Adriano Bernardo de Taveira e Cunha Pinto

Licenciado em Engenharia Informática

Memoization para Poupar Energia em Aplicações Android

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Jácome Cunha, Professor Auxiliar, Universidade do Minho

Co-orientador: Marco Couto, Investigador, Universidade do Minho

Júri

Presidente: Doutor Carlos Damásio
Vogais: Doutor Rui Abreu
Doutor Jácome Cunha

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Junho, 2018

***Memoization* para Poupar Energia em Aplicações Android**

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao meu orientador, professor Jácome Cunha e ao meu co-orientador Marco Couto. Sem eles nada disto teria sido possível. Toda a ajuda no desenrolar desta dissertação foi fundamental e não falo só do aspeto profissional como também do pessoal. São pessoas que levo para a vida porque não só me deram toda ajuda e apoio, como me transmitiram valores e essencialmente acreditaram nas minhas capacidades. Não podia deixar de mencionar o professor João Lourenço, pelos comentários efetuados aquando da apresentação da preparação da dissertação e em particular por ter sugerido abordar o tema de *memoization*.

Gostaria ainda de deixar uma palavra de apreço aos meus colegas de curso. Revelaram-se como autênticos irmãos para mim e sempre que precisei de apoio não hesitaram em ajudar.

Quero ainda agradecer à minha namorada que foi o meu pilar nesta etapa tão importante para mim. Ajudou-me muito em todos os momentos e nunca falhou quando mais precisava. Deixar o meu sincero obrigado à minha família que apesar de não fazerem ideia do que andei a fazer, sabem dizer as palavras certas no momento certo.

Por último, tenho de fazer um agradecimento muito especial a uma pessoa que infelizmente já não se encontra entre nós. O meu avô, um das pessoas que ficará para sempre no meu coração. Uma pessoa cheia de sabedoria, cheia de ideias e essencialmente um grande AVÔ. Acompanhou todo o meu percurso desde que nasci e agora o meu maior desejo era que ele me pudesse ver a atingir esta etapa da minha vida.

Resumo

Ao longo dos últimos anos, o interesse na análise do consumo de energia em aplicações Android tem vindo a aumentar significativamente. Na verdade, há um número considerável de estudos que visam analisar o consumo de energia de várias maneiras, como medir/estimar a energia consumida por uma aplicação ou bloco de código, detectar padrões de código energeticamente dispendiosos, chamadas a APIs e perceber quais os componentes de hardware que mais energia consomem.

No entanto, quando se trata de melhorar a eficiência energética de uma aplicação enfrentamos um novo desafio, que pode ser alcançado através de melhorias no código fonte, aproveitando técnicas de poupança de energia. Contudo, existe alguma escassez de informação sobre tais técnicas e o seu impacto no consumo de energia.

Com esta dissertação, analisamos o impacto da técnica de *memoization* no consumo de energia em aplicações Android. Apresentamos um estudo sistemático sobre o uso de *memoization*, onde comparamos as implementações de 18 métodos de diferentes aplicações, com e sem uso de *memoization*. Neste estudo, mostramos os resultados para 3 métricas: energia, tempo de execução e memória gasta para ambos as versões. Utilizando essa abordagem, foi-nos possível caracterizar os métodos como sendo propícios, imprevisíveis ou impróprios à técnica de *memoization*.

Os nossos resultados mostram que utilizar *memoization* pode ser claramente uma boa abordagem para poupar energia. Além disso, também descobrimos que o tempo de execução melhora, assim como a própria memória consumida. Para os 18 métodos testados, 13 melhoraram em todas as métricas de estudo. Embora a relação entre energia e tempo seja conhecida, neste trabalho descobrimos também que existe uma forte relação entre energia e memória consumida. De facto, um menor consumo de memória (induzido pela técnica de *memoization*) implica também um menor consumo de energia.

Palavras-chave: Android, *Memoization*, Consumo de energia, Tempo de execução, Consumo de memória, Análise de código, *Refactoring*, Optimização de código

Abstract

Over the last few years, the interest in analysing of energy consumption in Android applications has been increasing significantly. Indeed, there are a considerable number of studies to analyse the energy consumption in various ways, such as measure/estimate the energy consumed by an application or block of code, or even detecting energy expensive coding patterns, API's calls and realizing which hardware components consume more energy.

Nevertheless, when it comes to actually improving the energy efficiency of an application, we face a whole new challenge, which can be achieved through source code improvements that can take advantage of energy saving techniques. However, there is still a lack of information about such techniques and their impact on energy consumption.

In this thesis, we analyze the impact of the memoization technique in the energy consumption of Android applications. We present a systematic study of using memoization, where we compare implementations of 18 method from different applications, with and without using memoization. In this study, we show the results for 3 metrics: energy consumption, time and memory spent for both versions. Using this approach, we are able to characterize the methods as being prone, unpredictable or unfit for memoization.

Our results show that using memoization can clearly be a good approach for saving energy. Furthermore, we also find that the execution time improves, as well as the memory consumption. For the 18 methods tested, 13 improved in all study metrics. In spite of the relation between energy and time is already known, in this work we also find that there is a strong relation between energy and memory consumed. In fact, lower memory consumption (induced by the memoization technique) also implies a lower energy consumption.

Keywords: Android, Memoization, Energy Consumption, Time, Memory, Code Analysis, Refactoring, Code optimization

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Capítulo 1

Introdução

Nos dias de hoje, e cada vez mais, a maior parte da população é detentora de um *smartphone*. Além disso, a tecnologia tem evoluído com o passar dos anos e hoje em dia o telemóvel não é utilizado somente para as funcionalidades básicas, como a troca de chamadas e mensagens, mas também para navegar na Internet, aceder ao email, descarregar aplicações e jogar. Os *smartphones* são geridos por um sistema operativo, dos quais se destacam o Android e o iOS. No entanto, no nosso trabalho focámo-nos no sistema Android uma vez que é aquele que mais mercado tem vindo a abranger nos últimos anos [65]. Na última década, a sua utilização foi significativamente notória, tendo sido lançadas 12 versões principais do sistema nos últimos 9 anos [3].

Com o crescimento do Android, surgiram novas e variadas aplicações (o número de aplicações disponíveis na Google Play aumentou de 30K para 3.5M nos últimos 7 anos [51]), tendo o número de *downloads* realizados através da Google Play aumentado de 1B par 65B, entre 2010 e 2016 [50]. As aplicações têm evoluído cada vez mais, existindo uma maior complexidade e uma maior exigência por parte do utilizador. Aplicações mais poderosas e tarefas mais pesadas implicam a existência de hardware ao mesmo nível para colmatar estas necessidades. No entanto isso também faz com que a bateria (que delimita o tempo de utilização do telemóvel) dure menos.

Desta forma é preciso agir para ir ao encontro das necessidades dos utilizadores. É neste ponto que o interesse em poupar energia se estende aos programadores das aplicações. Embora cada vez mais comecem a existir práticas de programação que visam uma poupança energética, é ainda possível fazer muitas melhorias ao código das aplicações [57].

A nossa abordagem passa por aplicar a técnica de *memoization* [1] no código fonte das

[1] Daqui em diante neste documento, quando nos quisermos referir à versão de um método que utilize *memoization*, iremos usar o termo *memoized*, para estar em concordância linguística.

aplicações para conseguirmos melhorar o consumo de energia e ainda diminuirmos o tempo de execução e a memória gasta.

1.1 Contexto e Motivação

Nos últimos anos, a maioria dos trabalhos realizados nesta área focou-se em detetar ou prever o consumo de energia que é despoletado por um componente de software [31, 32]. Alguns deles apresentaram técnicas para monitorizar e classificar o consumo de energia em partes de código, como por exemplo, linhas de código [40], métodos [11], chamadas a APIs [44, 45, 53, 56], ou mesmo padrões de código [42, 62]. São ainda analisados os impactos energéticos de técnicas que visam melhorar a performance de aplicações Android [63]. Até mesmo a energia consumida na fase de testes pode ser uma preocupação [41], assim como a exibição de elementos visuais nas interfaces do utilizador de uma aplicação [14].

Foram ainda realizadas alterações automáticas de código para melhorar a eficiência energética de aplicações Android [15], recorrendo a uma ferramenta desenvolvida para um outro trabalho [55].

Outras contribuições têm sido feitas nesta área e num outro estudo [43], os autores conseguiram perceber quais os componentes que mais energia consomem (Rede, Câmara, operações da interface do utilizador) para uma série de 405 aplicações Android reais. Existem ainda trabalhos que se focam mais no *core* do Android e como este funciona [39, 48]. Uma vez que estes últimos não se focam em aplicações mas sim no núcleo do Android, distanciam-se um pouco do nosso trabalho. Assim, apesar de existirem artigos que nos ajudam com as suas descobertas importantes, aquilo que pretendemos com este trabalho ainda não foi realizado.

Além disso, a quantidade de informação existente sobre o que pode ser utilizado pelos programadores para reduzir o consumo de energia, na fase de desenvolvimento de uma aplicação, ainda é muito reduzida.

1.2 Solução Proposta

Nesta dissertação, apresentamos um estudo sistemático que mostra os ganhos de energia obtidos da utilização da técnica de *memoization* em aplicações Android. Para decidir os métodos dessas mesmas aplicações que são passíveis de sofrer *memoization*, utilizámos a técnica proposta por Yang et al. [68] que permite selecionar métodos com as características necessárias à *memoization*. Depois de filtrar esses métodos em várias aplicações Android, estes foram colocados numa aplicação por nós criada, denominada de original. À posteriori aplicámos a técnica de *memoization* e passámos a ter duas aplicações nossas. Deste modo, executámos alguns testes para poder comparar a aplicação original com a *memoized*.

Os nossos resultados mostram que com *memoization* é possível melhorar significativamente a eficiência energética de uma aplicação Android, sem ameaçar o seu normal

funcionamento e/ou eficiência. Os resultados mostram ainda que esta eficiência muitas vezes é melhorada, ou seja, descobrimos que o tempo de execução também melhorou, assim como a própria memória consumida.

Ao longo desta dissertação pretendemos responder às seguintes *research questions*:

RQ1 Será que a técnica de *memoization* pode ser utilizada para reduzir o consumo de energia em aplicações Android?

RQ2 Será que a técnica de *memoization* também contribui para a diminuição do tempo de execução?

RQ3 O que acontecerá à memória consumida pelo dispositivo aquando da aplicação da técnica de *memoization*?

1.3 Organização do Documento

O restante documento, está organizado da seguinte forma:

- No capitulo 2 é apresentado o estado da arte e em que medida é que o trabalho já efetuado difere do trabalho por nós realizado. São tidas em conta contribuições importantes de trabalhos anteriores e que nos foram úteis na realização do nosso trabalho.

- No capítulo 3 encontra-se explicada a solução por nós proposta, assim como, a experiência efetuada para a obtenção de resultados.

- No capítulo 4 apresentamos os resultados obtidos com a experiência apresentada no capitulo 3.

- No capítulo 5 avaliamos os resultados obtidos anteriormente e discutimos em detalhe cada um deles.

- No capítulo 6 elaboramos uma conclusão que contempla o que se obteve com a realização deste trabalho e sumariza-se um possível trabalho futuro.

Capítulo 2

Estado da Arte

Neste capítulo são descritos alguns trabalhos que têm como objetivo perceber qual o impacto de algumas técnicas de programação na performance de aplicações Android, assim como no consumo de energia dos *smartphones*. Na secção 2.1 discutem-se algumas propostas de práticas a serem seguidas aquando da elaboração de código. A maior parte das propostas apresentadas baseiam-se no site oficial para programadores Android [29]. Contudo estas são mais direcionadas ao melhoramento da performance e não tanto ao consumo energético. Na subsecção 2.1.1 é feita uma comparação entre algoritmos iterativos e recursivos. Na subsecção 2.1.2 são tidos em conta diversos algoritmos de ordenação e realizada uma comparação entre eles. Na subsecção 2.1.3 são comparados os algoritmos que fazem uso mais intensivo da memória com aqueles que são mais intensivos na utilização do CPU. Na subsecção 2.1.5 é feita uma introdução da técnica de *memoization* e são apresentados trabalhos que a utilizam no seu estudo. Na subsecção 2.1.4 é abordado o tema dos pedidos HTTP e como estes podem influenciar o consumo de energia. Na secção 2.2 falamos de *code smells* e apresentamos o impacto destes no consumo energético do telemóvel. Por último, a secção 2.3 retrata algumas das ferramentas utilizadas na medição de energia de dispositivos Android. Esta encontra-se dividida em duas subsecções, na medida em que considerámos duas categorias de ferramentas (hardware/software).

2.1 Relação entre Práticas de Programação e Consumo Energético

2.1.1 Algoritmos Iterativos *vs* Algoritmos Recursivos

A comparação entre os algoritmos iterativos e recursivos verifica-se num estudo de Vieira et al. [67], para o sistema Android com a versão 2.3. Os autores comparam algoritmos iterativos e recursivos da mesma complexidade tanto para o algoritmo de ordenação insertion sort como para o cálculo factorial. Para o insertion sort iterativo houve uma melhoria de 38% em relação ao tempo de execução do algoritmo recursivo e em termos de energia consumiu-se cerca de 43% menos, como demonstrado na Figura 2.1.

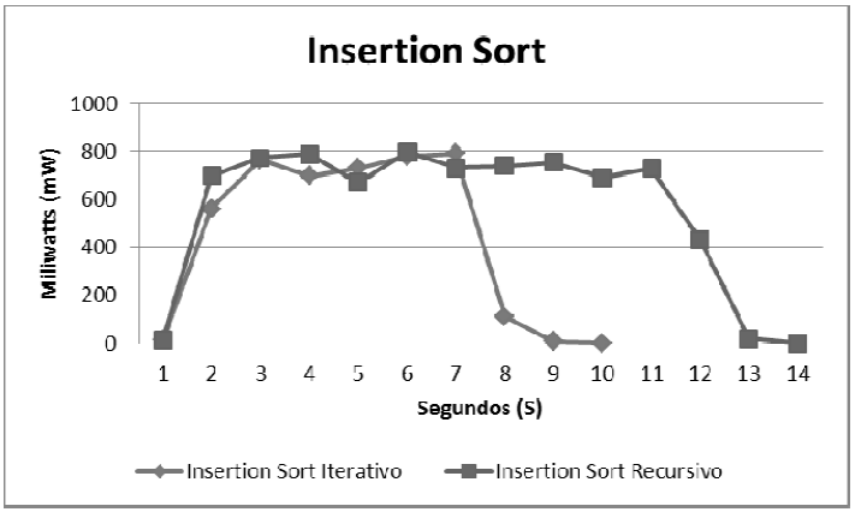

Figura 2.1: Comparação entre o insertion sort iterativo e recursivo [67]

O algoritmo do factorial iterativo obteve uma melhoria de 20% no que diz respeito ao tempo de execução e um consumo de energia 24% menor, como se pode ver na Figura 2.2. É percetível que uma implementação iterativa obtém um tempo de execução menor (melhor performance) e ainda um menor consumo de energia relativamente à implementação com recursividade. Isto deve-se ao facto dos algoritmos recursivos utilizarem de forma intensiva a pilha do sistema Android que faz com que tenham de existir alocações e desalocações de memória (empilhar e desempilhar, respetivamente). Assim, este artigo motiva o uso de algoritmos iterativos. Contudo, e uma vez que só se usaram dois algoritmos, não se podem extrapolar conclusões sobre a relação entre performance e energia. Pelo contrário, no nosso trabalho analisaremos código de aplicações reais e aplicaremos a técnica de *memoization* desde que seja possível.

2.1.2 Algoritmos de Ordenação

A comparação de algoritmos de ordenação no que diz respeito ao consumo de energia e à sua performance foi alvo de estudo por parte de Vieira et al. [67], onde compararam

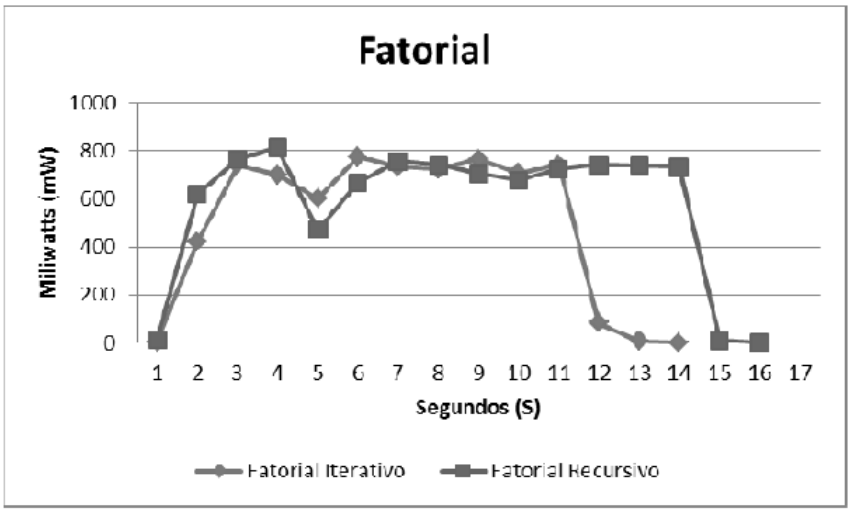

Figura 2.2: Comparação entre fatorial iterativo e recursivo [67]

algoritmos como o Bucket Sort, o Counting Sort, o Heap Sort, o Insertion Sort Iterativo, o Insertion Sort Recursivo e o Merge Sort. De entre estes seis algoritmos, o mais rápido foi o Counting Sort consumindo também menos energia (demorou 5 segundos a executar e gastou somente 0.5J de energia), como se pode verificar na Figura 2.3.

No entanto, considerando apenas os algoritmos em comum deste e de um outro estudo [6] (insertion sort, merge sort, heap sort), percebe-se que o resultado é díspar. Assim, enquanto que no artigo de Vieira et al. [67] é referido que o algoritmo mais eficiente em termos energéticos é o heap sort, no artigo de Bunse et al. [6] conclui-se que é o insertion sort. É de realçar que o insertion sort teve um pior resultado, comparativamente ao merge sort no estudo elaborado por Vieira et al. [67]. Apesar do desempenho ser semelhante (ambos demoraram 10 segundos para ordenar o conjunto de dados), o consumo energético foi diferente, tendo o insertion sort consumido cerca de 4.5J de energia e o merge sort 3.7J.

