonnanceur pour systèmes temps-réels est d'arriver à terminer l'exécution de la tâche en cours avant sa *deadline*. Ces systèmes peuvent alors être utilisés pour des tâches critiques (le temps réel strict / *hard real-time*), ou alors pour des tâches qui autorisent certains dépassement de borne temporel (le temps réel souple / *soft real-time*).

Une tâche i est caractérisée, pour un système temps-réel, par un temps de calcul (C_i), une échéance (D_i) et une période (T_i). L'échéance caractérise le moment (un *timestamp* limite) à laquelle la tâche doit avoir terminé son exécution, tandis qu'une période représente une durée séparant les instances d'activation de la tâche.

Il sera intéressant d'étudier les algorithmes utilisés dans le cas des systèmes temps-réel, afin de pouvoir extraire de bonnes méthodes *batchs* ou *stochastiques* concernant le calcul de la période d'une application donnée[12].

Les ordonnanceurs comme *FIFO* (`SCHED_FIFO`) ou *RR* (`SCHED_RR`) sont utilisés dans ces systèmes, et utilisent n tâches - contenues dans des listes de tâches.

3.3.1 Réduction de la consommation statique

En 2014, Vincent Legout, Mathieu Jan et Lauren Pautet on travaillé sur la réduction du nombre de périodes d'activité pour un processeur grande-consommation[13]. Ces travaux ont pour but de laisser le système utiliser le plus possible les processeurs basse consommation. Pour cela, un algorithme a été extrait : le LPDPM (pour *Linear Programming DPM*). Cet algorithme va créer de larges périodes d'inactivité sur les processeurs haute consommation et diminuer le nombre de réveils nécessaires pour retourner d'un état basse-consommation à l'état dit "nominal". Ce procédé réduit alors l'utilisation d'un processeur à haute consommation, ce qui permet d'économiser au final de l'énergie. Travaillant sur l'énergie statique (donc "OFF-LINE"), le système revient à résoudre un système d'équations linéaires, l'utilisation totale d'une tâche étant perçue comme son pire temps d'exécution (Worst Case Execution Time), en garantissant que le temps d'exécution de chaque tâche est égal à son propre pire temps d'exécution. Seulement validé sur les systèmes temps-réels, l'approche est en train d'être portée pour les systèmes hétérogènes.

4. `https://en.wikipedia.org/wiki/Real-time_computing`

3.3.2 Assignation des processeurs aux tâches temps-réel

Yan Wang, Kenli Li, Hao Chen, Ligang He et Keqin Li ont proposé une nouvelle méthode de planification de tâches pour les processeurs hétérogènes, comme le *big.LITTLE*[14], en 2014. Cette méthode, nommée *heterogeneous data allocation and task scheduling* (HDATS), permet de réduire la consommation totale d'énergie. Elle consiste à allouer les données dans les mémoires locales, puis utiliser un planificateur afin de réduire la consommation totale d'énergie. Les auteurs s'attaquent ainsi, au problème de réduire le plus possible la consommation d'énergie pour le processeur et la mémoire, sous une contrainte de temps. De même, la latence mémoire est réduite le plus possible.

3.3.3 Identification dynamique de la quantité CPU à fournir

Nous pouvons nous demander s'il est possible d'attribuer la quantité CPU la plus précise possible par application, afin de minimiser la consommation énergétique pour une application donnée, sans détériorer la qualité de service. De plus, il est plus courant de croiser sur le magasin d'application Android des applications propriétaires (*legacy applications*), dont on ne peut disposer du temps de démarrage et de fin de la tâche principale et donc ne pas associer correctement à chaque job sa *deadline*.

En 2009, des recherches menées par Tommaso Cucinotta, Luca Abeni, Luigi Palopoli et Fabio Checconi ont été réalisées sur l'identification de la quantité de CPU à fournir pour une application[15]. Ces travaux ont montrées qu'en interceptant des événements concernant le programme dans le système temps-réel, il est possible de suivre la période de l'application. Grâce à cette information, il est alors possible d'estimer un "budget énergétique" (quantité de CPU à fournir et fréquence du coeur à utiliser) à allouer pour le programme, et de faire une première estimation de la réservation CPU à faire. La mise-à-jour progressive des informations se fera par l'utilisation de l'algorithme LFS (Legacy Feedback Scheduling). Un problème avec cette méthode est qu'elle ne peut évaluer la période et le budget énergétique d'une application multi-threadée.

3.3.4 Création d'un OS Android avec gestion du temps-réel

De nombreuses recherches ont conduit à dire qu'Android n'est pas un système d'exploitation temps-réel proprement dit, mais qu'il pourrait être possible d'intégrer et de faire fonctionner des applications spécifiques à Android dans les systèmes temps-réel [16, 17].

Depuis 2013, un projet de Yin Tan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steven Y. Ko et Lukasz Ziark est de modifier le système d'exploiration Android pour en faire un système d'exploitation temps-réel [18].

D'autres recherches nous ont fait réfléchir sur le fait qu'il serait peut-être préférable d'utiliser des composants logiciels temps-réels pour Android. Des travaux de Yin Yan, Chunyun Chen, Karthik Dantu, Steven Y. Ko et Lukasz Ziarek ont été réalisé sur la machine virtuelle Dalvik [19], afin de pouvoir faire tourner toutes les applications dans une seule instance de la machine virtuelle. Cette implémentation permet de réduire le coût du changement de contexte, et active le contrôle détaillé du cycle de vie des applications.

Chapitre 4

Impact du *Machine Learning* dans la réduction de la consommation énergétique

4.1 Généralités

L'apprentissage automatique (ou *Machine Learning*) est un champ d'étude de l'intelligence artificielle [1], où l'objectif est de concevoir des programmes pouvant s'améliorer automatiquement avec l'expérience. Ce champ d'étude peut être découpé en 3 grandes parties :

- l'apprentissage supervisé : apprendre les relations entrée/sortie dans les données,
- l'apprentissage non-supervisé : apprendre à structurer dans une base de données,
- l'apprentissage par renforcement : apprendre à se comporter (apprentissage "ON-LINE" en prenant des décisions [2] séquentielles).

Des recherches récentes en *Machine Learning*, pour le problème donné, ont menés à utiliser les processus de décision Markoviens ainsi que les réseaux de neurones.

1. `https://fr.wikipedia.org/wiki/Apprentissage_automatique`
2. Une décision peut être vue comme le choix d'une action, dans un état donné.

4.2 Le processus de décision Markovien

4.2.1 Généralités

Le processus de décision Markovien (MDP) est un modèle de décision très utilisé dans un environnement stochastique. Dans ce modèle, un agent pourra prendre des décisions dans un environnement (connu, partiellement connu ou inconnu), et recevoir des récompenses en fonction des actions qu'il a effectué à un état donné. Nous pouvons considérer que l'agent peut effectuer un nombre infini d'actions dans l'environnement. Ce processus est représenté par 4 objets :

- l'espace contenant tous les états où l'agent peut aller,
- l'espace d'actions, pour un état appartenant à l'espace d'états,
- la probabilité qu'une action a dans un état S au temps t mènera à un état S' au temps $t+1$ [3],
- la récompense immédiate associée à une action a [4].

Un exemple d'un MDP est donné figure 4.1. Dans cette figure, l'espace d'états est S_0, S_1, S_2, l'espace d'actions pour l'ensemble des états est a_0, a_1, et chaque probabilité de transition d'un état à un autre via une action est représentée sur chaque arête de l'exemple, de même pour les récompenses.

Pour utiliser le processus de décision Markovien, il est nécessaire de définir un but à partir d'une fonction de récompense basée seulement sur l'état courant (S). Une politique π est le choix d'une décision pour chaque état de l'environnement. On peut voir une politique comme une suite de décisions pour atteindre un but.

Les processus de décision Markovien sont utilisés pour les algorithmes d'apprentissage par renforcement.

4.2.2 Recherches

EN 2009, Tang Lung Cheung, Kari Okamoto, Frank Maker, Xin Lio et Venkatesh Akella ont proposé une implémentation de framework, basé sur les processus de décision Markovien, dont le but est de minimiser ou maximiser dynamiquement la qualité de l'exécution d'une application utilisant le Wifi et/ou la synchronisation des données[20]. Cette dynamique dépendra de paramètres fixés par l'utilisateur, comme le temps souhaité avant la prochaine recharge du matériel.
Un exemple concernant la synchronisation des courriers électroniques dans une application mail est donné dans l'article. Dans cet exemple, la décision de synchroniser les courriers électroniques dépendra de l'heure actuelle, du temps restant jusqu'à la

3. Cette probabilité peut être plus formellement représenté par $P(s'|s,a)$, où $a \in A(s)$.
4. Cela peut aussi être vu comme l'utilité d'être dans un état au moment t.