A par destes resultados, é ainda importante referir que Bunse et al. [6] concluem que nem sempre uma melhor eficiência em termos de performance implicam um melhor consumo energético. Os autores mencionam assim, que o consumo de energia é influenciado por diversos fatores, tais como o consumo da memória e a performance, onde o algoritmo mais rápido não é de todo o mais eficiente em termos de consumo energético. Para o artigo em questão, o algoritmo de ordenação mais rápido a executar foi o quicksort e no entanto, ao contrário do que seria expectável, não foi o mais eficiente no que diz respeito ao consumo de energia. No nosso trabalho iremos demonstrar que existe de facto uma relação muito forte entre consumo de energia e memória gasta.

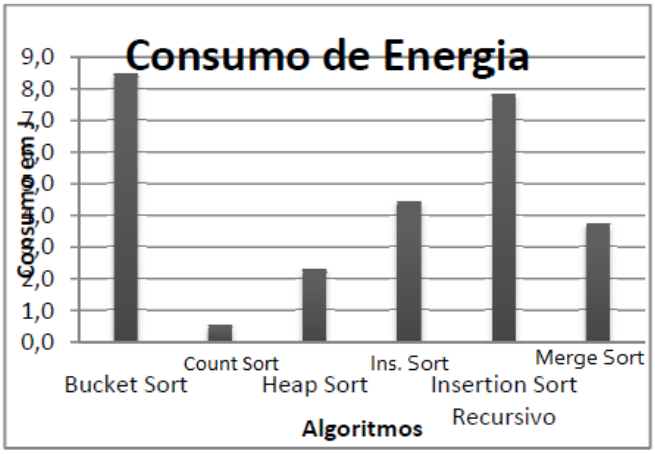

Figura 2.3: Comparação entre os vários algoritmos de ordenação [67]

2.1.3 Memória vs CPU

A análise dos algoritmos que fazem uma utilização mais intensiva da memória e aqueles que utilizam mais o CPU foi alvo de estudo por parte de Vieira et al. [67]. O exemplo dado no seu trabalho é para a função seno, em que num primeiro caso foram armazenados os valores obtidos para o seno desde 0 a 90 graus com intervalos de 0.1, obtendo uma *hash table* com 900 valores – seno tabelado. Num segundo caso, o programa apresentado pelos autores tem a necessidade de estar constantemente a realizar todos os cálculos, de forma exaustiva, cada vez que se deseja obter um determinado valor da função seno – seno calculado. Comparando os resultados ao nível do gasto energético por unidade de tempo, conclui-se que o seno tabelado obteve um consumo de energia de 90% menos, relativamente ao seno calculado, visível na Figura 2.4.

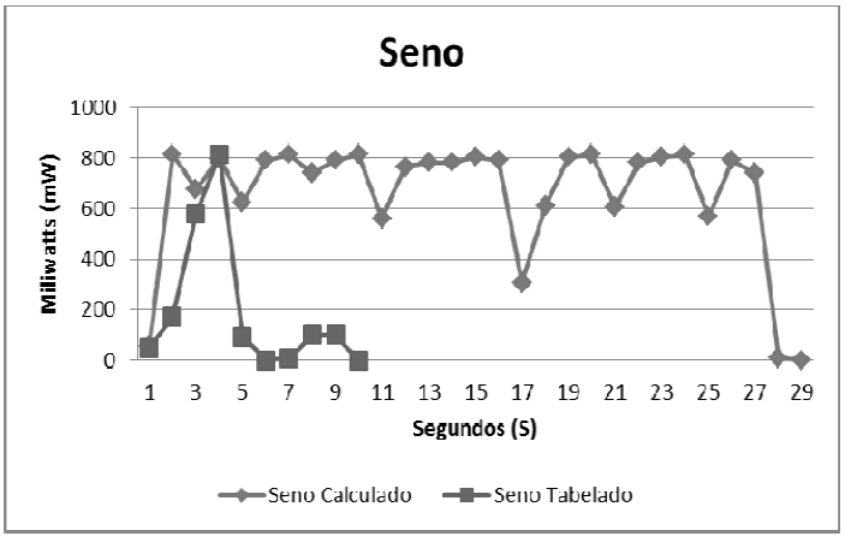

Figura 2.4: Comparação entre Seno Tabelado e Calculado [67]

Para um outro trabalho, Li et al. [42] realizaram um estudo em relação à utilização da memória. Os autores criaram um pequeno programa, mostrado na Figura 2.5, onde foi avaliado o consumo de energia. O tamanho do vetor exibido no código foi variável entre números inteiros de 512 a 5.120.000. O N do ciclo foi definido para 100.000, para assegurar que a maior parte do consumo de energia provinha do ciclo, tentando isolar da melhor maneira o programa.

```
1  protected void onCreate(Bundle savedInstanceState) {
2      super.onCreate(savedInstanceState);
3      setContentView(R.layout.activity_main);
4      int[] array=new int[size]
5      for(int k=0;k<N;k++)
6          for(int i=0;i<size;i++)
7          {
8              array[i]=1;
9          }
10 }
```

Figura 2.5: Código efetuado para testes da memória [42]

Em relação aos resultados do programa apresentado, Figura 2.6, estes revelam que o valor médio de energia consumida por cada acesso a uma determinada posição do vetor aumenta, quando o tamanho do vetor também aumenta. No entanto, este aumento é pequeno e pouco significativo. No fundo, para um aumento da memória de 10000 vezes, apenas existe um aumento de 21.7% em termos energéticos por cada acesso ao vetor. Deste modo é feito um encorajamento por parte dos autores, no que diz respeito a sacrificar mais a memória se isto se refletir num menor consumo energético por parte dos outros componentes. Um exemplo dado passa por alocar uma *cache* maior para reduzir o número de acessos à rede.

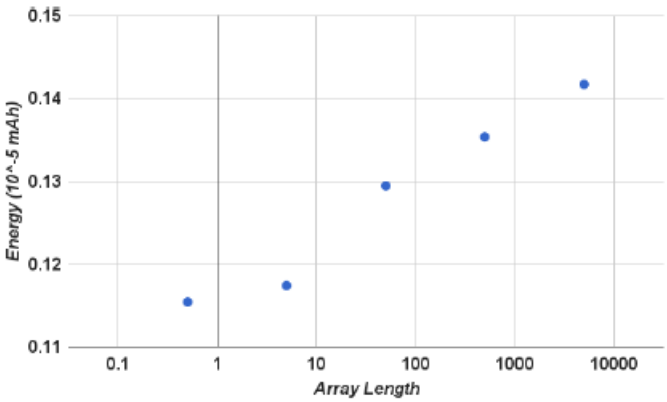

Figura 2.6: Consumo energético para diferentes níveis de utilização da memória [42]

2.1.4 Pedidos HTTP

Os pedidos via HTTP são um ponto-chave nas aplicações Android, na medida em que é um dos métodos mais importantes para aceder à Internet. Assim, o que já tinha sido demonstrado é que transmitir quantidade maiores de dados é mais eficiente do que transmitir pequenas parcelas [17]. No entanto, não existia qualquer tipo de indício quanto à eficiência energética. Contudo, um estudo efetuado relacionou a forma como se efetuam os pedidos HTTP com a energia gasta pela utilização da rede *Wi-Fi*, para a versão 4.2.2 do Android. Assim, Li et al. [42] realizaram este estudo e concluíram que agrupar os pequenos pedidos HTTP num único pedido permitia uma poupança energética significativa. Mais detalhadamente, este estudo passou por medir o consumo de energia obtido com o download de diferentes tamanhos de dados através de pedidos HTTP. O programa para o efeito é mostrado na Figura 2.7, onde se percebe que o pedido à rede foi executado N vezes (os autores definiram o valor de 1000 para o número de execuções).

```
1  protected void onCreate(Bundle savedInstanceState) {
2      super.onCreate(savedInstanceState);
3      setContentView(R.layout.activity_main);
4      URL url = new URL(resourcelink);
5      for(int i=0;i<N;i++)
6      {
7        URLConnection urlConn = url.openConnection();
8        InputStream in = new BufferedInputStream(urlConn.
               getInputStream());
9        try {
10           readStream(in);
11       finally {
12           in.close();
13       }
14     }
15     finish();
16 }
```

Figura 2.7: Código efetuado para testes dos pedidos HTTP [42]

Um reparo a ter em conta é o facto de terem utilizado a classe HttpURLConnection para efetuar o pedido HTTP, em vez da classe HTTPClient. De facto, é preferível a utilização da primeira opção [22] e sendo assim apenas esta foi verificada para os estudos. Por forma a iniciar as medições, o programa exibido foi executado no *smartphone* com o ciclo vazio, de modo a perceber e a isolar os custos. De seguida correram então o programa na íntegra e realizaram o download de um conjunto de ficheiros a partir do servidor com um número variável de bytes (1 a 4000 bytes). Por cada download foi então calculada a energia consumida, posteriormente subtraída à energia base (ciclo sem a chamada HTTP) para obter o custo de acesso à rede. O resultado é apresentado na Figura 2.8, onde é possível constatar que os valores de energia se mantêm razoavelmente estáveis quando os dados de download são inferiores a, aproximadamente, 1024 bytes. Há medida que o tamanho dos dados aumenta, cria-se uma relação linear com a energia consumida.

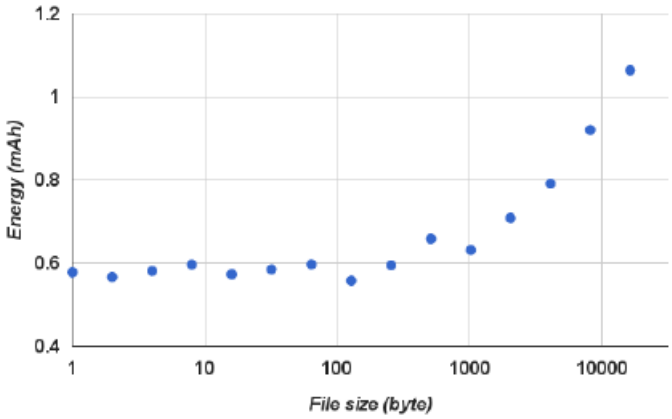

Figura 2.8: Consumo energético para o download de ficheiros com diferentes tamanhos [42]

A razão inferida pelos autores para este acontecimento, prende-se com o facto do protocolo HTTP e dos seus protocolos de nível inferior, TCP e IP, possuírem uma sobrecarga fixa, introduzida pelo cabeçalho dos pacotes e por informação de controlo. Este cabeçalho tem um tamanho fixo, independentemente dos dados transmitidos. Já o tamanho da informação de controlo depende essencialmente do número de pacotes, ou seja, da quantidade de dados enviados. Assim, o consumo de energia é dominado pela transmissão de cabeçalhos de pacotes e informações de controlo quando o tamanho do pacote é pequeno. À medida que a quantidade de dados aumenta, a situação inverte-se e o tamanho dos dados passa a dominar. Tendo em conta novamente os resultados obtidos, concluiu-se que agrupar pequenos pedidos HTTP pode poupar energia relativamente à despesa que se tem ao enviar um único pedido solto. Se para 1 byte se consome 0.58mAh e para 1000 bytes se consome 0.61mAh (valores aproximados), então é vantajoso em termos energéticos, agrupar os vários pedidos e enviar 1000 bytes de uma só vez do que enviar byte a byte.

2.1.5 *Memoization*

Em ciências da computação, *memoization* é uma técnica de otimização utilizada principalmente para acelerar os programas de computador. A técnica consiste em armazenar os resultados de chamadas de funções dispendiosas em *cache* e retornar estes valores quando a mesma função é chamada para um mesmo *input* [1, 68]. Este conceito será explorado mais à frente no capítulo da solução proposta, uma vez que foi a nossa abordagem para a realização desta dissertação.

De seguida apresentamos alguns trabalhos onde esta técnica foi utilizada. No entanto, o contexto diferiu um pouco do nosso como vamos poder constatar.

Yang et al. [68] propuseram uma técnica e uma ferramenta para determinar a pureza funcional dos métodos Java, de forma a determinar com mais facilidade quais métodos poderiam ser alterados e onde poderia ser aplicada a técnica de *memoization*. Em particular,

se um método é uma função pura, então pode ser *memoized*. No seu estudo, eles conseguiram efetuar *memoization* com sucesso em vários métodos de 3 bibliotecas Java diferentes e reduzir o seu tempo de execução. No entanto, a utilização da memória consumida também aumentou. Na nossa experiência a diferença mais importante é que mostramos que a técnica de *memoization* também tem um impacto positivo no consumo de energia e que a memória consumida diminuiu, ao contrário dos resultados obtidos para este trabalho apresentado.

Uma abordagem bastante semelhante também foi proposta por Agosta et al. [1]. No seu trabalho, eles também definiram quais os métodos Java onde se pode aplicar *memoization* com base na sua pureza funcional. Ambas as definições são semelhantes. A avaliação foi realizada utilizando funções financeiras e obtiveram bons resultados tanto para a energia como para o tempo. Muito interessante também é o fato de terem definido um modelo teórico para prever a eficácia da aplicação da técnica de *memoization* em termos de consumo de energia. No entanto, o seu estudo foi aplicado a um determinado conjunto de cálculos em um computador e não para um dispositivo móvel.

2.2 Impacto de *Code Smells* no Consumo Energético

Nesta secção são abordados alguns dos blocos de código conhecidos por reduzir a performance de uma aplicação (podem causar perda de recursos ao nível do CPU, memória e bateria) e por terem um grande impacto no que diz respeito ao consumo de energia. Estes serão referidos daqui em diante como *code smells*.

Os *code smells* podem ainda ser vistos como más práticas de programação que levam a uma qualidade baixa do software desenvolvido. A maior parte destes blocos de código não foram escolhidos ao acaso, uma vez que já tinham sido identificados por terem um impacto negativo na performance do Android [34].

2.2.1 Tipos de *Code Smells*

Ao longo desta subsecção descrevemos dois tipos de *code smells*:

1. ***Code smells* de performance do Android:**

   - **Internal Getter/Setter (IGS)** - ocorre quando se acede a um campo dentro da própria classe através de um método de *get* ou *set*, em vez do acesso direto ao mesmo [7, 34, 42, 47, 63, 66]. É importante referir que este *code smell* se encontra obsoleto, uma vez que foi reportado pela *Google* que já não tem qualquer efeito na performance, desde a versão 2.3 do Android [14, 21].
   - **Member Ingoring Method (MIM)** - é recomendada a utilização de métodos estáticos que visam o aumento da performance, em situações em que um método não acede ao atributo de um objeto ou não se trata de um construtor da

classe [7, 34, 42]. Métodos deste tipo são potencialmente propícios à utilização de *memoization* e por isso a nossa técnica pode ser aplicada neles.

- **HashMap Usage (HMU)** - é de evitar a sua utilização quando existe pouca quantidade de dados (até alguns milhares de valores). Assim, a não ser que seja realmente necessário um mapa mais complexo para um conjunto grande de objetos, é recomendada a utilização do ArrayMap [24] e do SimpleArrayMap [30] como alternativa uma vez que são mais eficientes em termos de memória e despoletam menos *garbage collection* [7].
- **Array Length** - deve-se evitar aceder ao atributo *length* de um vetor no corpo de um ciclo. Assim, fora do corpo do mesmo, guarda-se numa variável o valor do tamanho do vetor e utiliza-se posteriormente essa mesma variável [42, 47, 66].
- **DrawAllocation** - é uma má prática alocar objetos durante uma operação de desenho ou *layout*. Esta alocação pode causar operações de *garbage collection* fazendo com que uma interface deixe de ser suave (*smooth*). A solução recomendada é a alocação de objetos *up front* e reutilizá-los para cada operação de desenho [14].
- **WakeLock** - são mecanismos para controlar o estado de energia do *smartphone*. Assim, podem ser usados para "acordar" o ecrã ou o CPU com o objetivo de executar tarefas mesmo quando o dispositivo está em estado de suspensão. Se uma aplicação não libertar um *wake lock* ou usá-lo sem ser estritamente necessário, esta pode consumir a bateria do *smartphone* [14].
- **Recycle** - muitos recursos, tais como TypedArrays e VelocityTrackers, devem ser reciclados depois de serem utilizados. Assim, se forem libertados, podem voltar a ser usados desde esse ponto [14].
- **ObsoleteLayoutParam** - Durante o desenvolvimento de uma aplicação, as vistas da interface do utilizador (UI) são modificadas várias vezes. Com todo este processo, existem parâmetros que são deixados inalterados, não influenciando a vista atual. Isto causa um processamento inútil de atributos em tempo de execução. Assim sendo, tais parâmetros devem ser completamente removidos para não criar incoerências [14].
- **ViewHolder** - é um padrão que é utilizado para tornar o scroll em List Views mais suave, uma vez que numa List View o sistema precisa de desenhar cada elemento da lista. Para tornar este processo mais eficiente, os dados do elemento anteriormente desenhado podem ser reutilizados. Deste modo, o número de chamadas ao método findViewById (é conhecido por ser um método dispendioso) diminui com a utilização desta técnica [14].
- **Overdraw** - pintar regiões mais do que uma vez, ou seja, o mesmo pixel ser desenhado várias vezes. Isto leva a um processamento desnecessário e pode

ser melhorado removendo o fundo das vistas ou recortando o desenho quando possível [14].

- **UnusedResources** - recursos, como ícones ou elementos da interface gráfica do utilizador, podem tornar-se obsoletos com as alterações que vão sendo efetuadas no programa. Assim, estes devem ser removidos caso não estejam a ser utilizados [14].
- **UselessParent** - uma vez que os *layouts* de interface apresentam várias alterações ao longo do processo de desenvolvimento, por vezes tornam-se inúteis. Assim, estes podem eventualmente ser substituídos por um descendente (*layout* "filho" no lugar do "pai") [14].