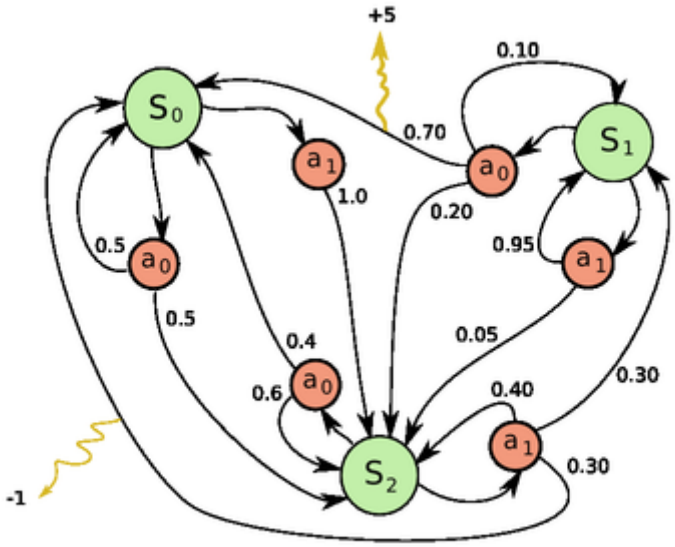

FIGURE 4.1 – Représentation d'un processus de décision Markovien, à 3 états et 2 actions. L'agent obtiendra une récompense de +5 s'il transite de l'état S_1 à S_0 en faisant l'action a_0, et une récompensé négative s'il transite de l'état S_2 à S_0 en effectuant l'action a_1.

prochaine charge, et du temps depuis la dernière synchronisation (ces 3 paramètres définissent l'état courant du MDP). Ainsi, la décision de synchroniser les e-mails à cet instant va produire une récompense immédiate (correspondant au coût en énergie), qui influencera sa prochaine action.
Les paramètres contrôlant l'environnement sont fixes. Ainsi, le nombre d'actions est limité, et le temps d'horizon est fini. De plus, le temps pour chaque prise de décision est fixé, et la décision sera prise au début de ce temps fixe, en tenant compte des informations de l'état courant.

D'autres études se basent sur le fait que l'environnement est partiellement observable voire inconnu, mais qu'il est possible cependant d'en bâtir un simulateur pour entraîner un algorithme d'apprentissage par renforcement type *Q-Learning*[5]. Un autre papier de recherche datant de 2009, écrit par Ying Tan, Wei Liu et Qinru Qiu, présente une méthode modifiée de *Q-Learning* permettant d'apprendre la politique optimale de gestion d'énergie, sans connaître d'information concernant la quantité CPU *a priori* à utiliser, et en se basant sur son historique d'actions[21]. Contrairement à d'autres papiers proposant un algorithme d'apprentissage "OFF-LINE" qui apprendra à choisir la meilleure politique parmi un jeu de politiques données[22][23], cette méthode "ON-LINE" apprendra une nouvelle politique de consommation. Les éléments connus dans l'environnement sont l'espace des états possibles ainsi que la vitesse de transition des fournisseurs de service.

5. `https://en.wikipedia.org/wiki/Q-learning`

L'algorithme *Q-Learning* a été modifié afin de pouvoir prendre en compte le délai causé par l'action (implicitement, la perte de performance associée à cette action) par l'ajout d'un coût Lagrangien [6] calculant le délai de l'action associé à son coût en performance. Cela a permi de montrer que cet algorithme réduit près 24% la consommation énergétique et 3% la latence, comparés aux résultats donnés par l'algorithme DPM.

Des recherches plus récentes, menées par Hao Shen, Ying Tan, Jun Lu, Qing Wu et Qinru Qiu, ont discutées de la modélisation du système et l'élaboration de la fonction de coût pour 2 types d'agents *Q-Learning* : les périphériques d'entrée/sortie et le microprocesseur[24]. En effet, les objectifs d'optimisation pour ces 2 types d'agents sont différents, tout comme leurs mesures de performance, étant donné que la consommation énergétique d'un périphérique entrée/sortie est donnée par une fonction décroissante monotone et non celle d'un microprocesseur. En effet, pour un microprocesseur, l'énergie sera perdue pour maintenir ce dernier en fonctionnement perpétuel[25].
Ces recherches ont permis de confirmer l'intérêt de l'apprentissage dynamique quant à son adaptation pour différents types d'architecture, et d'améliorer la vitesse de convergence de l'algorithme, démontré dans l'article précédent, pour un environnement non-Markovien.

Un travail de Yanzhi Wang, Qing Xie, Ahmed C. Ammari et Massoud Pedram, datant de 2011, présente une technique adaptative supervisée, héritée de l'algorithme DPM. Cette technique va apprendre des données d'entrée et du système afin de pouvoir ajuster la politique de gestion de l'énergie "ON-LINE"[26], la mieux possible. Ces recherches utilisent une version modifiée de l'apprentissage par différence temporelle pour les MDPs partiellement observable, ce qui a pour but d'accélérer la convergence et alléger la confiance sur les propriétés Markoviennes. Quant à la prédiction de la quantité CPU à allouer pour une application, elle est donnée par le classificateur naïf de Bayes. Cette technique montre que la réduction de la consommation d'énergie peut aller jusqu'à 16.7% comparé à une l'approche DPM. De même, la réduction de la latence peut monter jusqu'à 28.6% comparé à cette même approche.

6. On parle ici du Lagrangien d'optimisation. Voir `https://fr.wikipedia.org/wiki/Lagrangien_(optimisation)`

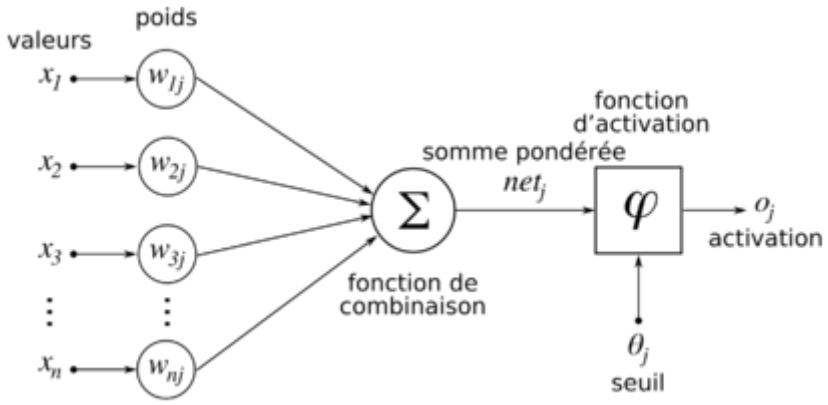

FIGURE 4.2 – Représentation schématique d'un neurone artificiel. Le neurone calcule la somme de ses entrées. Si cette valeur est supérieure à un seuil donné, le neurone est activé.

4.3 Le réseau de neurones artificiels

4.3.1 Généralités

Un réseau de neurones artificiels est un modèle de calcul inspiré par la biologie, notamment par le fonctionnement et la communication entre les neurones biologiques [7]. Dans notre cerveau, les neurones biologiques sont connectées entre eux, formant un réseau. Un neurone biologique [8] reçoit une information (un signal électrique), qu'il transférera à ses extrémités (dendrites) afin de pouvoir retransmettre ou inhiber l'information à un ou plusieurs neurone(s) voisin(s). Cette retransmission ou inhibition est faite par l'intermédiaire de substances chimique appelées "neuromédiateurs" [9].

Artificiellement, un neurone peut manipuler des valeurs binaires ou réelles. Différentes fonctions peuvent être utilisées pour le calcul de la sortie, qui peut être déterministe ou probabiliste. L'architecture du réseau de neurones artificiels peut être sans rétroaction [10], ou avec rétroaction (totale ou partielle). De même, la dynamique du réseau peut être synchrone (toutes les cellules calculent leurs sorties respectives simultanément) ou asynchrone [11]. Un exemple de neurone artificiel est donné figure 4.2.