2. ***Code smells* de imagem:**

- **Picture Format** - a utilização do formato PNG é conhecida como uma má prática quando se tratam de imagens de grande tamanho e com muitas cores. Nestes casos, o uso do formato JPG ou GIF é preferencial. Deste modo, o programador deve decidir que formato usar, dependendo do contexto [7] .
- **Picture Size** - um problema comum é o *trade-off* existente entre o tamanho das imagens e a sua qualidade. Assim, o objetivo passa por reduzir o tamanho do ficheiro mas mantendo a qualidade da imagem, para esta não ser degradada [7].
- **Picture Bitmap Usage** - os programadores Android devem considerar a forma como guardam e leem os bitmaps, utilizando por exemplo o formato RGB_565, em vez do ARGB_8888. Este último é o pré-definido para a versão KITKAT do Android [25]. A utilização de formatos dispendiosos pode ser considerado como uma má prática em situações onde a imagem exibida na aplicação é demasiado pequena para se perceber a diferença relativamente à imagem que adotou uma baixa densidade dos pixéis [7].

2.2.2 Correção de *Code Smells* no Android

A *Google* fornece uma ferramenta que pode ser utilizada para analisar o código e detetar uma lista de *code smells*, o *lint* [21]. Contudo, para a sua correção foram desenvolvidas outras ferramentas.

Carette et al. [7] desenvolveram uma ferramenta chamada de HOT-PEPPER para detetar e corrigir alguns destes *code smells* e avaliar o seu impacto energético de forma automática em aplicações Android. Esta ferramenta está divida em duas componentes: a componente de deteção destes blocos de código, PAPRIKA [33], que foi posteriormente melhorada para também os corrigir. A segunda componente da ferramenta, NAGA VIPER, que foi desenvolvida neste artigo para avaliar o impacto de cada versão do APK [1] de cada

[1] O nome advém de *Android Package* e trata-te de um ficheiro compilado que é utilizado para instalar programas no sistema Android [23].

aplicação Android. Ou seja, esta componente de software, recebe todas as versões de APK para uma dada aplicação (geradas pelo PAPRIKA, consoante a existência ou não de *code smells*), um cenário que simula uma possível utilização real da aplicação e um *smartphone* Android com a versão 4.4.4 ligado a um medidor físico de energia, o Yocto-Amp [69]. No fim, são comparadas as diferentes versões da aplicação, o código fonte associado e a lista de correções efetuadas pela ferramenta PAPRIKA, com o objetivo de devolver a versão mais eficiente em termos energéticos. Todo o funcionamento do sistema é demonstrado na Figura 2.9.

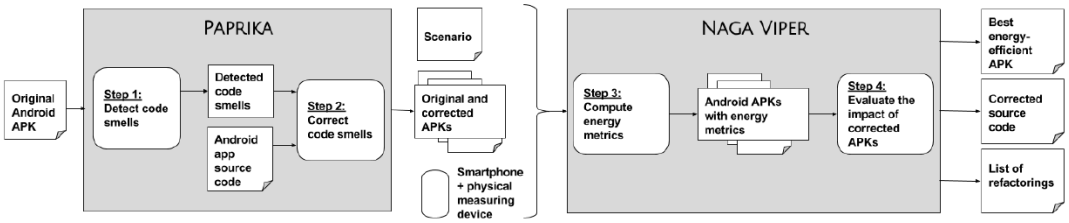

Figura 2.9: Ferramenta HOT-PEPPER [7]

Uma outra ferramenta de correção automática dos *code smells* chama-se Leafactor e foi desenvolvida por Cruz e Abreu [13]. Esta permite melhorar o consumo energético em aplicações Android, tal como a anterior. Isto torna-se possível com a alteração do código para padrões conhecidos como sendo eficientes em termos energéticos.

Os autores utilizaram um conjunto de 140 aplicações *open-source*, obtidas através do F-droid (um repositório de aplicações Android open source) [18] e validaram a sua ferramenta num conjunto total de 222 alterações de código. A ferramenta foi criada baseando-se no projeto *open-source*, *AutoRefactor*[2]. Os autores já tinham testado e corrigido os *code smells* [14]. No entanto, tinham realizado esse trabalho de forma manual, sem recurso a nenhuma ferramenta. Na Figura 2.10 é possível perceber o número de correções efetuadas pelo Leafactor [13] para o conjunto total de aplicações analisadas.

Optimization Rule	W	R	DA	VH	OLP
Total Refactors	1	58	0	7	156
Affected Projects	1	23	0	5	30
Affected Projects (%)	1	16	0	4	21

Wakelock (W), Recycle (R), DrawAllocation (DA), ViewHolder (VH), ObsoleteLayoutParam (OLP)

Figura 2.10: Ferramenta Leafactor [13]

Segundo os autores, esta ferramenta está preparada para receber um único ficheiro, um pacote ou um projeto Android completo como *input*. A sua função é procurar nos ficheiros Java e XML, os *code smells* identificados, com o objetivo de os corrigir de forma automática, gerando novas versões compiladas e otimizadas.

[2]Este plugin do Eclipse, permite a alteração automática de código Java, fornecendo uma API para manipular *Abstract Synstax Trees (AST) do Java.*

2.2.3 Resultados

Com o intuito de perceber o impacto energético de alguns dos *code smells* da categoria de performance, nomeadamente do Internal Getter/Setter (IGS), Member Ingoring Method (MIM) e HashMap Usage (HMU), explicados na subsecção 2.2.1, os autores do artigo [7] analisaram 5 aplicações Android obtidas a partir do F-droid [18].

As 5 aplicações foram avaliadas e estudadas para as cinco versões da aplicação inicial, como se percebe pela Figura 2.11. Os cenários de utilização, simulando eventos despoletados pelo utilizador, foram executados 20 vezes cada para cada uma das versões. Por último, os autores recolheram 75 valores de intensidade energética por segundo que juntamente com o tempo de execução da aplicação permitiram computar as métricas energéticas pela ferramenta NAGA VIPER.

Assim obteve-se o valor do consumo global de energia para cada aplicação e conclui-se que este foi melhor aquando das alterações efetuadas nos *code smells*, Figura 2.12.

Version	Corrected Code Smells
V_0	None
V_{HMU}	HashMap Usage (HMU)
V_{IGS}	Internal Getter/Setter (IGS)
V_{MIM}	Member Ignoring Method (MIM)
V_{ALL}	All (IGS + MIM + HMU)

Figura 2.11: Versões das várias aplicações [7]

Apps	HMU	IGS	MIM	ALL
Aizoban	**-2.00%**	-1.09%	+0.08%	-1.38%
Calculator	-	-0.18%	-0.45%	**-1.69%**
SoundWaves	-0.38%	**-1.43%**	+0.29%	-1.29%
Todo	-2,40%	-2,04%	-	**-4,83%**
Web Opac	-2.06%	-2.08%	**-3.86%**	-3.50%

Figura 2.12: Resultados da melhoria do consumo energético [7]

O melhor resultado foi obtido pela aplicação "Todo" quando foram alterados todos os *smells* presentes, resultando numa poupança de **4.83%** em termos energéticos.

Com a correção do *code smell* MIM, o melhor resultado foi uma poupança de **3.86%** para a aplicação "Web Opac" no artigo [7], enquanto no artigo [42] foi de **15%** para uma aplicação fictícia. Para o *code smell* IGS, o melhor resultado foi uma poupança de **1.43%** para a aplicação "SoundWaves" no artigo [7], enquanto no artigo [42] foi de **30 a 35%** para uma aplicação fictícia.

Em ambos os *code smells* (MIM e IGS), os autores do artigo [7], justificam a sua diferença de resultados, comparativamente aos autores do artigo [42] com o facto do contexto de experimentação ter sido bastante diferente. Ou seja, num caso os autores utilizaram aplicações reais [7] e no outro apenas se basearam numa aplicação criada para o efeito [42]. Além disso, enquanto os autores do artigo [7] efetuaram os seus testes com base em

cenários reais de utilização das aplicações, os outros autores [42] correram cada invocação de um *code smell* 50 milhões de vezes.

A conclusão obtida pelos autores do artigo [7] foi que realmente a correção de pelo menos um *code smell* reduz o consumo de energia do *smartphone*. Graças ainda aos resultados obtidos através do Cliff's δ effect size concluíram que a correção de todos os *code smells* numa aplicação tem um melhor impacto do que apenas a correção de um deles.

Na categoria de *smells* de imagem não foram utilizadas aplicações reais, na medida em que não era possível o acesso às imagens das mesmas. Assim, para este caso, os autores do artigo [7] utilizaram 7 aplicações customizadas em que o formato das imagens, o tamanho e a configuração do seu bitmap variavam de modo a avaliar o seu impacto energético. Neste caso, desenvolveram um cenário que simulasse a navegação nas aplicações, assim como o toque em diversas imagens. Desta vez optaram por correr 30 vezes cada cenário para cada versão da aplicação e os valores obtidos pela ferramenta NAGA VIPER são obtidos da mesma forma.

Algumas conclusões diferentes do que se estava à espera foram obtidas, por exemplo, o formato das imagens não tem qualquer influência no consumo global de energia de uma aplicação Android e ao contrário do que se esperava, segundo as dicas de performance para Android, o formato mais "pesado" de bitmap (ARGB_8888) obteve um melhor consumo energético quando comparado com o mais "leve" (RGB_565). De forma sucinta, utilizar a melhor compressão de imagem, juntamente com o formato de bitmap pré-definido no Android tem um impacto positivo na performance e no consequente consumo energético do *smartphone*.

Como conclusão do trabalho apresentado [7], este foi bem conseguido para os três *code smells* (HMU, IGS, MIM) analisados para cinco aplicações Android. Existiu uma aproximação da realidade, inclusive nos cenários que simulavam uma utilização possível por parte de um utilizador. Contudo, para os restastes três *smells* das imagens, estes não foram avaliados da melhor forma uma vez que foram baseados numa aplicação feita para este propósito.

Tal como Carette et al. [7], também Cruz e Abreu [14] utilizaram aplicações disponibilizadas pelo F-droid [18]. Contudo, os critérios de seleção foram um pouco diferentes, escolhendo as aplicações que se encontravam em desenvolvimento continuo e aquelas que não utilizavam operações pesadas de rede, uma vez que este parâmetro não foi otimizado no trabalho. Os autores obtiveram um leque vasto de aplicações e escolheram de forma aleatória seis, devido à complexidade dos estudos. Enquanto que os *code smells* identificados pelos autores do artigo [7] eram provenientes de um outro artigo [34] e já haviam sido estudados ao nível da performance, para este caso [14] as sugestões de *code smells* basearam-se na ferramenta *lint* [21, 27]. Assim, e para um conjunto de 18 APKs diferentes, com um total de 900 execuções, os autores demoraram 94 horas, em todo o processo. Para cada uma das aplicações consideradas existiram *scripts* criados com o intuito de simular a utilização da interface do utilizador.

Os resultados obtidos pelos autores encontram-se visíveis na Figura 2.13, onde se

consegue observar a poupança, em minutos, num dia de utilização.

Application	Pattern		MD	Cohen's d	IMP (%)	Savings (min)
Writeily Pro	ViewHolder	↓	-5.39	-0.78	4.50	65
	All	↓	-5.42	-0.76	4.53	65
Talalarmo	DrawAllocation	↓	-0.86	-1.11	1.47	21
	WakeLock	↓	-0.85	-1.17	1.46	21
	All	↓	-0.48	-0.57	0.82	12
GnuCash	ObsoleteLayoutParam	↓	-1.41	-0.67	0.72	10
	Recycle	↓	-1.28	-0.66	0.65	9
	All	↓	-1.53	-0.64	0.78	11
Acrylic Paint	Overdraw	↑	1.42	1.64	-2.26	-33
	All	↑	1.37	1.51	-2.18	-31
Simple Gallery	Overdraw	↑	3.08	1.04	-2.11	-30

Figura 2.13: Resultados das alteração dos *code smells* identificados [14]

De forma algo surpreendente, resolver o problema de Overdraw fez com que as aplicações consumissem mais energia. Possuir uma hierarquia de *layout* da interface do utilizador é sempre uma boa prática, no entanto adicionar código extra para evitar o Overdraw requer processamento que pode não compensar, dependendo do cenário.

Deste modo, é ainda possível perceber que os melhores resultados obtidos pelos autores foram para a versão que continha o ViewHolder corrigido e a versão com a correção de todos os *code smells* da aplicação Writeily Pro. Neste caso, para um dia de utilização os autores conseguiram provar que se pouparia 65 minutos de energia. Como se constata pela análise da figura acima, houve uma aplicação que deixou de aparecer, assim como alguns dos *code smells* identificados. Isto aconteceu por não existir um resultado significativo quanto aos testes de significância.

Para além dos resultados já apresentados para o artigo [42], os autores obtiveram ainda uma melhoria de **10%** quando corrigido o acesso ao atributo *length* do vetor. Assim, é recomendada a utilização da seguinte estrutura do código, Figura 2.14, face à implementação mais trivial, Figura 2.15. Contudo, e como apenas foi verificado para uma aplicação fictícia, não é possível tirar conclusões significativas. Assim, no nosso trabalho a ideia é tentar recriar algumas destas alterações ao código mas sempre com o foco em aplicações reais.

```
1  Object[] array;
2  int l=array.length;
3  for(int i=0;i<l;i++)
4  {
5      array[i]=null;
6  }
```

Figura 2.14: Estrutura recomendada para uso do *array.length* [42]

```
1   Object[] array;
2   for(int i=0;i<array.length;i++)
3   {
4       array[i]=null;
5   }
```

Figura 2.15: Estrutura básica para uso do *array.length* [42]

2.3 Medição de Energia no Android

Nesta secção são abordadas algumas ferramentas de medição de energia existentes. São contempladas as ferramentas tanto ao nível de hardware como ao nível de software, com o objetivo essencial de se compreender qual a melhor ferramenta a usar para obter medições o mais fidedignas possível.

2.3.1 Ferramentas de medição por hardware

Relativamente às ferramentas de hardware existentes, o *Monsoon* [46] tem sido a ferramenta mais utilizada por investigadores que optam por medições reais em vez das estimadas [42, 43, 49]. No entanto, o custo deste aparelho é muito elevado e como tal a sua utilização ficou excluída do nosso trabalho.

Um outro dispositivo de medição é o *Yocto-Amp*, utilizado no trabalho [7]. Este poderia ser utilizado devido à sua precisão (1% de erro) [69], mas como não temos um disponível e existem outras alternativas também fica de fora.

Por último, mas não menos importante, apresentamos um outro dispositivo que pode ser utilizado como ferramenta de medição, o *ODROID* [52].

Este mais sofisticado que os anteriores, na medida em que se trata de um mini computador e também já foi utilizado num dos artigos falados ao longo deste documento [14]. No entanto também apresenta as suas desvantagens, como por exemplo, não ser possível obter métricas relativamente ao GPS e ao ecrã do *smartphone*. Na Figura 2.16 é possível ter uma melhor noção dos passos a seguir aquando da utilização do dispositivo com a finalidade de obter medições energéticas. É possível obter os resultados da energia gasta pelo CPU principal, o secundário, a memória e o GPU.

2.3.2 Ferramentas de medição por software

No que diz respeito às ferramentas de software, existem algumas alternativas, mas nem todas são simples de utilizar uma vez que certos programas apenas são compatíveis com modelos específicos de *smartphones*. Além disso, por vezes, a validação das medições também acaba por ser bastante complicada [14]. Num dos artigos, também se utilizou uma destas ferramentas de medição, o PowerTutor [67]. Este permite estimar o consumo de energia em tempo real, implementado para dispositivos com o sistema Android. Esta ferramenta fornece algumas estimativas para componentes de hardware como o CPU,

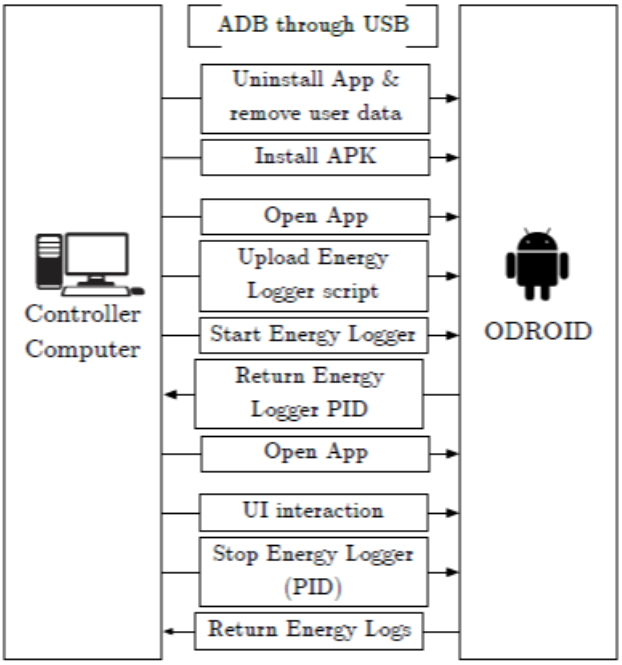

Figura 2.16: Utilização do ODROID para as medições de energia [14]

o ecrã, o GPS, o Wi-Fi, o áudio e interfaces de rede do *smartphone*. No entanto, a sua compatibilidade é muito limitada.

Apesar disso, existe uma outra ferramenta, de nome Trepn que também permite efetuar medições num conjunto de componentes, tais como, CPU, GPU, estatísticas de rede (*bluetooth*, dados móveis, *Wi-Fi*), ecrã, memória, GPS, entre outros [59] e se destaca das restantes. Esta ferramenta não só fornece um vasto leque de métricas, como também é mencionado na sua página uma lista com diversos modelos de *smartphones* para os quais o sistema proporciona as medições com precisão.

Recentemente, surgiu uma nova ferramenta *open-source* de nome PETrA [49]. Os autores realizaram testes com a mesma para 54 aplicações Android, obtendo um erro médio abaixo de 0.05J para todas as aplicações, comparativamente com o Monsoon [46]. É ainda referido que em 95% dos métodos analisados o erro estimado não ultrapassou os 5% dos valores medidos através da ferramenta de hardware, Monsoon. O PETrA é um programa instalado no computador para ambiente Linux e que permite, através de alguns *inputs* (APK da aplicação, SDK do Android), obter medições energéticas dos diversos componentes de um *smartphone*. Isto é possível, recebendo como último *input* da ferramenta um ficheiro XML designado por *power profile* e que é fornecido pelos próprios fabricantes dos *smartphones*. Infelizmente, este ficheiro nem sempre é disponibilizado pelos fabricantes.