7. https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels
8. https://fr.wikipedia.org/wiki/Neurone
9. https://fr.wikipedia.org/wiki/Neurotransmetteur
10. Une rétroaction est définie comme l'influence d'une sortie d'une cellule sur son entrée.
11. http://www.grappa.univ-lille3.fr/polys/apprentissage/sortie005.html

4.3.2 Recherches

Une équipe de recherche, en 2012, a essayée d'établir des modèles d'énergie pour 3 calculs à haute-performance (*High Performance Computing*, ou HPC), essentiellement pour le CPU et le DIMM [12]. Ce projet, mené par Ananta Tiwari, Michael A. Laurenzano, Laura Carrington et Allan Snavely, a permis d'étudier le comportement de ces deux matériels dans différentes situations logicielles et matérielles[27]. Les chercheurs ont utilisés les réseaux de neurones pour établir la prédiction du comportement énergétique en fonction du matériel et du logiciel. Le test a pu être effectué sur 3 grands algorithmes : la multiplication de matrices, la décomposition LU ainsi que la modification de grands tableaux. Le réseau neuronal a été entraîné sur des données empiriques d'énergie (le temps d'exécution du programme et l'énergie consommée pour ce temps d'exécution) collectées sur l'architecture cible. Le but de l'utilisation du réseau de neurones est de pouvoir identifier les paramètres du comportement énergétique. En moyenne, l'erreur est de 5.5% pour ces 3 algorithmes.

Les processeurs graphiques [13] [14] ont un certain nombre de différentes configurations possible (la fréquence des coeurs, le nombre des unités de calcul en parallèle ainsi que la bande passante en mémoire). Il est intéressant de pouvoir estimer quelle est la performance et l'énergie consommée pour une application, pour un processeur graphique sur le marché. En 2015, une équipe de recherche menée par Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena et Derek Chiou a entraîné un réseau de neurones sur une collection d'applications pour différentes configurations matérielles. Le but de cet entraînement est d'estimer la performance et l'énergie consommée d'une application pour ces différentes configurations matérielles[28]. Les résultats ont montrés que le modèle prédit a permis de prédire ces dernières avec 10 à 15% d'imprécision. Cependant, après l'apprentissage, le modèle estimé par le réseau de neurones artificiels s'exécute plus vite que le programme natif exécuté directement sur le matériel.

12. Le DIMM (*Dual Inline Memory Module*) est un format de barette de RAM, exploité dans la SDRAM et la DRR SDRAM.

13. Un processeur graphique (*Graphics Processing Units*, ou GPU) est un circuit intégré, présent sur une carte graphique, une carte-mère ou un CPU, assurant les fonctions de calcul de l'affichage.

14. `https://fr.wikipedia.org/wiki/Processeur_graphique`

Chapitre 5

Protocoles de mesures de l'énergie

5.1 Généralités

Une étape cruciale, dans la proposition d'une nouvelle méthode ou d'un nouvel algorithme, est leur validation par un protocole de tests. Ce protocole de tests doit être effectué sur des données réelles. Pour notre problème, nous pouvons définir nos données réelles comme : un système d'exploitation Android, ainsi qu'un certain nombre d'applications à tester. Dans ce chapitre, nous étudierons les différentes catégories d'applications à prendre en compte dans les protocoles de validation, tout comme les différents outils permettant de mesurer la consommation énergétique d'une application Android.

5.2 Système Android

Le choix d'une version du système d'exploitation Android, pour les expérimentations, dépendra intuitivement de plusieurs critères. Nous pouvons lister ces derniers :

- la date de sortie de cette version,
- un taux d'acceptation concernant cette version parmi les consommateurs (voir figure 5.1 [1]),
- le nombre d'applications appartenant au système d'exploitation et le retard de performance engendré sur les autres applications (installées par l'utilisateur) via leur fonctionnement en arrière-plan.

Par exemple, compte tenu des derniers résultats donnés par Google (figure 5.1), nous pouvons remarquer qu'une version intéressante du système d'exploitation Android, compte-tenu de sa date de sortie et du taux d'acceptation par le grand public, pourrait être la version 5.1 de Lollipop, contrairement à la version Kit-Kat qui est

1. `http://developer.android.com/about/dashboards/index.html`

Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.7 %
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.5 %
4.1.x	Jelly Bean	16	8.8%
4.2.x		17	11.7%
4.3		18	3.4%
4.4	KitKat	19	35.5%
5.0	Lollipop	21	17.0%
5.1		22	17.1%
6.0	Marshmallow	23	1.2%

FIGURE 5.1 – Représentation des différentes versions Android utilisées. Données collectées mondialement le 1er Février 2016.

plus vieille de deux générations d'API. De plus, cette version contient la nouvelle machine virtuelle d'Android : ART.