Deste modo, neste trabalho decidimos utilizar o Trepn [59]. Complementando o que já foi dito sobre esta ferramenta, o Trepn ™ Profiler é uma aplicação Android que permite medir o desempenho e o consumo de energia de dispositivos móveis. Esta ferramenta

tem como objetivo ajudar a identificar aplicações que utilizam demasiado o CPU, consomem dados em excesso ou acabam com o tempo útil da sua bateria. Embora o Trepn Profiler possa ser executado na maioria dos dispositivos Android, recursos adicionais estão disponíveis quando este é executado em dispositivos com processadores Qualcomm® Snapdragon ™, uma vez que é um produto da Qualcomm Technologies, Inc [26].

Decidimos utilizar esta ferramenta para a medição da energia e do CPU por relatar uma precisão na ordem dos 99% [35] ou de 97.9% [36] , quando comparada com ferramentas externas de medição, através de hardware (por exemplo, Monsoon). Outro factor extremamente importante foi o facto desta estar bastante acessível a todos, através da *Google Play Store* e podermos interagir de forma simples com a mesma, integrando-a numa aplicação nossa.

Capítulo 3

Memoization para Aplicações Android

Neste capítulo apresentamos a solução por nós proposta, assim como toda a experimentação para diminuir o consumo de energia nas aplicações Android. Começamos por dar a conhecer a técnica de *memoization* por nós aplicada, na secção 3.1. Na secção 3.2, abordamos o processo do estudo experimental que conduzimos e explicamos como conseguimos obter os resultados mostrados no capítulo seguinte.

3.1 Técnica de *Memoization*

Memoization é uma técnica que permite reduzir o número de operações de uma dada função. Isto é possível armazenando o resultado de cada um dos seus *inputs* em *cache* e retornando esse mesmo valor quando a função for chamada posteriormente para um mesmo conjunto de argumentos. Os cálculos duplicados são assim evitados, sacrificando menos o CPU com a ajuda da memória [68].

A técnica de *memoization* pode ser aplicada em métodos que sigam um conjunto de requisitos [68]:

1. O método deve ser uma função pura, isto é, uma função que não têm efeitos colaterais observáveis durante a execução (também implica que o valor de retorno não pode ser `void`);
2. Os argumentos devem ser imutáveis (como os tipos primitivos do Java);
3. O valor de retorno não pode depender de campos estáticos e/ou campos de membros públicos.

Neste trabalho aplicamos esta técnica a um conjunto de aplicações Android de forma a perceber qual o impacto em termos energéticos que *memoization* pode ter. Tal como

no trabalho de Yang et al. [68] utilizámos um `HashMap` como estrutura de dados onde armazenamos os resultados (valor do mapa) de cada método para cada conjunto de argumentos (chave do mapa). Nos casos em que o método apenas tem um argumento, este é passado diretamente como chave do mapa. Se existirem dois argumentos utilizámos a classe `Pair` [28]. E no caso de existir mais do que dois argumentos, então utilizámos os tipos de tuplos fornecidos por uma biblioteca adicional (javatuples [37]), seguindo a metodologia dos trabalho de Yang et al. [68].

No nosso caso particular, relativamente ao segundo requisito, utilizámos tipos primitivos, listas e vetores de *strings* (imutáveis), e parâmetros do tipo `Context` da aplicação, uma vez que se mantêm inalterados ao longo do tempo.

Na próxima secção, descrevemos detalhadamente o estudo experimental que realizámos: numa primeira fase mostramos quais as aplicações que foram utilizadas e os métodos que foram alvo de alterações. Num segundo ponto, explicamos como ocorreram as alterações desses métodos para que passassem a utilizar a técnica de *memoization*. Finalmente clarificamos todo o procedimento para executar os testes e obter resultados de consumo energético, tempo de execução e memória utilizada.

3.2 Estudo Experimental

Foi realizado um estudo experimental baseado em três fases:

1. Extração e análise de métodos de aplicações Android
2. Modificação de métodos
3. Execução dos testes

Extração e análise de métodos de aplicações Android: O processo de procura de métodos para posterior análise e modificação precisava de ser automatizado. Para isso foi criado um *script* em python que permitiu perceber quantos métodos por aplicação é que cumpriam os pré-requisitos mencionados na secção anterior. Assim, para cada um dos repositórios e por cada ficheiro .java utilizados executou-se o *script* que verificava a existência de métodos nessas condições através de uma expressão regular que os identificava. Desta forma garantia-se que os métodos tinham retorno, não podendo ser do tipo `void` e ainda que os argumentos dos métodos eram de tipos primitivos e/ou imutáveis do java. Assim, a escolha das aplicações apresentadas daqui em diante baseou-se naquelas que maior número de métodos tinham nas condições pretendidas.

A pesquisa de aplicações Android foi realizada no repositório MUSE, uma extensão do repositório sourcerer [4] (que contém contém inúmeros repositórios Java retirados do GitHub). No entanto, focámo-nos num sub-conjunto do repositório que continha apenas aplicações Android. Segundo o critério apresentado anteriormente, selecionámos duas

aplicações: a `Pixate Freestyle` [58] e a `android-demos`. Também do F-droid foi retirada mais uma aplicação (`Chanu` [9]).

`Pixate Freestyle` [58] é uma aplicação gratuita que permite ao utilizador modelar as vistas nativas do Android com as suas folhas de estilo e é bastante baseada na componente gráfica. Esta aplicação contém 219 classes e pode ser encontrada no GitHub. Nesta aplicação fizemos *memoization* de 4 métodos conforme descrito na tabela 3.1.

`android-demos` é uma aplicação muito específica do repositório e apenas sabemos que contém 34 classes. Nesta aplicação, fizemos igualmente *memoization* de 4 métodos conforme descrito na tabela 3.1.

`Chanu` [9] é uma aplicação com 538 classes. Basicamente, é a aplicação do site 4chan onde é possível procurar imagens de vários conteúdos. Esta é a maior aplicação entre os três e, portanto, foi onde nós mais aplicámos a técnica de *memoization*. Neste caso, como é possível ver na tabela 3.1, 10 métodos foram modificados.

Com o intuito de evitar problemas de compatibilidade ao testar os métodos individualmente e para garantir que eram todos testados com a mesma configuração, juntámos todos os métodos numa aplicação criada por nós, contendo apenas os métodos referidos. Após isto, duplicámos a aplicação e numa segunda versão aplicámos a técnica de *memoization* anteriormente descrita, a todos os métodos nela contidos. A tabela 3.1 mostra um resumo dos métodos selecionados, indicando quais os tipos dos argumentos, o valor de retorno e a aplicação de onde foram extraídos.

Método	**Input**	**Output**	**Aplicação**
createIntent	Context,String,String	Intent	Chanu
replyText	long[]	String	Chanu
countLines	String	int	Chanu
planifyText	String	String	Chanu
join	List<String>,String	String	Chanu
threadSubject	String,String	String	Chanu
quoteText	String	String	Chanu
textViewFilter	String,boolean	String	Chanu
getUrl	Context,String	String	Chanu
exifText	String	String	Chanu
getNumeral	String,String	String	Pixate Freestyle
removeLocaleInfoFromFloat	String	String	Pixate Freestyle
addNegativeSign	String	String	Pixate Freestyle
addPositiveSign	String	String	Pixate Freestyle
isMobile	String	boolean	android-demos
readableFileSize	long	String	android-demos
dip2px	Context,float	int	android-demos
px2dip	Context,float	int	android-demos

Tabela 3.1: Caracterização dos métodos utilizados na experimentação.

Modificação de métodos: A modificação dos métodos ocorreu aquando da aplicação da técnica de *memoization*. Para efetuar *memoization* guardámos o *input* dos métodos como chave num `HashMap` e o seu *output* como valor. Deste modo, quando executamos o mesmo

método várias vezes para o mesmo *input* apenas temos de aceder ao seu resultado (guardado no mapa aquando da primeira execução do método). No caso de ser um novo *input* uma nova entrada é criada no mapa, uma vez que a chave ainda não existe. Nos casos em que os métodos apenas recebiam um parâmetro de entrada, este era directamente a chave do mapa. No entanto, existiram casos, como se pode constatar pela tabela 3.1, nos quais mais do que um argumento era passado como parâmetro. Para isso utilizou-se a classe `Pair` [28] no caso de existirem dois argumentos ou uma biblioteca chamada javatuples [68], quando existiram mais. Deste modo, os valores de entrada foram guardados em tuplos e isso passou a ser a chave no `HashMap`.

Execução dos testes: Por último, executámos a aplicação, mais concretamente a nossa classe de testes. Para isso, foi necessário criar mais versões da aplicação. Ou seja, para que cada um dos métodos fosse testado isoladamente, criámos 36 versões distintas (existia uma classe de testes por cada um dos 18 métodos no seu estado original e *memoized*) com o intuito de obter as medições com uma granularidade mais fina.

Cada método foi chamado 50 vezes com diferentes parâmetros (na sua maior parte eram frases ou palavras aleatórias de comprimentos variados e únicas), de modo que o comportamento dos testes fosse diversificado. Essas 50 chamadas foram realizadas ciclicamente 10, 20, 30, 40 e 50 vezes, representando assim os nossos 5 cenários de teste. No caso das versões da aplicação com *memoization*, isso permitiu a inserção de 50 valores no mapa e a consequente consulta dos mesmos há medida que várias iterações foram feitas. Na versão original isso implicou a execução dos métodos na integra, a cada chamada. Cada um dos 5 cenários existiu para perceber se a técnica por nós aplicada era estável com o número de execuções e se tinha ganhos ou perdas à medida que esse número aumentava.

Para o primeiro cenário de teste (10 execuções), existiram assim, 500 chamadas por cada um dos métodos (50×10). No caso da *memoization* isto representa 450 consultas ao mapa, uma vez que as primeiras 50 chamadas do método vão inserir 50 valores no mapa. No caso dos métodos no seu estado original, estamos perante 500 execuções na totalidade, uma vez que não se aplicou a técnica. Nos restantes cenários de teste aplica-se o mesmo raciocínio.

Com o intuito de ter um conjunto de dados fiáveis e consistentes, decidimos executar 25 vezes cada cenário de teste, tanto na sua versão original como na *memoized*. Uma vez que os métodos são de pequena dimensão, as medições variam. Assim, existirem 25 medições ajuda a ter um conjunto que no seu todo representa a tendência do método.

Para obter as medições recorremos a dois telemóveis conectados ao computador via ADB (*Android Debug Bridge*)[1] que permitiram toda a experiência: Um LG Nexus 4 com a versão Android 5.1.1 - API no nível 22 e um LG Nexus 5 com a versão Android 7.1.2 - API no nível 25 (estes telemóveis reportavam resultados fiáveis aquando da utilização do Trepn).

[1] É uma ferramenta que permite aos utilizadores aceder e/ou interagir com o seu dispositivo Android, através da linha de comandos [20].

O procedimento e o método pelo qual efetuámos os testes encontra-se resumido na Figura 3.1. Nela é possível perceber os passos seguidos até à obtenção de resultados. Assim, numa primeira fase foi necessário criar alguns *scripts* para automatizar todo o processo de testes. Desta forma, o *script* num primeiro passo desinstalava o ficheiro apk da aplicação, assim como da classe de testes para garantir que não existiam incompatibilidades. No segundo passo procedeu-se então à instalação do apk da aplicação, assim como do apk que continha os testes. Finalmente, a classe de testes foi executada (terceiro passo), as medições foram feitas e de seguida guardados os resultados (quarto passo). Isto foi possível através da integração na classe de testes do Trepn, a ferramenta de medição de energia/tempo que se encontrava instalada nos telemóveis mencionados em cima e que nos permitiu obter os resultados apresentados ao longo desta dissertação (na subsecção 3.2.1 retomaremos este tópico).

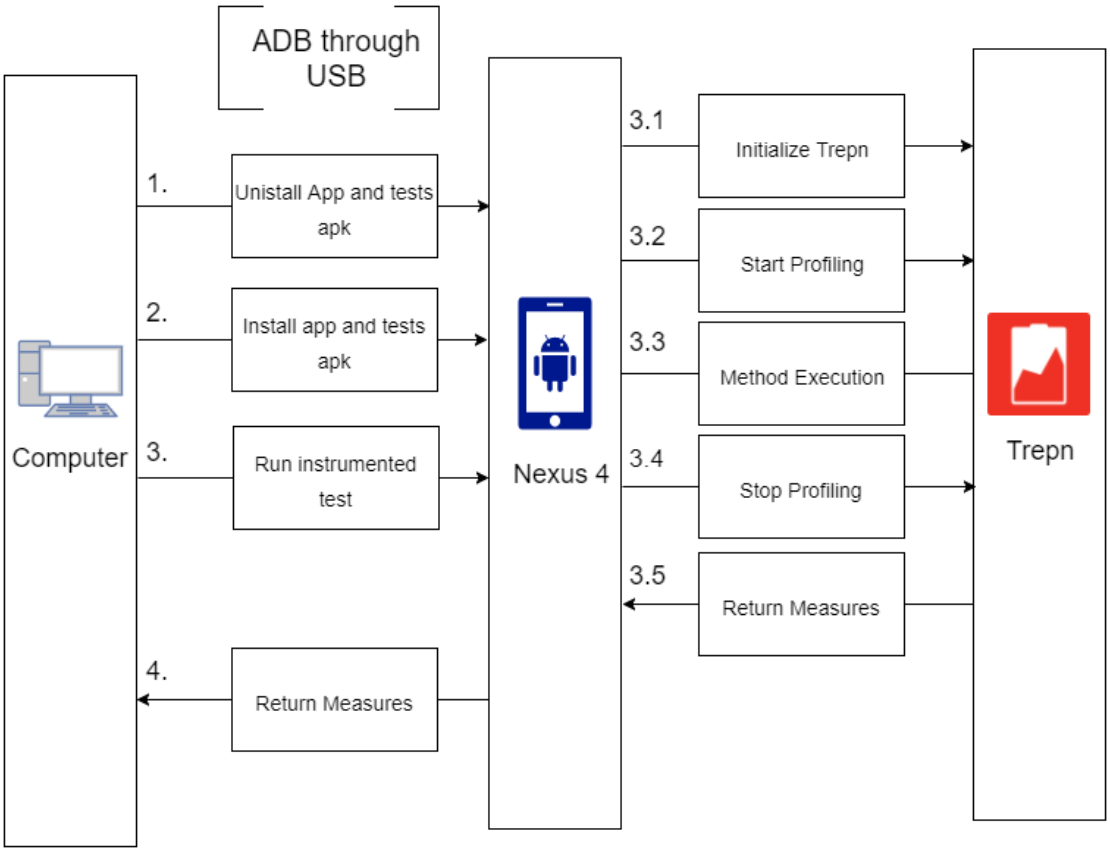

Figura 3.1: Esquema experimental para medição de energia e tempo de execução de aplicações Android.

Para a obtenção dos resultados ao nível da memória RAM consumida pelo dispositivo utilizámos uma outra ferramenta. Falaremos dela mais à frente na subsecção 3.2.2 e explicaremos como obtivemos esses resultados, uma vez que o processo diferiu do apresentado em cima para o consumo de energia e tempo.

É importante referir ainda que os testes foram executados num ambiente controlado,

ou seja, tentamos minimizar ao máximo a influência de fatores externos. Para isso, foi necessário baixar o brilho dos telemóveis para o mínimo (para garantir que a energia consumida pelo ecrã era a mínima possível), ativar o modo de voo, desligar o *bluetooth*, o GPS e desligar o Wi-Fi. Ambos os telemóveis encontravam-se ainda sem outras aplicações instaladas, excepto o Trepn para retirar as medições e a nossa aplicação (apk da aplicação e o apk que continha os testes).

3.2.1 Trepn - Utilização

Na nossa aplicação utilizámos o Trepn para fazer *profiling* da energia consumida por cada cenário de teste. Começámos por fazer um *warm-up* de 5 segundos para iniciar o Trepn e colocá-lo operacional antes de começar a tentar efetuar medições (uma vez que as chamadas ao Trepn são assíncronas). Caso não se fizesse isto, o teste podia terminar e o trepn ainda estar a iniciar, ou estarmos a tentar executar o método que dá inicio às medições `startEnergyProfiling()`, sem efetuar a inicialização e a devida preparação do trepn dentro do mesmo (listagem 3.1).

Nesta listagem de código 3.1, é possível ter uma ideia mais clara de todo o processo de integração do Trepn na nossa aplicação. O método *before()* contém o código que é executado antes das operações existentes (isso é possível graças à anotação "@Before" fornecida pela ferramenta de teste JUnit4 [38] que permite escrever testes em Android de forma mais simples) no nosso cenário de teste, para que comece a ser registado o seu gasto de energia e tempo. Quando a execução do cenário de teste terminou (método *after()* executado depois do teste com a anotação "@After"), parámos as medições e guardámos os resultados obtidos no telemóvel. No final, extraímos os resultados do telemóvel para os analisarmos e retirarmos as devidas conclusões.

Resumindo, conseguimos iniciar o Trepn a partir da nossa aplicação, colocá-lo a retirar medições e pará-lo quando assim foi necessário. Tudo de forma automática e transparente. No fim, ficheiros csv foram gerados e executámos os nossos *scripts* para obter apenas os dados que desejávamos (consumo de energia e tempo de execução).

Listagem 3.1: Integração do Trepn

```
1     @Test
2     function test10x() {
3         // test case operations
4     }
5
6     @Before
7     function before() {
8         startEnergyProfiling();
9     }
10
11    @After
12    function after() {
13        stopEnergyProfiling();
```

```
14          saveResults();
15      }
```

3.2.2 Ferramenta de medição de Memória - *Memory Monitor*

O Monitor de Memória é uma ferramenta fornecida pelo Android Studio. Esta permite uma monitorização mais simples do desempenho da aplicação e da utilização da memória. Assim, é possível encontrar objetos desalocados, localizar vazamentos de memória (*memory leaks*) e rastrear a quantidade de memória que o dispositivo conectado está a utilizar [2].