Même si l'utilisation pour le grand public est plus restreinte, il n'est pas à exclure les systèmes d'exploitation alternatifs basés sur Android, comme CyanogenMod [2], qui utilise moins d'applications natives.

5.3 Applications Android

Une étude préliminaire doit aussi être effectuée sur le type d'applications à prendre en compte.

Afin de pouvoir instrumenter une application, il est nécessaire d'obtenir le code-source de l'application[10], ce qui peut s'avérer très compliqué pour des applications propriétaires. Ainsi, il peut être nécessaire d'utiliser des applications open-source ou libre de droit. Cependant, de nombreuses solutions proposent directement de travailler sur le *bytecode* de l'application. Par exemple, en décompilant directement le fichier Dex, présent dans l'archive *apk*[29][30], ou alors de "deviner" la consommation énergétique en analysant en mode "ON-LINE" ses caractéristiques matérielles (fréquence du processeur, période de l'application, etc...) dans le cas des applications multimédia mono-threadées[15]. Il est intéressant de constater, d'après des travaux récents, que l'obfuscation de code n'implique pas forcément une consommation éner-

2. `http://www.cyanogenmod.org/`

gétique plus ou moins importante par rapport au code non-obfusqué [31].

Une liste d'applications Android open-source est disponible via un magasin d'applications alternatif au magasin d'applications officiel d'Android (F-Droid [3]), ainsi que sur Github [4].

Le mieux est de différencier les applications d'après leur forme (interactivité, consommatrice CPU, etc...).[32] Ainsi, il serait intéressant de mesurer la performance et la consommation d'énergie pour des applications de type :

- interactive (un jeu-vidéo),
- non-interactive (un lecteur audio/vidéo),
- grande consommatrice CPU (un décodeur vidéo),
- grande consommatrice entrée-sortie (une application de chat, ou un client FTP).

Enfin, la performance de l'application, pour une période de temps donnée, doit être quantifiable.

5.4 Mesure des performances et de la consommation d'énergie

En partant d'une approche système, il faut mesurer, pour une seule application : la fréquence du coeur ou du processeur *big* ou *LITTLE* alloué à la tâche, la performance liée à l'exécution du processeur associé à la fréquence mesurée et l'impact de cette performance sur la batterie de l'appareil. Un des gros problèmes dans ces mesures est l'isolation du processus que l'on veut mesurer, parmi le grand nombre d'applications lancées sur le système. Nous ne voulons pas mesurer les interférences liées au calcul des autres tâches en parallèle, lancées sur le système d'exploitation. Dans notre cas, nous pourrions seulement être intéressé de mesurer la fréquence du processeur ainsi que la caractéristique principale de l'application en fonction de son type : le nombre d'images par seconde pour un jeu-vidéo, un débit entrée-sortie acceptable pour les applications connectées, un temps de calcul acceptable pour une application consommatrice en CPU, etc... La définition d'"acceptable" pourra se faire par la mise en place d'un paramètre jouant le rôle de seuil d'acceptabilité. Si la valeur correspondant à la qualité de service de l'application est inférieure à ce seuil, alors la qualité de service est considérée comme mauvaise, et inversement.

Divers outils existent afin de pouvoir mesurer la fréquence du CPU pour une tâche, la performance de la tâche et l'impact de cette dernière sur la batterie, de façon statique

3. `https://f-droid.org/`
4. `https://github.com/pcqpcq/open-source-android-apps`

ou dynamique. Ces outils peuvent être intégrés directement à l'IDE Android Studio [5], comme `Profile GPU Rendering` [6], `BatteryStats` [7] ou encore `Battery Historian` [8].

Ces derniers peuvent utiliser directement des outils intégrés au noyau Linux. Ainsi, l'outil `OProfile` [9] peut, par exemple, retourner une liste de *logs* concernant le lancement d'une application à une fréquence donnée.

De nouveaux outils sont développés dans des travaux de recherche liés directement à la consommation d'énergie pour les smartphones Android, comme le profiling d'une application par son *bytecode*[30], ou alors en mesurant la consommation énergétique dans les fonctions bas-niveaux [33].