O nosso foco incidiu sobre o último ponto, uma vez que o nosso objetivo era compreender a quantidade de memória RAM consumida pela nossa aplicação no dispositivo.

Com o intuito de observar as variações de memória antes e depois da execução de um método de teste, foi colocado um *breakpoint* no código dessa classe. Isso permitiu a execução do código por passos, o que nos permitiu retirar os valores da memória consumida pelo dispositivo. Deste modo, ao analisar o gráfico que se foi formando no Monitor de Memória, conseguimos saber qual o valor de memória inicial (antes da execução do método) e final (depois da execução do método). No início e para garantir que não havia memória alocada da execução anterior, forçamos um evento por parte do *garbage collector* (esta funcionalidade é igualmente permitida no monitor de memória).

Capítulo 4

Resultados do uso de *memoization*

Neste capítulo são apresentados os resultados obtidos após a realização da experiência mencionada no capítulo anterior. Os resultados foram obtidos para o telemóvel LG Nexus 4 e posteriormente como forma de validação de toda a experiência, para o LG Nexus 5 (a maioria dos gráficos e Tabelas apresentados dizem respeito ao LG Nexus 4, a não ser que seja explicitamente dito algo em contrário). Este capítulo encontra-se dividido em três secções onde descrevemos os resultados para o consumo de energia, tempo e memória, respetivamente. Com exceção da memória, para cada uma delas, apresentamos os valores obtidos nos 5 cenários de teste (10, 20, 30, 40 e 50 execuções). Estes valores são sempre relativos a 25 execuções de cada um dos cenários de teste, tal como referido anteriormente. Para a memória, apenas queríamos perceber se a diferença entre as versões originais e com *memoization* se mantinham constante para o caso extremo de teste. Ou seja, os valores apresentados são para o primeiro (10 execuções) e quinto (50 execuções) cenário de teste.

4.1 Energia

Nesta secção apresentamos os resultados obtidos em termos energéticos. Esta secção encontra-se dividida em duas subsecções. Na primeira (4.1.1) comparamos os consumos energéticos dos métodos *memoized* e originais. Nesta subsecção os gráficos exibidos, dizem respeito ao primeiro cenário de teste (10 execuções) uma vez que os resultados já demonstram que a técnica funciona bem apenas para 10x. No entanto, para 3 métodos (`countLines`, `dip2px`, `px2dip`) mostramos todos os cenários de teste para perceber se os valores tendiam para alguma das versões com o aumento do número de execuções. E para os 2 métodos restantes (`join` e `threadSubject`) também é possível observar todos os cenários de teste para compreender se os valores não favoráveis à técnica de *memoization* se mantinham.

4.1.1 Comparação entre os consumos energéticos dos métodos *memoized* e originais

Os resultados do consumo energético, encontram-se representados sob a forma de três diagramas de caixa (*box plots*), nas Figuras 4.1, 4.2 e 4.3. Escolhemos este tipo de gráfico uma vez que permite representar sucintamente muita informação. Como foram efetuadas 25 medições, com o intuito de tentar ter um conjunto de valores representativo do real consumo. Deste modo, não quisemos reduzir essas execuções apenas para a sua média ou mediana, porque se perderia muita informação. Apresentamos os métodos em gráficos diferentes uma vez que os consumos são de ordens de grandeza diferentes, o que dificultaria a sua visualização conjunta. Estes diagramas contemplam os 13 dos 18 métodos onde foi possível encontrar uma diferença estaticamente significativa entre o consumo de energia dos métodos na sua versão original (marcados a amarelo) e na versão com *memoization* (marcados a verde). Assim, encontram-se representados os consumos onde os métodos *memoized* gastaram menos energia do que os originais.

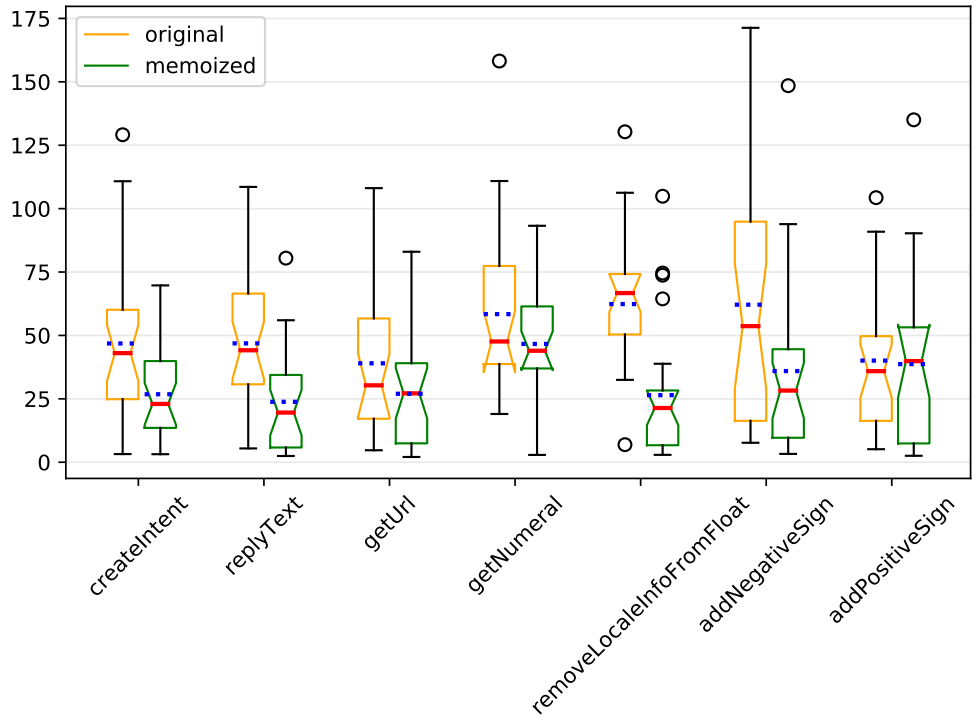

Figura 4.1: Métodos com consumo entre 0 e 175 mJ para o primeiro cenário de teste.

As medições energéticas foram obtidas a partir de 25 execuções do método em estudo e os gráficos contêm informação dos valores mínimos e máximos (representados por uma reta que se estende verticalmente denominada de *whisker*), possíveis *outliers* (representados com pequenos círculos), os primeiros e terceiros quartis (inferior e superior da caixa, respetivamente), a mediana (representada pela linha vermelha dentro da caixa) e por último a média (representada pela linha tracejada azul). Para além desta informação também se encontram representados os entalhes dos diagramas de caixa (*notched box plots*)

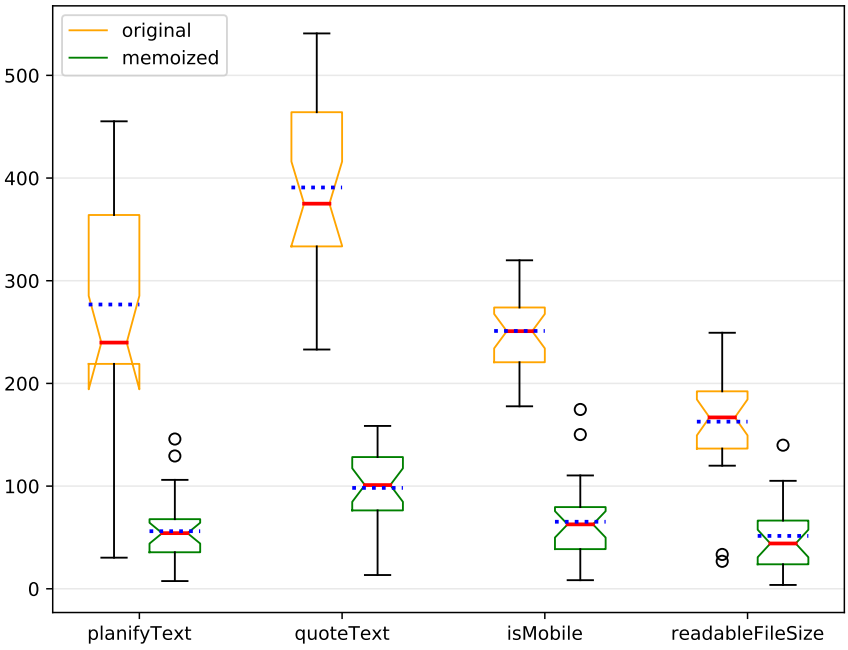

Figura 4.2: Métodos com consumo entre 0 e 600 mJ para o primeiro cenário de teste.

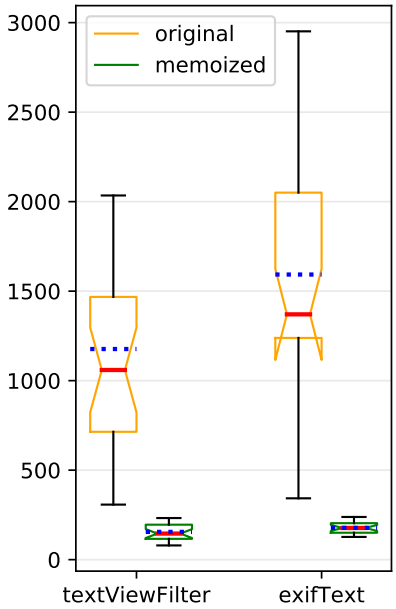

Figura 4.3: Métodos com consumo entre 0 e 3000 mJ para o primeiro cenário de teste.

que podem ser vistos como um teste informal da hipótese nula em que as medianas são iguais. Isto é, se dois entalhes se sobrepuserem então não é possível rejeitar a hipótese nula com uma confiança de 95% [8], caso contrário existe uma diferença estatisticamente significante entre as medianas.

Na Figura 4.4 mostramos os diagramas de caixa com o consumo de energia dos dois únicos métodos em que a energia gasta pelas versões *memoized* é estatisticamente superior relativamente aos métodos originais.

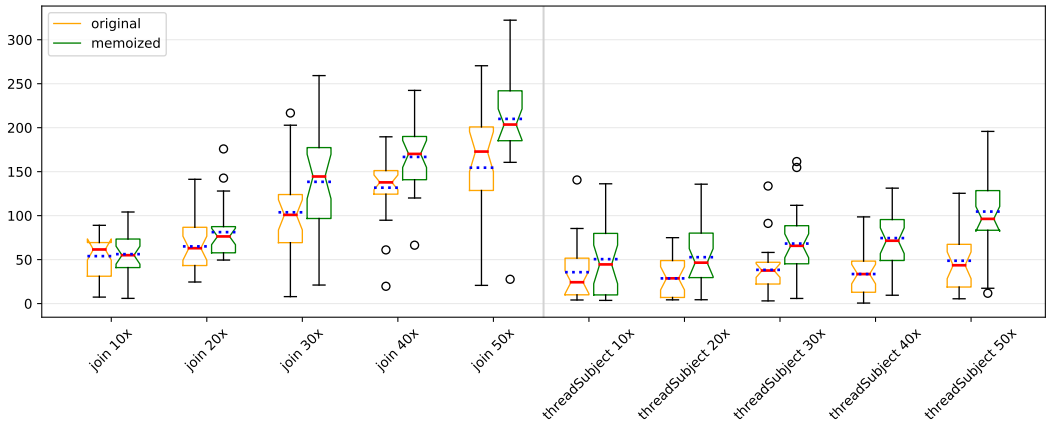

Figura 4.4: Diagramas de caixa que representam a energia gasta pelos métodos originais e *memoized*, considerando todos os cenários de teste, onde a versão original gastou menos energia.

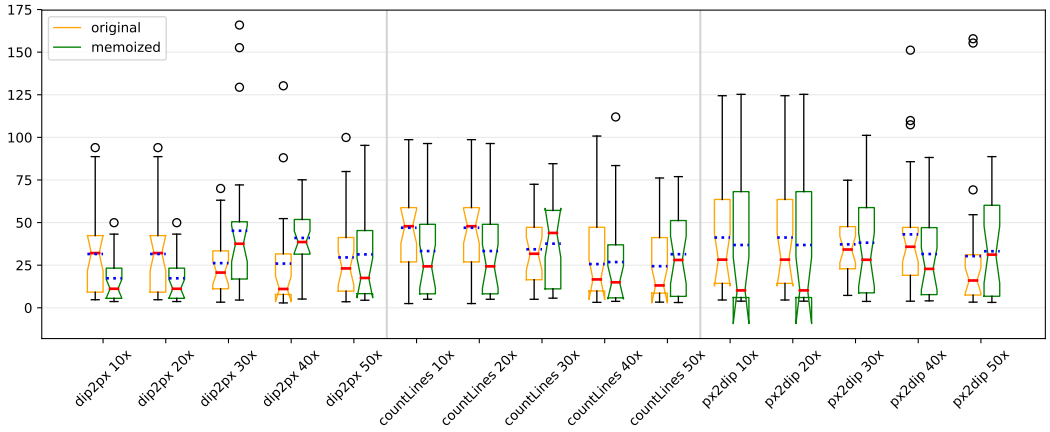

Figura 4.5: Diagramas de caixa que representam a energia gasta pelos métodos originais e *memoized*, considerando todos os cenários de teste, onde não foi encontrada nenhuma diferença estatística consistente entre as várias execuções dos testes.

A Figura 4.5 apresenta mais um diagrama de caixa, mas desta vez para os três métodos onde não foi encontrada diferença estatística entre métodos originais e *memoized*.

Para estes três métodos, executámos o mesmo conjunto de testes, mas desta vez para um maior número de vezes, ou seja, as 10 anteriores, 20, 30, 40 e ainda 50 vezes. O objetivo foi perceber se com o aumento do número de execuções passava a existir alguma tendência nos valores, isto é, se deixavam de ser imprevisíveis e começavam a tender para alguma das versões (original ou com *memoization*). O incremento de 10 execuções, em cada cenário de teste, fez com que o método *memoized* lesse mais valores armazenados no mapa, enquanto a versão original teria de executar cada método na íntegra mais vezes.

Apesar disso, a conclusão que se retira para esses métodos é que apesar de por vezes a versão *memoized* ser melhor, nem sempre isso acontece.

4.1.2 Ganhos energéticos da técnica de *memoization*

Os ganhos energéticos observados durante toda a experiência efetuada encontram-se exibidos nesta subsecção.

Na Figura 4.6, apresentamos a percentagem de perdas ao nível energético dos métodos da versão original quando comparados com os métodos *memoized*, considerando aqueles em que a *memoization* produziu resultados positivos (os métodos representados na Figura 4.1, Figura 4.2 e Figura 4.3). Assim, para cada método e para cada uma das 25 medições obtidas, calculámos a percentagem de redução energética entre os métodos sem alterações (originais) e os métodos por nós modificados (*memoized*), utilizando a seguinte fórmula $\frac{original - memoized}{original} \times 100$. Desta forma, os valores positivos representam que de facto existe um ganho no que diz respeito ao consumo de energia dos métodos *memoized* relativamente aos originais, corroborando assim a ideia da utilização da técnica de *memoization*. Os valores negativos demonstram um aumento no consumo energético, apontando situações onde a *memoization* não foi vantajosa. No entanto apenas dois métodos se encontram nesta situação (Figura 4.7). Nesta figura não mostramos valores inferiores -500 uma vez que eram poucos e não permitiam ver o gráfico em condições. Apesar dos dois métodos não produzirem resultados a favor da técnica de *memoization* e consequentemente do consumo de energia, no geral, os ganhos energéticos são bastante positivos.

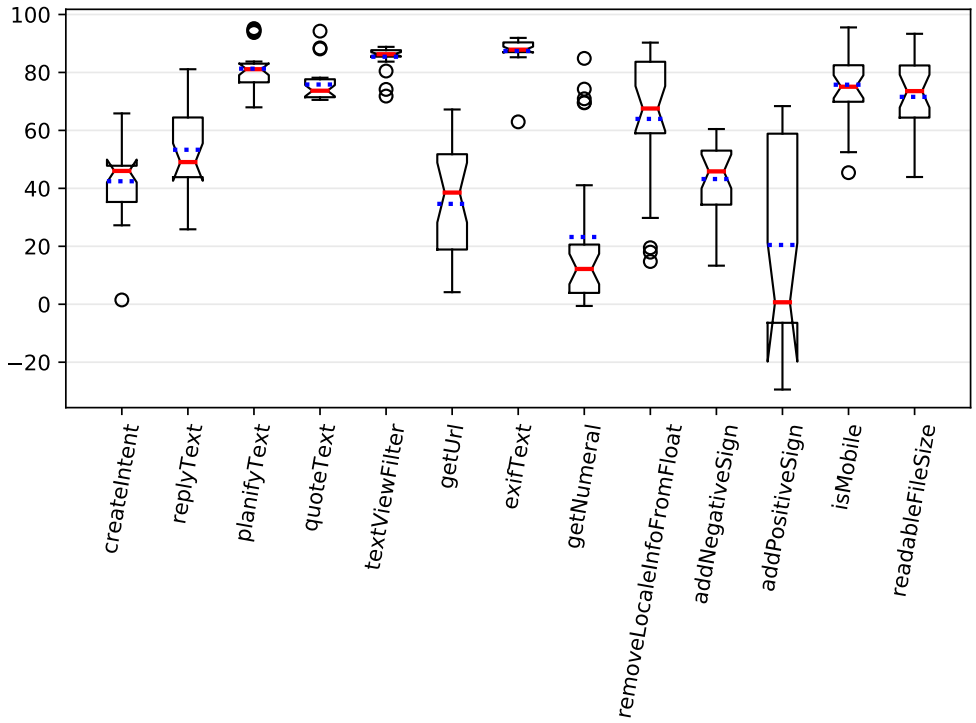

Figura 4.6: Variação do consumo de energia do método original para o *memoized* nos métodos que têm resultados positivos.

A par dos resultados anteriores, na Figura 4.8, apresentamos os valores para os três

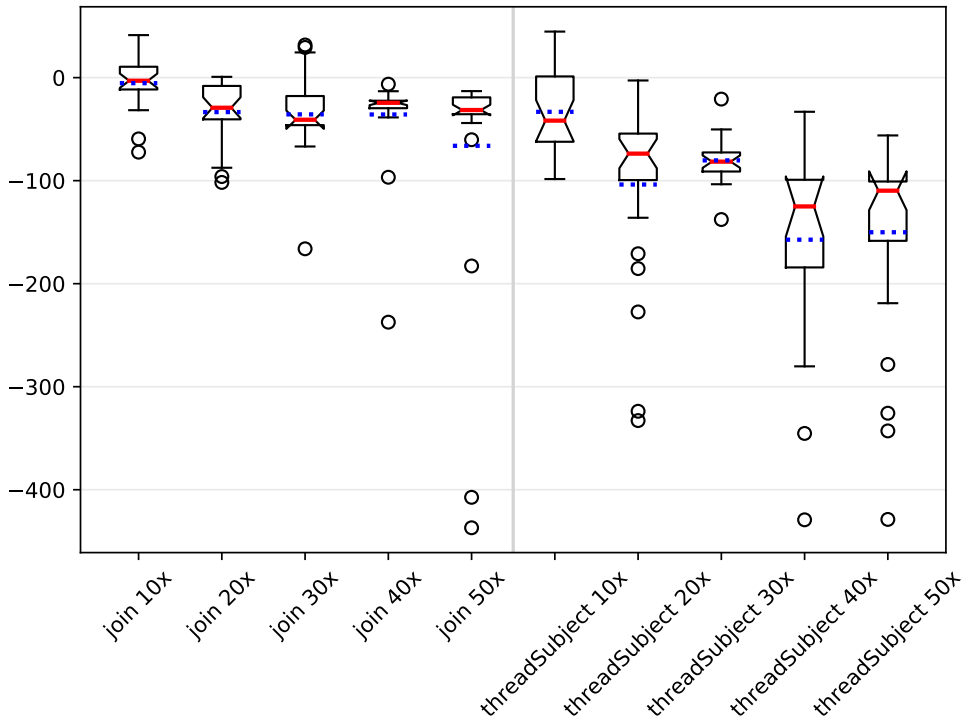

Figura 4.7: Variação do consumo de energia do método original para o *memoized* nos métodos que têm resultados negativos.

métodos onde não há indicação clara de perdas ou ganhos, considerando 10, 20, 30, 40 e 50 execuções do conjunto de teste.

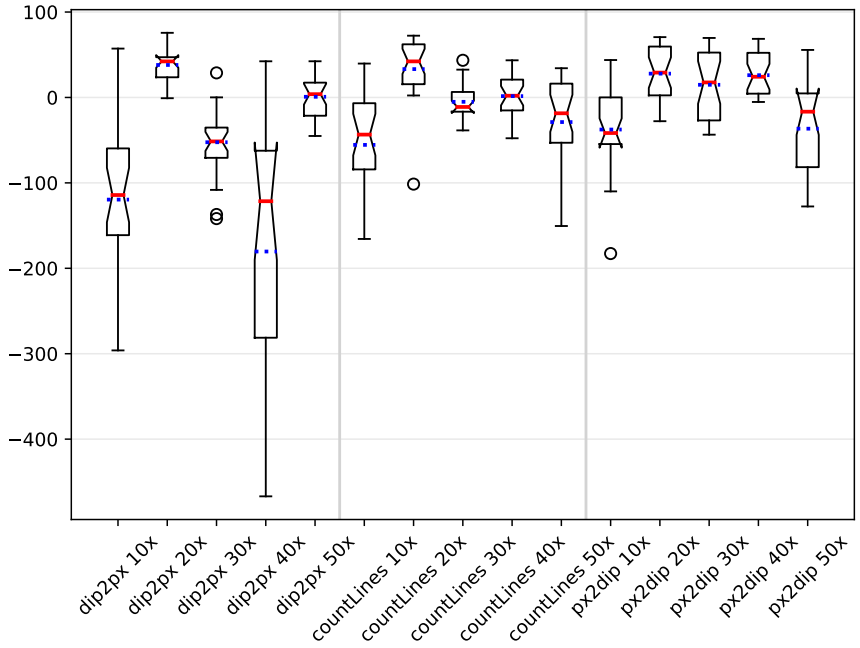

Figura 4.8: Variação do consumo de energia do método original para o *memoized* nos métodos que têm resultados imprevisíveis.

Método	10x	20x	30x	40x	50x
exifText	(87, 90)	(92, 95)			
textViewFilter	(85, 88)	(94, 94)			
planifyText	(77, 83)	(86, 89)			
quoteText	(71, 78)	(88, 92)			
isMobile	(70, 82)	(84, 96)			
readableFileSize	(64, 82)	(82, 89)			
removeLocaleInfoFromFloat	(59, 84)	(66, 71)			
replyText	(44, 64)	(-9, 17)	(41, 76)	(41, 71)	(48, 76)
createIntent	(35, 48)	(-21, 6)	(35, 54)	(21, 43)	(49, 66)
addNegativeSign	(34, 53)	(62, 80)	(61, 75)	(68, 94)	(71, 86)
getUrl	(19, 52)	(32, 66)	(45, 57)	(37, 56)	(49, 62)
getNumeral	(4, 21)	(-21, 2)	(9, 30)	(27, 45)	(60, 75)
addPositiveSign	(-6, 59)	(40, 48)	(51, 65)	(36, 57)	(10, 26)
join	(-11, 11)	(-40, -8)	(-46, -18)	(-30, -22)	(-36, -19)
px2dip	(-55, 0)	(2, 60)	(-27, 53)	(4, 52)	(-82, 5)
threadSubject	(-62, 1)	(-99, -54)	(-92, -75)	(-280, -101)	(-219, -102)
countLines	(-84, -7)	(16, 62)	(-17, 6)	(-15, 21)	(-53, 16)
dip2px	(-161, -60)	(24, 47)	(-71, -35)	(-281, -62)	(-21, 17)

Tabela 4.1: 1º (primeiro valor do par) e 3º (segundo valor do par) quartis da variação energética do uso do método original para o uso da versão *memoized*.

Para facilitar a leitura de todos estes resultados e a partir da informação proveniente dos diagramas de caixa, surge a Tabela 4.1, apresentando a percentagem da energia gasta entre o método original e o *memoized* para o primeiro e terceiro quartis. É possível constatar ainda a falta de alguns valores, para o terceiro, quarto e quinto cenários de teste. Isto justifica-se pelo facto dos 7 métodos em causa terem sido bastante a nosso favor e já não termos a necessidade de executá-los para um maior número de vezes.

Na Tabela 4.2, para o LG Nexus 4, mostramos a percentagem de vezes que a energia gasta pelo método *memoized* é menor que a original, considerando as 25 execuções de cada método, para os 5 casos de teste. Note-se ainda que os valores sublinhados são aqueles para os quais não conseguimos encontrar uma diferença de significância estatística entre as medições para a versão original e *memoized*. Para todos os outros métodos, a diferença entre o consumo de energia do método original e o método *memoized* foi notável.

A par da Tabela anterior, apresentamos também os resultados para o LG Nexus 5 para podermos comparar os valores obtidos em ambos os telemóveis. Estes encontram-se representados na Tabela 4.3.

4.2 Tempo

Nesta secção apresentamos os resultados relativos aos tempos de execução da nossa experiência. A partir desses resultados queremos perceber a sua relação com a energia e memória e de que modo varia o tempo de execução para os diferentes cenários de teste.

Os resultados da análise do tempo de execução para o LG Nexus 4 estão expressos de

	10x		**20x**		**30x**		**40x**		**50x**	
	E	T	E	T	E	T	E	T	E	T
createIntent	72	60	56	64	68	76	68	76	72	80
getUrl	60	88	72	72	68	60	76	80	76	80
replyText	68	68	52	68	84	80	92	88	84	88
getNumeral	56	68	56	68	64	88	68	96	84	96
addNegativeSign	56	72	88	92	84	96	100	100	100	100
addPositiveSign	56	72	72	84	80	96	84	88	80	92
planifyText	100	100	100	100	-	-	-	-	-	-
quoteText	100	100	100	100	-	-	-	-	-	-
textViewFilter	100	100	100	100	-	-	-	-	-	-
exifText	100	100	100	100	-	-	-	-	-	-
removeLocaleInfoFromFloat	80	88	88	100	-	-	-	-	-	-
isMobile	100	100	100	100	-	-	-	-	-	-
readableFileSize	88	100	100	100	-	-	-	-	-	-
join	48	44	32	24	28	32	28	12	24	20
threadSubject	40	20	36	4	20	8	12	8	20	8
countLines	36	36	68	44	48	44	44	48	44	40
dip2px	24	40	68	40	32	40	28	40	56	32
px2dip	40	32	56	48	48	48	64	36	44	36

Tabela 4.2: Para cada método e para cada cenário de teste, apresentamos a percentagem (das 25 execuções) em que a versão *memoized* consumiu menos energia (E) e menos tempo (T) do que a versão original, para o LG Nexus 4. Os valores sublinhados não têm significância estatística ao contrário de todos os outros.

	10x		**20x**		**30x**		**40x**		**50x**	
	E	T	E	T	E	T	E	T	E	T
createIntent	56	72	36	64	72	92	60	80	80	96
getUrl	64	72	72	84	68	88	68	84	96	100
replyText	68	80	64	48	60	60	64	84	44	92
getNumeral	100	100	100	100	100	100	100	100	100	100
addNegativeSign	80	84	64	80	68	76	52	76	80	64
addPositiveSign	52	84	60	80	68	80	44	60	60	80
planifyText	100	100	100	100	-	-	-	-	-	-
quoteText	100	100	100	100	-	-	-	-	-	-
textViewFilter	100	100	100	100	-	-	-	-	-	-
exifText	100	100	100	100	-	-	-	-	-	-
removeLocaleInfoFromFloat	88	100	100	100	-	-	-	-	-	-
isMobile	100	100	100	100	-	-	-	-	-	-
readableFileSize	88	100	100	100	-	-	-	-	-	-
join	24	28	48	32	20	12	12	28	0	4
threadSubject	20	16	16	8	8	0	8	0	4	8
countLines	40	60	60	20	64	60	28	56	52	52
dip2px	40	48	32	36	40	48	60	32	60	32
px2dip	76	60	48	32	36	36	36	36	20	36

Tabela 4.3: Para cada método e para cada cenário de teste, apresentamos a percentagem (das 25 execuções) em que a versão *memoized* consumiu menos energia (E) e menos tempo (T) do que a versão original, para o LG Nexus 5. Os valores sublinhados não têm significância estatística ao contrário de todos os outros.

forma compactada na mesma Tabela já apresentada para a energia (Tabela 4.2). Nela é possível perceber a percentagem de vezes que os métodos com *memoization* foram mais rápidos que os originais. Decidimos colocar os dados em conjunto com os da energia, com o objetivo de perceber a relação que existe entre ambos.

No que diz respeito aos resultados para o LG Nexus 5 os resultados são apresentados na Tabela 4.3. Assim, é possível compreender não só a relação existente entre energia e tempo para cada um dos telemóveis, como comparar os valores entre telemóveis diferentes.

4.3 Memória

Nesta secção apresentamos os resultados do consumo da memória do dispositivo, uma vez que aplicámos uma técnica que faz uso disso mesmo. Apresentamos assim Tabelas que nos permitem observar as diferenças obtidas com e sem *memoization*. Deste modo, queremos perceber a relação dos valores já obtidos para o tempo/energia com a memória e perceber se o uso de *memoization* implica ou não uma poupança ao nível da memória consumida pelo *smartphone*. Também queremos saber se nos casos em que as versões com *memoization* consomem mais energia do que as versões originais, o consumo de memória foi maior ou menor.

Na Tabela 4.4 apresentamos os resultados que obtivemos da medição da memória por cada método, para 10 execuções. Assim, encontram-se representados os valores de memória antes e após a execução do método de estudo. Apresentamos também a diferença entre esses valores para ambas as versões (original: orig e *memoized*: memo).

Método - 10x	MI (memo)	MF (memo)	MF-MI (memo)	MF-MI (orig)	MI (orig)	MF (orig)	orig-memo
createIntent	15.01	15.07	0.06	0.14	15.01	15.15	**0.08**
getUrl	15.01	15.07	0.06	0.2	15.01	15.21	**0.14**
replyText	15.02	15.07	0.05	0.26	15.01	15.27	**0.21**
getNumeral	15.01	15.08	0.07	0.37	15.01	15.38	**0.3**
addNegativeSign	15.01	15.06	0.05	0.18	15.01	15.19	**0.13**
addPositiveSign	15.01	15.05	0.04	0.1	15.01	15.11	**0.06**
planifyText	15.01	15.1	0.09	0.5	15.01	15.51	**0.41**
quoteText	15.01	15.82	0.81	7.62	15.01	22.63	**6.81**
textViewFilter	15.02	15.4	0.38	3.4	15.01	18.41	**3.02**
exifText	15.01	15.29	0.28	2.36	15.01	17.37	**2.08**
removeLocaleInfoFromFloat	15.02	15.07	0.05	0.3	15.01	15.31	**0.25**
isMobile	15.01	15.06	0.05	0.15	15.01	15.16	**0.1**
readableFileSize	15.01	15.07	0.06	0.27	15.01	15.28	**0.21**
join	15.01	15.13	0.12	0.74	15.01	15.75	**0.62**
threadSubject	15.01	15.19	0.18	0.05	15.01	15.06	-0.13
countLines	15.01	15.06	0.05	0.04	15.01	15.05	-0.01
dip2px	15.01	15.06	0.05	0.03	15.01	15.04	-0.02
px2dip	15.01	15.06	0.05	0.03	15.01	15.04	-0.02

Tabela 4.4: Variação da memória RAM (em MB) gasta para todos os métodos na versão *memoized* e original, em 10 execuções. Onde MI - Memória Inicial, MF - Memória Final

Na Tabela 4.5 apresentamos os mesmos resultados mas desta vez para 50 execuções. Decidimos então experimentar com o mínimo e com o máximo número de execuções definido para o nosso conjunto de testes de forma a perceber e corroborar a avaliação dos

nossos dados. Assim, poderíamos perceber se a diferença entre as versões originais e com *memoization* para os diferentes métodos se mantinham constantes.

Os valores apresentados nestas tabelas serão explicados e analisados no próximo capítulo de análise de resultados.

Método - 50x	MI (memo)	MF (memo)	MF-MI (memo)	MF-MI (orig)	MI (orig)	MF (orig)	orig-memo
createIntent	15.01	15.1	0.09	0.58	15.01	15.59	**0.49**
getUrl	15.01	15.07	0.06	0.85	15.01	15.86	**0.79**
replyText	15.02	15.08	0.06	1.13	15.01	16.14	**1.07**
getNumeral	15.01	15.12	0.11	1.69	15.01	16.7	**1.58**
addNegativeSign	15.02	15.06	0.04	0.76	15.01	15.77	**0.72**
addPositiveSign	15.02	15.05	0.03	0.37	15.01	15.38	**0.34**
planifyText	15.01	15.1	0.09	2.28	15.01	17.29	**2.19**
quoteText	15.01	15.82	0.81	8.07	15.01	23.08	**7.26**
textViewFilter	15.01	15.43	0.42	9.12	15.01	24.13	**8.7**
exifText	15.01	15.29	0.28	9.4	15.01	24.41	**9.12**
removeLocaleInfoFromFloat	15.01	15.07	0.06	1.39	15.01	16.4	**1.33**
isMobile	15.01	15.06	0.05	0.61	15.01	15.62	**0.56**
readableFileSize	15.01	15.12	0.11	1.23	15.01	16.24	**1.12**
join	15.01	15.16	0.15	3.55	15.01	18.56	**3.4**
threadSubject	15.01	15.7	0.69	0.05	15.01	15.06	-0.64
countLines	15.01	15.06	0.05	0.04	15.01	15.05	-0.01
dip2px	15.01	15.12	0.11	0.03	15.01	15.04	-0.08
px2dip	15.01	15.12	0.11	0.03	15.01	15.04	-0.08

Tabela 4.5: Variação da memória RAM (em MB) gasta para todos os métodos na versão *memoized* e original, em 50 execuções. Onde MI - Memória Inicial e MF - Memória Final

Capítulo 5

Análise de resultados

Neste capítulo são analisados os resultados apresentados no capítulo anterior. Este capítulo encontra-se dividido em 4 secções. Na secção 5.1 analisamos os resultados para o consumo de energia, na secção 5.2 para o tempo de execução e na secção 5.3 memória consumida. Por último, na secção 5.4 apresentamos alguns dos obstáculos que foram surgindo aquando da validação dos resultados por nós obtidos.

5.1 Energia

Para os 18 métodos estudados e analisados é possível apontar algumas observações interessantes, discutindo-as em termos energéticos ao longo desta secção.

A primeira observação é a mais óbvia: a *memoization* tem um claro impacto no consumo de energia em aplicações Android. Na verdade, para a maioria das situações o impacto é bastante positivo. Considerando os 18 métodos testados, o consumo de energia para 13 deles diminuiu consistentemente ao usar *memoization* (como pode ser visto nas Figuras 4.1, 4.2 e 4.3). Para esses métodos, podemos observar que o consumo médio das 25 medições é sempre menor ao usar a *memoization*, bem como os valores mínimo e máximo, e os valores para o primeiro e terceiro quartil. As Figuras 4.2 e 4.3 mostram os métodos que de todos tiveram o maior impacto. Desta forma, é possível afirmar que de facto a técnica de *memoization* pode ser utilizada para reduzir o consumo de energia em aplicações Android, respondendo assim à questão **RQ1**.