Quant à la mesure de la qualité de service, nous pouvons choisir de mesurer le nombre d'images par seconde (pour une application de type interactive ou non-interactive) avec les outils `dumpsys` [10] ou `framestats` [11], voir même un protocole plus élaboré pour les services de streaming vidéo en ligne[34], dans un laps de temps donné. Il faudra ainsi définir pour cette étape un seuil acceptable de qualité de service.

5. https://developer.android.com/sdk/index.html
6. http://developer.android.com/tools/performance/profile\discretionary{-}{}{}gpu\discretionary{-}{}{}rendering/index.html#ProfileGPURendering
7. http://developer.android.com/tools/performance/batterystats-battery-historian/index.html
8. https://github.com/google/battery-historian
9. http://oprofile.sourceforge.net/news/
10. https://source.android.com/devices/tech/debug/dumpsys.html
11. http://developer.android.com/training/testing/performance.html#fs-data-format

Chapitre 6

Contributions, perspectives et conclusion

Ce travail, effectué avec l'équipe Emeraude du laboratoire CRIStAL, nous a permis de nous forger une meilleure idée de l'organisation du système d'exploitation Android, et du lien unissant le système d'exploitation et une application. De plus, nous avons pu explorer une partie du champ de la gestion de l'énergie sur Android et sur les systèmes d'exploitation temps-réels, des méthodes d'apprentissage automatique permettant de performer la consommation d'énergie en apprenant une politique énergétique sur un système donné, et établir les outils nécessaires pour un protocole de validation des tests. Cette problématique est toujours d'actualité, et les contributions sont toutes aussi intéressantes que diversifiées.

Il pourrai être intéressant d'améliorer les travaux de Pietro Mercati, Andrea Bartolini, Francesco Paterna, Tajana Simunic Rosing et Luca Benini[8], en implémentant un algorithme d'apprentissage qui classifiera automatiquement les applications installées, à partir de critères énergétique (comme la quantité CPU minimale à fournir, ou encore la période de l'application). Cela reviendrai ainsi à apprendre les paramètres à mettre en valeur dans la classification des applications, et réduire les erreurs de classification humaines qui pourraient intervenir à travers l'interface mise en place.
De même, grâce aux travaux de Tommaso Cucinotta, Luca Abeni, Luigi Palopoli et Fabio Checconi, nous pouvons désormais identifier la quantité de CPU minimale à fournir pour un logiciel[15]. En l'adaptant pour le système d'exploitation Android, il pourra être possible de réduire la fréquence du processeur travaillant sur une application à sa fréquence minimale, afin de pouvoir économiser de l'énergie.
Enfin, comme nous l'avons vu, une même application Android pourra consommer une quantité différente de CPU sur 2 architectures matérielles différentes, malgré les mêmes paramètres. Il peut être de mettre en pratique un des modèles donnés

pour l'apprentissage du modèle énergétique pour une application et une architecture matérielle donnée, afin de pouvoir prédire les valeurs des paramètres à prendre en compte, spécifiquement à l'application et à cette architecture.

Personnellement, ce projet de fin d'année m'a permis d'améliorer mes connaissances sur le système d'exploitation Android, les systèmes d'exploitation temps-réel et sur la rigueur quant au choix d'un protocole de validation. De même, il a été très intéressant de pouvoir travailler sur le côté algorithmique et *Machine Learning* de ce problème, notamment sur la modélisation du problème. L'état de l'art établi m'a également donné les clefs pour mieux aborder mon stage de fin d'année sur cette même problématique, à l'Université du Québec À Montréal (UQAM), en étudiant l'impact des bonnes et mauvaises pratiques de code (patterns et anti-patterns) sur la consommation énergétique d'un système Android. Pour finir, ce projet m'a également permis de pouvoir collaborer avec une équipe de recherche dynamique et intéressante sur les problèmes concernant la consommation de l'énergie sur les systèmes embarqués.

Références

[1] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed java-based systems. *IEEE*, 2007.

[2] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Monitoring energy hotspots in software. 2015.

[3] H.E. Zahaf, R. Olejnik, G.Lipari, and A.E. Benyamina. Modeling the energy consumption of soft real-time tasks on heterogeneous computing architectures. January 2016.

[4] Kisoo Yu, Donghee Han, Changhwan Youn, Seungkon Hwang, and Jaechul Lee. Power-aware task scheduling for big.little mobile processor. *IEEE*, 2013.

[5] Samuel Chiang. Advances in big.little technology for power and energy savings. ARM.

[6] Jonathan A. Winter, David H. Albonesi, and Christine A. Shoemaker. 2010.

[7] C. Isci, A. Buyuktosunoglu, C-Y. Cher, P. Bose, and M.Martonosi. An analysis of efficient multi-core global power management policies : Maximizing performance for a given power budget. *In Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, 2006.

[8] Pietro Mercati, Andrea Bartolini, Francesco Paterna, Tajana Simunic Rosing, and Luca Benini. Workload and user experience-aware dynamic reliability management in multicore processors. 2013.

[9] Pietro Mercati, Andrea Bartolini, Francesco Paterna, Tajana Simunic Rosing, and Luca Benini. A linux-governor based dynamic reliability manager for android mobile devices. 2014.

[10] Pi-Cheng Hsiu, Po-Hsien Tseng, Wei-Ming Chen, Chin-Chiang Pan, and Tei-Wei Kuo. User-centric scheduling and governing on mobile devices with big.little processors. 2014.

[11] Borko Furht, Dan Grostick, and al. *Realtime UNIX systems design and application guide*. Kluwer Academic Publishers Group Norwell MA USA, 1991.

[12] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. Energy-aware scheduling for real-time systems : A survey. *ACM Trans. Embed. Comput. Syst. 15*, 2016.

[13] Vincent Legout, Mathieu Jan, and Laurent Pautet. Réduction de la consommation statique des systèmes temps-réel multiprocesseurs. 2014.

[14] Yan Wang, Kenli Li, Hao Chen, Ligang He, , and Keqin Li. Energy-aware data allocation and task scheduling on heterogeneous multiprocessor systems with time constraints. 2014.

[15] T. Cucinotta, L. Abeni, L. Palopoli, and F. Checconi. The wizard of os : a heartbeat for legacy multimedia applications. 2009.

[16] Bhupinder S. Mongia and Vijay K. Madisetti. Reliable real-time applications on android os.

[17] Claudio Maia, Luis Nogueira, and Luıs Miguel Pinho. Evaluating android os for embedded real-time systems.

[18] Yin Yan, Sree Harsha Konduri, Amit Kulkarni, Varun Anand, Steven Y. Ko, and Lukasz Ziarek. Rtdroid : A design for real-time android.

[19] Yin Yan, Chunyu Chen, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. Using a multi-tasking vm for mobile applications.

[20] Tang Lung Cheung, Kari Okamoto, Frank Maker, Xin Liu, and Venkatesh Akella. Markov decision process framework for optimizing software on mobile phones. 2009.

[21] Ying Tan, Wei Liu, and Qinru Qiu. Adaptive power management using reinforcement learning. *ICCAD*, 2009.

[22] G. Dhiman and T. Simunic Rosing. Dynamic power management using machine learning. 2006.

[23] V.L. Prabha and E. C. Monie. Hardware architecture of reinforcement learning scheme for dynamic power management in embedded systems. 2007.

[24] Hao Shen, Ying Tan, Jun Lu, Qing Wu, and Qinru Qiu. Achieving autonomous power management using reinforcement learning. 2013.

[25] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. 2004.

[26] Yanzhi Wang, Qing Xie, Ahmed C. Ammari, and Massoud Pedram. Deriving a near-optimal power management policy using model-free reinforcement learning and bayesian classification. 2011.

[27] Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. Modeling power and energy usage of hpc kernels. 2012.

[28] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. Gpgpu performance and power estimation using machine learning. 2015.

[29] Georey Hecht, Benomar Omar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the software quality of android applications along their evolution. 2015.

[30] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating android applications cpu energy usage via bytecode profiling. *IEEE*, 2012.

[31] Cagri Sahin, Mian Wan, Philip Tornquist, Ryan McKenna, Zachary Pearson, William G. J. Halfond, and James Clause. How does code obfuscation impact energy usage ? *Journal of software : Evolution and Process*, 2016.

[32] Ding Li, Shuai Hao, Jiaping Gui, and William G.J. Halfond. An empirical study of the energy consumption of android applications. *IEEE*, 2014.

[33] Yi-Fan Chung, Chun-Yu Lin, and Chung-Ta King. Aneprof : Energy profiling for android java virtual machine and applications. *IEEE*, 2011.

[34] Ozgur Oyman and Sarabjot Singh. Quality of experience for http adaptive streaming services. *IEEE*, 2012.