Para tornar todo o processo de análise dos resultados mais significativo, queremos saber se há ou não evidências estatísticas que sustentem estas observações, ou seja, se a energia consumida pela versão *memoized* desses métodos é consistentemente menor que a original. Por outras palavras, queremos validar se os valores obtidos para a nossa experiência são estatisticamente significativos. Assim, testámos a seguinte hipótese:

- $H_0 : P(A > B) = 0.5$
- $H_1 : P(A > B) \neq 0.5$

Em que A e B representam o acto de obter aleatoriamente um valor a partir do conjunto de 25 medições, sem utilizar *memoization* e utilizando, respectivamente. Portanto, $P(A > B)$ representa a probabilidade de obter um valor de A (sem *memoization*) maior do que o retirado de B (usando *memoization*). Assim, a nossa hipótese nula é então obter uma probabilidade de 50%, enquanto a hipótese alternativa passa por obter uma probabilidade diferente de 50%. Para entender se existe uma relevância significativa global entre as distribuições de A e B, executámos o teste de Wilcoxon (Wilcoxon signed-rank test), com o *two-tail-p-value* considerando $\alpha = 0.01$. O teste foi repetido para todos os 13 métodos onde o uso da *memoization* levou a melhorias no consumo de energia. No final, o teste produziu uma relevância significativa, obtendo o p-value < 0.01 para 11 dos 13 casos. As excepções ocorreram com os métodos `getUrl` e `addPositiveSign`. Para obter um tamanho de efeito não paramétrico, Field [19] sugere utilizar a fórmula de Rosenthal [60, 61] para calcular uma correlação e comparar os seus valores com os limites sugeridos por Cohen [10]:

- 0.1: pequena significância;
- 0.3: significância média;
- 0.5: grande significância

Dos 11 cenários, 8 não eram paramétricos e os valores obtidos foram: 0.4 (médio) para os métodos `createIntent` e `replyText` e 0.6 (grande) para os restantes. Para os 3 métodos em falta, um teste de D'Agostino e Pearson [16] revelou que estávamos perante distribuições normais e calculámos assim o coeficiente *Cohen's d* para determinar a magnitude do *effect size*. De acordo com Sawilowsky [64], os limiares (thresholds) de referência e o seu tamanho de efeito respetivo deve ser:

- 0.01: muito pequeno;
- 0.2: pequeno;
- 0.5: médio;
- 0.8: grande;
- 1.2: muito grande;
- 2: enorme significância.

Obtivemos valores de 0.4 (pequenos) para o método `getNumeral`, 0.6 (médio) para `addNegativeSign` e 1.4 (muito grande) para o método `removeLocaleInfoFromFloat`. Considerando esses valores de referência, temos o suporte estatístico suficiente para dizer que, para esses métodos, o uso de *memoization* leva a uma poupança de energia.

Os valores de tamanho do efeito calculados até agora foram apenas para o cenário em que cada conjunto de testes foi executado 10 vezes. Ainda pode ser possível que para métodos com valores de significância mais baixos, repetir a experiência com um número crescente de invocações levaria a resultados mais ou menos favoráveis. Assim sendo, repetimos a experiência com 20, 30, 40 e 50 invocações, calculando os valores de significância e o tamanho do efeito (Tabela 4.2). É possível observar que todos os 13 métodos acima mencionados continuam a ter mais de 50% dos 25 testes a favor da *memoization*. Embora em alguns casos específicos (método `getUrl` e `addPositiveSign`), a experimentação não tenha resultado em qualquer significância estatística, a percentagem de testes favoráveis à *memoization* continuou a aumentar. Como tal, classificámos os 13 métodos como propícios a *memoization*.

Em alguns casos, mais concretamente em 3 dos métodos, não é possível determinar se a *memoization* é adequada para economizar energia ou não (Figura 4.5). Ao executar a mesma experiência estatística realizada para os 13 métodos, onde foi possível perceber que a *memoization* diminuiu o consumo de energia, observámos que os resultados para os métodos `dip2px`, `px2dip` e `countLines` não foram estatisticamente significativos. Isto significa que o motivo por trás do consumo de energia ser menor não está diretamente relacionado com o uso da *memoization*, nem com o uso da versão original do método. Da mesma forma, a percentagem de testes a favor da *memoization* foi na sua maioria cerca de 50%, diminuindo ou aumentando alguns níveis percentuais de forma imprevisível ao aumentar o número de invocações do conjunto de testes. Deste modo, classificámos estes métodos como sendo imprevisíveis e inconclusivos.

5.1.1 Ganhos energéticos da técnica de *memoization*

É possível observar os dados que demonstram os ganhos calculados (valores positivos) ou perdas (valores negativos) ao usar *memoization*, na Tabela 4.1. Para obter esses valores, primeiro ordenámos, para cada método, a energia consumida pela versão original e *memoized*, para comparar os valores de consumo de energia mais baixos/mais altos de uma versão com os valores mais baixos/mais altos da outra. Assim, o que fizemos foi calcular os ganhos dois a dois. O primeiro elemento do par em cada célula da Tabela é o ganho calculado para o primeiro quartil dos valores, enquanto o segundo elemento é o ganho considerando o terceiro quartil. Com isso, tentamos demonstrar que os ganhos/perdas são independentes do valor de medição do consumo de energia. De facto, em alguns casos, o consumo de energia tende a ser baixo (mais próximo do 1º quartil), enquanto outros tendem a ser altos (mais próximo do 3º quartil). Em qualquer um destes casos, os ganhos/-perdas são aproximados. Em cada coluna estão ainda representados os ganhos/perdas

calculados para os resultados obtidos de executar a mesma experimentação, mas variando o número de vezes de execução do conjunto de testes de 10x para 50x.

Como esperado, vemos que a maioria dos valores são positivos, isto é, a *memoization* na grande maioria dos casos, é uma técnica adequada para economizar energia. Nos primeiros 7 métodos, o impacto é mais notável, já que em todas as 25 medições a versão *memoized* tem um consumo de energia significativamente menor. Ao passo que quando repetimos a experiência para 20 invocações por teste, o impacto é ainda maior. Além disso, os valores de significância continuaram a aumentar e dessa forma decidimos parar com as medições, categorizando os métodos como fortemente propícios à *memoization*.

Se olharmos apenas para os pares com ganhos positivos na coluna para 10 execuções (métodos propícios a *memoization*), os mesmos podem variar de 3% a 90%. Esses valores tendem a aumentar se aumentarmos o número de invocações: para esses mesmos métodos, considerando 20 execuções e se excluirmos aqueles com valores negativos, já que foram os casos sem significância estatística (ver Tabela 4.2), passam de 31 % para 96%. Para os outros cenários (30, 40 ou 50 execuções), os valores são sempre positivos, uma vez que a significância se mantém.

Para os métodos imprevisíveis, os ganhos são também, como se poderia esperar, imprevisíveis: no 3º quartil o valor é negativo (perda), mas para o 1º quartil é positivo (ganho). Há medida que o número de execuções de teste aumenta, os métodos vão sendo ainda mais imprevisíveis: o método `px2dip`, por exemplo, tem um ganho/perda negativo no 1º quartil para os cenários de 10, 30 e 50 execuções, enquanto os outros valores são positivos. O método `countLines` tem um comportamento semelhante, mas os valores não são proporcionais. Já o método `dip2px` passa de uma perda de 161% no 1º quartil em 10 execuções para 24% de ganho em 20 execuções. Para 30 e 40 execuções volta a perder, desta vez, 71% e 281%, respetivamente, podendo-se verificar o mesmo comportamento para o 3º quartil. Além disso, ao observar a Figura 4.5, podemos ver que para esses métodos, o intervalo interquartil do diagrama de caixa para a versão *memoized* por vezes está acima e outras vezes chega mesmo a sobrepor a versão original, o que corrobora a análise anterior.

Da mesma forma, os valores para os métodos considerados impróprios à técnica de *memoization* também foram os esperados. Existem perdas em todos os cenários testados, tanto para o 1º quanto para o 3º quartil, excepto para o primeiro cenário (10 execuções), onde o 3º quartil tem ganhos pequenos de 11% e 1% para o método `join` e `threadSubject`, respectivamente. Mesmo assim, nesse cenário, o teste estatístico mostrou que não houve significância para ambos os métodos. À medida que aumentámos o número de execuções, os valores para os ganhos no 1º e 3º quartil permaneceram negativos e tenderam a diminuir proporcionalmente.

5.1.2 Validação de resultados

Como se pode constatar, a classificação dos métodos manteve-se inalterada aquando da utilização de um outro telemóvel na experimentação (Tabela 4.3). No entanto, os

resultados não foram exatamente os mesmos e é importante destacar alguns factos.

Por exemplo, para o método `getNumeral`, apesar dos resultados terem sido a favor da técnica de *memoization* no LG Nexus 4, para o LG Nexus 5 esses resultados foram muito mais acentuados. Ou seja, em todos os casos de teste a versão *memoized* consumiu menos energia do que a versão original para as 25 execuções, isto é, em 100% das vezes. Para este método em particular decidimos executá-lo para todos os casos de teste para percebermos se os resultados se mantinham (apesar de se ter obtido 100% para o primeiro cenário de teste, continuámos a executar uma vez que o mesmo não tinha acontecido para o outro telemóvel), algo que se veio a confirmar.

Um facto curioso é que pela primeira vez existiu um método impróprio à técnica de *memoization* em 100% das vezes. O método foi o `join` e tal situação verificou-se para as 50 execuções (à medida que o número de execuções aumentava, este método piorava em relação à utilização da técnica). É possivel constatar ainda que o outro método igualmente impróprio (`threadSubject`) também foi piorando com o incremento do número de execuções, apesar do seu valor nunca ter chegado a 0.

Relativamente aos métodos que já tinham tido resultados muito favoráveis à aplicação da técnica, é importante destacar que o mesmo se sucedeu para o LG Nexus 5. Os métodos `planifyText`, `quoteText`, `textViewFilter`, `exifText`, `removeLocalInfoFromFloat`, `isMobile`, `readableFileSize` mantiveram-se assim bastante consistentes.

A categorização dos métodos por nós sugerida, como propícios/impróprios e imprevisíveis à aplicação da técnica de *memoization*, ajuda-nos a começar a perceber quando é que vale a pena utilizar a técnica de *memoization* para poupar energia. Embora essas classificações sejam baseadas numa abordagem dinâmica (precisámos de executar várias experiências primeiro e fazer uma análise aprofundada à posteriori para chegar a essas conclusões), acreditamos que as observações aqui discutidas podem ser usadas para realizar uma análise mais estática, analisando as semelhanças dos métodos em cada categoria.

5.2 Tempo

Os resultados apresentados para o tempo mostram 13 métodos onde a técnica de *memoization* se superiorizou poupando tempo de execução. Conseguimos assim responder à questão **RQ2** em que realmente a *memoization* contribui para a diminuição do tempo de execução na maioria dos casos. Ou seja, a par da análise energética, estes dados vêm corroborar que as alterações efetuadas nos métodos das aplicações têm ganhos, na sua maior parte.

Incidindo o nosso foco sobre as primeiras 10 execuções, na Tabela 4.2, percebemos que 10 execuções era o suficiente para que esses 13 métodos tanto consumissem menos tempo, como menos energia quando comparados com a sua versão original. Algo que poderia ser expectável é que se para 10 execuções já obtivemos resultados convincentes, então para 50 execuções só iríamos acentuar ainda mais esses resultados. Na verdade isso foi o que aconteceu, exceto para o método `getUrl`. Isto é, apesar deste ter consumido

sempre menos tempo na versão *memoized*, o resultado foi melhor para as 10 execuções do que para as 50. O melhor exemplo de um caso que melhorou com o aumento do número de iterações é o método `addNegativeSign` onde o incremento de 10 execuções se fez notar bastante no seu desempenho. Assim, este foi sendo cada mais rápido para a versão *memoized*, chegando mesmo a sê-lo em 100% das vezes. Isto indica que em 25 valores retirados para o tempo, os 25 eram menores na versão *memoized*, quando comparada com a original. No entanto e como já foi referido anteriormente, houve um método (`getUrl`) que apesar de ter melhorado o seu consumo energético a cada 10 execuções não o fez para o tempo, piorando até um pouco. Desta forma, nem sempre gastar menos tempo significa gastar menos energia [5, 55].

Se repararmos, para os 13 métodos considerados propícios à técnica de *memoization*, apenas em 4 dos 44 valores, o tempo de execução foi inferior aos valores do consumo de energia. Isto indica que quando poupamos energia, acabamos por reduzir também o tempo de execução.

O mesmo se passa para os métodos com resultados menos positivos (restantes 5 métodos). Enquanto que no consumo de energia, os métodos foram divididos em 3 grupos porque os resultados assim o indicavam, neste caso, o grupo em que os resultados foram imprevisíveis convergiu para o grupo dos resultados impróprios a *memoization*. O que se pode perceber dos dados observados é que em 25 valores, 18 deles tiveram um tempo de execução inferior à energia consumida. Quer isto dizer que para os métodos que são piores em termos de consumo energético, são ainda piores relativamente ao tempo de execução. Os valores são assim mais acentuados e as diferenças mais visíveis. Por exemplo, no pior método observado (`threadSubject`), no mínimo a nossa solução era melhor apenas em 12% das vezes para o consumo de energia. Agora, em termos de tempo de execução a nossa solução apenas foi melhor em 4% das vezes. Isto representa um único valor melhor em 25 (uma vez que a nossa base de estudo são sempre as 25 medições realizadas).

Relativamente aos dados para o LG Nexus 5 (Tabela 4.3), comparativamente com o LG Nexus 4, as diferenças são muito semelhantes com as já observadas e discutidas na secção da energia. Assim, o método que mais se destacou pela sua diferença foi o `getNumeral`. Tanto a nível de energia como de tempo passou a ser favorável a *memoization* em 100% dos casos. Assim, mais uma vez, conseguimos perceber que tirando casos bastante particulares os resultados se mantiveram consistentes e na sua maior parte propícios à técnica de *memoization*.

Como já foi referido, apesar dos resultados para os tempo de execução dos casos de teste no LG Nexus 4 serem em muito semelhantes aos tempos obtidos posteriormente para o LG Nexus 5, a divisão dos métodos por categorias não foi a mesma. Para o caso do LG Nexus 5 os métodos mantiveram a classificação da energia (métodos propícios, impróprios e imprevisíveis), enquanto que no LG Nexus 4 isso não aconteceu (os métodos imprevisíveis deixaram de existir).

Deste modo, podemos concluir que o tempo e a energia estão muito relacionados (esta

era já uma intuição que acabou por ser confirmada). No entanto, a energia não se comporta de forma exatamente igual ao tempo de execução. Observámos ainda que quando é favorável para a energia, é ainda mais favorável para o tempo. Também foi possível perceber o oposto, em que quando não é favorável energeticamente, então também não o é de todo temporalmente.

5.3 Memória

Quanto aos resultados apresentados para a memória, queríamos perceber se a diferença entre as versões originais e com *memoization* se mantinham constantes para o caso extremo de teste. Assim sendo, os valores demonstrados no capítulo anterior e discutidos nesta secção dizem respeito ao 1º (10 execuções) e 5º (50 execuções) cenários de teste.

A classificação dos métodos seguiu a lógica da apresentada para os tempos de execução e energia. No entanto, neste caso os valores para os métodos propícios a *memoization* encontram-se a negrito, para os métodos impróprios à técnica de *memoization* estão sublinhados e os restantes foram considerados métodos imprevisíveis.

Na análise dos resultados é importante concentrarmo-nos em dois pontos: 1) no consumo de memória inicial (obtido à priori, antes de qualquer execução dos métodos da aplicação) e 2) no consumo de memória final (obtido à posteriori, após a execução dos métodos).

Assim, observando as Tabelas 4.4 e 4.5 para ambas as execuções (10 e 50x, respectivamente), é percetível que o valor inicial se mantém praticamente o mesmo em todas as medições, para ambas as versões da aplicação. Isto justifica-se com o facto de não haver diferenças neste ponto entre a versão *memoized* e a original. Após a execução de cada método é que as diferenças começam a ser notáveis.

Desta forma, é possível ver que a técnica por nós aplicada compensa na grande maioria dos casos. Respondemos assim à última questão, **RQ3**, em que a *memoization* diminui a memória gasta pelo *smartphone*. Se olharmos para a coluna que tem a diferença entre as memórias (final e inicial de cada versão), é possível ver que o valor rege a nosso favor para 14 métodos, sendo que para 3 deles é inconclusivo e apenas para 1 é negativo. Como é possível constatar, houve mais 1 método a ter resultados positivos relativamente aos obtidos para a energia e tempos de execução (tinham sido 13 métodos). O método `join` tinha sido classificado como "impróprio" e no entanto passou a ser um dos métodos a favor de *memoization*.

Os resultados apresentados para 50 execuções comprovam mais uma vez o que foi dito anteriormente, os resultados para os 14 métodos mantiveram-se bastante favoráveis e com uma diferença significativa para os originais. Um dado interessante é que aumentando o número de execuções, os valores da memória para a versão original da aplicação aumentaram igualmente. No entanto, para a versão de *memoization* os valores apesar de terem aumentado, este aumento foi muito mais subtil. O método que era dado como

negativo, continuou a sê-lo mas apesar disso os métodos `dip2px` e `px2dip` passaram de inconclusivos para impróprios.

Estes são resultados bastante consistentes na medida que vão ao encontro dos resultados obtidos previamente para a energia e para o tempo, algo que nem sempre aconteceu noutros estudos [54]. Com isto, o único método que difere um pouco é o `join` que teve resultados positivos na memória mas tinha sido dos piores em termos de energia e tempo.

A grande maioria dos resultados obtidos é compreensível e faz sentido que a nossa solução seja melhor. Constatamos que no caso em que aplicamos a técnica de *memoization* estamos a gastar memória ao criar o nosso mapa, assim como ao inserir novos valores no mesmo. No entanto, apenas na primeira iteração estamos a criar novos valores. A partir daí iremos sempre estar a consultar os mesmos e nunca a inserir mais. Ou seja, tanto para as 10x como para as 50x inserimos 50 valores no mapa (fazemos 50 chamadas ao mesmo método com *inputs* diferentes, como explicado na secção 3.2). Deste modo, apesar de na versão original não termos o custo extra de criar um mapa de tamanho 50, temos o custo acrescido de executar sempre o método cada vez que queremos obter um *output*. Assim, estamos constantemente a alocar memória e para métodos que façam algo mais trabalhoso isso vai pesar, como já foi visto anteriormente.

Sucintamente, ao passo que na versão original estamos a executar o mesmo método na íntegra 500x, para as 10 execuções e 2500x para as 50 execuções, na versão em que fazemos *memoization* nunca chegamos a executar o método tantas vezes, na medida em que o resultado deste se encontra alocado no hashmap e apenas o temos de consultar. Isto justifica a melhor performance em termos de memória na grande maioria dos casos para a metodologia por nós seguida.

5.4 Obstáculos à validação

Nesta secção, abordamos possíveis problemas que podem influenciar as descobertas que relatamos ao longo deste trabalho. Detalhamos ainda como minimizámos esses problemas para tornar os resultados confiáveis e generalizáveis.

Medir o consumo de energia de um *smartphone* é uma tarefa complexa [5]. Isso deve-se principalmente ao facto de que é muito difícil isolar totalmente o código ou a aplicação que está a ser estudada. Para atenuar esse problema, decidimos executar a nossa aplicação 25 vezes por cada um dos testes elaborados. Desta forma, existe margem suficiente para obter, em média, resultados que realmente correspondam à verdade e com os menores efeitos colaterais possíveis.

Além disso, executámos a aplicação em dois *smartphones* Android completamente limpos, ou seja, sem qualquer outra aplicação instalada exceto as aplicações do próprio sistema operativo e o Trepn, para obter as medições. Garantimos ainda que todos os possíveis serviços em segundo plano estavam desativados, colocámos os *smartphones* no modo voo, e com o menor nível possível de brilho, para que a energia consumida pelo ecrã

fosse a mais baixa possível. Ao utilizar dois *smartphones* diferentes conseguimos validar os resultados obtidos para duas versões distintas do Android.

Com o intuito de obter dados consistentes, as várias versões da aplicação *memoized* e não-*memoized* foram executadas no mesmo ambiente, metade das vezes iniciando as medições com a versão *memoized* e a outra metade com a não-*memoized*. Assim, isso dá-nos confiança de que as diferenças encontradas entre as duas versões se devem apenas ao uso de *memoization* ou não. De facto, os valores absolutos de energia em si não são o que mais importa. O que é verdadeiramente importante é que as medições são consistentemente diferentes, sendo os resultados estaticamente significativos e a favor da versão *memoized*.

A aplicação testada foi criada por nós para executar os 18 métodos obtidos de 3 aplicações Android reais nos quais aplicámos a técnica de *memoization*. Não executámos cada uma dessas 3 aplicações individualmente porque quisemos isolar os métodos onde realmente se podia utilizar *memoization* e medir o seu consumo de energia. As aplicações foram escolhidas sem nenhum critério específico, ordenando-as apenas pelo número de métodos onde seria possível aplicar *memoization*. A nossa aplicação apenas contém o código dos métodos copiados das outras aplicações, os métodos por nós alterados e a classe de testes que criámos.

Finalmente, outro possível problema é a qualidade dos testes executados. Geralmente, é muito difícil encontrar testes para aplicações Android [11, 12, 14]. Assim, criámos um conjunto de testes para cada método que queríamos analisar. Embora não sejam usos reais dos métodos, acreditamos que os testes elaborados podem representar um uso razoável de métodos *memoized*. Os testes foram escritos de forma pseudo-aleatória, ou seja, na grande maioria dos casos utilizámos frases/palavras aleatórias como *input* de cada um dos métodos estudados.

Em última análise, somente o programador pode ter alguma ideia sobre o uso real do método e, portanto, saber se faz sentido ou não que este seja *memoized*. De qualquer forma, mostramos que, nas circunstâncias avaliadas, a técnica de *memoization* por nós aplicada é benéfica para poupar energia, diminuindo o tempo de execução e reduzindo o consumo de memória.

Capítulo 6

Conclusões e trabalho futuro

Nesta dissertação, explorámos a utilização de *memoization* em diferentes aplicações Android. Efetuámos um estudo que não só incidiu sobre o impacto no consumo de energia, como também no tempo e na memória gasta.

Foram selecionados 18 métodos de 3 aplicações diferentes e elaborámos uma experiência para avaliar as 3 métricas acima referidas. Para isso tivemos em conta a versão dos métodos por nós alterados aquando da aplicação da técnica de *memoization*, assim como a versão sem qualquer alteração (original). Os resultados dessa experiência comprovaram que, de facto, o uso de *memoization* promove poupanças de energia significativas na maioria dos casos, assim como de tempo e de memória.

No que diz respeito ao consumo de energia, foi possível identificar 13 métodos onde isso aconteceu. Apenas 2 métodos tiveram o comportamento oposto e, para 3, as conclusões foram imprevisíveis, ou seja, não se verificou uma tendência para nenhuma das implementações. Estes resultados ocorreram para dois telemóveis distintos (LG Nexus 4 e LG Nexus 5), o que permitiu validar os resultados da experiência realizada.

Por definição, a energia está relacionada ao tempo de execução, tendo em conta a equação de energia: $E = P \times T$, onde o P representa a potência e T o tempo. De facto, em geral, tanto a energia quanto o tempo diminuem ou aumentam em conjunto. O nosso estudo comprovou isso mesmo e o tempo revelou-se também menor para os mesmos 13 métodos. No entanto e como os resultados foram mais acentuados, os 3 métodos que tinham sido imprevisíveis passaram a ser melhores para a versão original. Assim, existiram 13 métodos a favor da técnica por nós aplicada e 5 métodos onde isso não se verificou.

Por último, a memória revelou-se a favor da *memoization* não só nos 13 métodos já mencionados, como também, em mais um método (join). Neste, a técnica de *memoization* foi benéfica para o consumo de memória, mas apesar disso gastou mais energia e

tempo quando comparada com a versão original. Tirando o caso já referido, o consumo de memória foi imprevisivel em 3 dos restantes 4 métodos (countLines, dip2px, px2dip) e foi pior na versão com *memoization* para o threadSubject. No entanto, para o maior número de execuções o único método que se manteve imprevisível foi o countLines. Os métodos dip2px e px2dip passaram assim a consumir mais memória aquando da versão com *memoization*.

Embora os resultados sejam bastante positivos e apesar dos ganhos significativos, é preciso considerar o facto de que não podemos descrever o impacto na aplicação geral, uma vez que isolámos os métodos das aplicações de estudo na nossa própria aplicação. Assim, e como na maioria dos casos, cabe aos programadores decidir se utilizam ou não *memoization*. No entanto, mostramos que os métodos onde a técnica foi aplicada são positivos para uma aplicação Android, tendo em conta os ganhos que obtivemos. Em qualquer caso, futuramente, seria bastante relevante estudar o impacto desses ganhos no consumo geral das aplicações.

Para além da análise ao consumo da energia, poderia analisar-se também o consumo geral da energia/tempo/memória nestas aplicações estudadas, poder-se-ia alargar o leque de aplicações e ainda aplicar a técnica numa larga escala de métodos. Para isso, a criação de uma ferramenta que tratasse de todo o processo (desde a identificação dos métodos possíveis a *memoization* até à sua consequente alteração) de forma automatizada seria algo bastante útil e que proporcionaria uma enorme escalabilidade deste trabalho.

Concluindo, este trabalho é um primeiro estudo sobre o tema de *memoization* no sistema Android e como tal as nossas contribuições são uma base inicial para futuros investigadores trabalharem neste sentido. Assim, não só poderão existir estudos do impacto de *memoization* a outros níveis, como o desenvolvimento de ferramentas/técnicas mais robustas baseadas no princípio que se conseguiu provar com esta dissertação: *memoization* pode de facto ajudar na poupança de energia.

Bibliografia

[1] G. Agosta, M. Bessi, E. Capra e C. Francalanci. "Automatic memoization for energy efficiency in financial applications". Em: *Sustainable Computing: Informatics and Systems* 2.2 (2012). IEEE International Green Computing Conference (IGCC 2011), pp. 105 –115. ISSN: 2210-5379. DOI: 10.1016/j.suscom.2012.02.002. URL: http://www.sciencedirect.com/science/article/pii/S2210537912000066.

[2] *Android Studio - Memory Monitor*. URL: https://developer.android.com/studio/profile/am-memory.html (acedido em 23/06/2018).

[3] *Android version history*. URL: https://en.wikipedia.org/wiki/Android_version_history (acedido em 23/06/2018).

[4] S. Bajracharya, J. Ossher e C. Lopes. "Sourcerer: An Infrastructure for Large-scale Collection and Analysis of Open-source Code". Em: *Sci. Comput. Program.* 79 (jan. de 2014), pp. 241–259. ISSN: 0167-6423. DOI: 10.1016/j.scico.2012.04.008. URL: http://dx.doi.org/10.1016/j.scico.2012.04.008.

[5] A. Banerjee e A. Roychoudhury. "Future of Mobile Software for Smartphones and Drones: Energy and Performance". Em: *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 1–12. ISBN: 978-1-5386-2669-6. DOI: 10.1109/MOBILESoft.2017.1. URL: https://doi.org/10.1109/MOBILESoft.2017.1.

[6] C. Bunse, H. Höpfner, S. Roychoudhury e E. Mansour. "Choosing the "Best" Sorting Algorithm for Optimal Energy Consumption." Em: *ICSOFT (2)*. Ed. por B. Shishkov, J. Cordeiro e A. Ranchordas. INSTICC Press, 19 de set. de 2009, pp. 199–206. ISBN: 978-989-674-010-8. URL: http://dblp.uni-trier.de/db/conf/icsoft/icsoft2009-2.html#BunseHRM09.

[7] A. Carette, M. A. Ait Younes, G. Hecht, N. Moha e R. Rouvoy. "Investigating the Energy Impact of Android Smells". Em: *24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER)*. Ed. por A. Marcus e G. Bavota. Proceedings of the 24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER). Klagenfurt, Austria: IEEE, fev. de 2017, p. 10. URL: https://hal.inria.fr/hal-01403485.

[8] J. M. Chambers, W. S. Cleveland, B. Kleiner e P. A. Tukey. *Graphical Methods for Data Analysis*. Chapman e Hall/CRC, 2017. ISBN: 9781315893204.

[9] *Chanu app's FDroid page*. URL: https://f-droid.org/en/packages/com.chanapps.four.activity (acedido em 23/06/2018).

[10] J. Cohen. "Statistical power analysis for the behavioral sciences . Hilsdale". Em: *NJ: Lawrence Earlbaum Associates* 2 (1988).

[11] M. Couto, C. T., J. Cunha, J. P. Fernandes e J. Saraiva. "Detecting Anomalous Energy Consumption in Android Applications". Em: *Programming Languages*. Ed. por F. M. Quintão Pereira. Vol. 8771. LNCS. Springer Int. Publishing, 2014, pp. 77–91.

[12] M. Couto, J. Cunha, J. P. Fernandes, R. Pereira e J. Saraiva. "GreenDroid: A tool for analysing power consumption in the android ecosystem". Em: *Scientific Conf. on Informatics, 2015 IEEE 13th International*. 2015, pp. 73–78.

[13] L. Cruz, R. Abreu e J. Rouvignac. "Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring". Em: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. Vol. 00. 2017, pp. 205–206. DOI: 10.1109/MOBILESoft.2017.21. URL: doi.ieeecomputersociety.org/10.1109/MOBILESoft.2017.21.

[14] L. Cruz e R. Abreu. "Performance-based Guidelines for Energy Efficient Mobile Applications". Em: *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. MOBILESoft '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 46–57. ISBN: 978-1-5386-2669-6. DOI: 10.1109/MOBILESoft.2017.19. URL: https://doi.org/10.1109/MOBILESoft.2017.19.

[15] L. Cruz e R. Abreu. "Using Automatic Refactoring to Improve Energy Efficiency of Android Apps". Em: *CoRR* abs/1803.05889 (2018). arXiv: 1803.05889. URL: http://arxiv.org/abs/1803.05889.

[16] R. B. D'Agostino. "An Omnibus Test of Normality for Moderate and Large Size Samples". Em: *Biometrika* 58.2 (1971), pp. 341–348. ISSN: 00063444. URL: http://www.jstor.org/stable/2334522.

[17] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre e A. Pras. "Inside Dropbox: Understanding Personal Cloud Storage Services". Em: *Proceedings of the 2012 Internet Measurement Conference*. IMC '12. Boston, Massachusetts, USA: ACM, 2012, pp. 481–494. ISBN: 978-1-4503-1705-4. DOI: 10.1145/2398776.2398827. URL: http://doi.acm.org/10.1145/2398776.2398827.

[18] *F-droid*. URL: https://f-droid.org (acedido em 23/06/2018).

[19] A. Field. *Discovering statistics using SPSS*. Sage publications, 2009.

[20] Google. *ADB - Android Debug Bridge*. URL: https://developer.android.com/studio/command-line/adb.html (acedido em 23/06/2018).

[21] Google. *Android Lint Checks*. URL: http://tools.android.com/tips/lint-checks (acedido em 23/06/2018).

[22] Google. *Android's HTTP Clients*. URL: https://android-developers.googleblog.com/2011/09/androids-http-clients.html (acedido em 23/06/2018).

[23] Google. *APK - Android Package*. URL: https://pt.wikipedia.org/wiki/APK (acedido em 23/06/2018).

[24] Google. *ArrayMap*. URL: https://developer.android.com/reference/android/util/ArrayMap.html (acedido em 23/06/2018).

[25] Google. *Bitmap.Config*. URL: https://developer.android.com/reference/android/graphics/Bitmap.Config.html (acedido em 23/06/2018).

[26] Google. *Google Play - Trepn Profiler*. URL: https://play.google.com/store/apps/details?id=com.quicinc.trepn (acedido em 23/06/2018).

[27] Google. *lint*. URL: https://developer.android.com/studio/write/lint.html (acedido em 23/06/2018).

[28] Google. *Pair*. URL: https://developer.android.com/reference/android/util/Pair (acedido em 23/06/2018).

[29] Google. *Performance Tips*. URL: https://developer.android.com/training/articles/perf-tips.html (acedido em 23/06/2018).

[30] Google. *SimpleArrayMap*. URL: https://developer.android.com/reference/android/support/v4/util/SimpleArrayMap.html (acedido em 23/06/2018).

[31] S. Hao, D. Li, W. Halfond e R. Govindan. "Estimating Android applications' CPU energy usage via bytecode profiling". Em: *Green and Sustainable Software (GREENS), 2012 First Int. Workshop on*. 2012, pp. 1–7.

[32] S. Hao, D. Li, W. G. J. Halfond e R. Govindan. "Estimating Mobile Application Energy Consumption using Program Analysis". Em: *Proc. of 35th Int. Conf. on Software Engineering (ICSE)*. 2013.

[33] G. Hecht, R. Rouvoy, N. Moha e L. Duchien. *Detecting Antipatterns in Android Apps*. Research Report RR-8693. INRIA Lille ; INRIA, mar. de 2015. URL: https://hal.inria.fr/hal-01122754.

[34] G. Hecht, N. Moha e R. Rouvoy. "An Empirical Study of the Performance Impacts of Android Code Smells". Em: *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft'16)*. Ed. por L. Flynn e P. Inverardi. Vol. 1. Proceedings of the 3rd IEEE/ACM International Conference on Mobile Software Engineering and Systems 1. Austin, Texas, United States: IEEE, mai. de 2016. URL: https://hal.inria.fr/hal-01276904.

[35] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao e S. Tarkoma. "Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices". Em: *ACM Comput. Surv.* 48.3 (dez. de 2015), 39:1–39:40. ISSN: 0360-0300. DOI: 10.1145/2840723. URL: http://doi.acm.org/10.1145/2840723.

[36] *How does Trepn Power measure battery power?* URL: https://developer.qualcomm.com/forum/qdn-forums/software/trepn-power-profiler/32991 (acedido em 23/06/2018).

[37] *Javatuples*. URL: https://www.javatuples.org/ (acedido em 23/06/2018).

[38] E. G. e Kent Beck. *JUnit*. URL: https://junit.org/junit4/ (acedido em 23/06/2018).

[39] T. K. Kundu e K. Paul. "Improving Android Performance and Energy Efficiency". Em: *2011 24th Internatioal Conference on VLSI Design*. 2011, pp. 256–261. DOI: 10.1109/VLSID.2011.63.

[40] D. Li, S. Hao, W. G. J. Halfond e R. Govindan. "Calculating Source Line Level Energy Information for Android Applications". Em: *Proc. of 2013 Int. Symposium on Software Testing and Analysis*. ISSTA 2013. ACM, 2013, pp. 78–89.

[41] D. Li, Y. Jin, C. Sahin, J. Clause e W. G. J. Halfond. "Integrated Energy-directed Test Suite Optimization". Em: *Proc. of 2014 Int. Symposium on Software Testing and Analysis*. ISSTA 2014. ACM, 2014, pp. 339–350.

[42] D. Li e W. G. J. Halfond. "An Investigation into Energy-saving Programming Practices for Android Smartphone App Development". Em: *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. GREENS 2014. Hyderabad, India: ACM, 2014, pp. 46–53. ISBN: 978-1-4503-2844-9. DOI: 10.1145/2593743.2593750. URL: http://doi.acm.org/10.1145/2593743.2593750.

[43] D. Li, S. Hao, J. Gui e W. G. J. Halfond. "An Empirical Study of the Energy Consumption of Android Applications". Em: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 121–130. ISBN: 978-1-4799-6146-7. DOI: 10.1109/ICSME.2014.34. URL: http://dx.doi.org/10.1109/ICSME.2014.34.

[44] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta e D. Poshyvanyk. "Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study". Em: *Proc. of 11th Working Conf. on Mining Software Repositories*. MSR 2014. ACM, 2014, pp. 2–11.

[45] K. Liu, G. Pinto e Y. D. Liu. "Data-Oriented Characterization of Application-Level Energy Optimization". English. Em: *Fundamental Approaches to Software Engineering*. Ed. por A. Egyed e I. Schaefer. Vol. 9033. LNCS. Springer Berlin Heidelberg, 2015, pp. 316–331.

[46] *Monsoon*. URL: https://www.msoon.com/LabEquipment/PowerMonitor (acedido em 23/06/2018).

[47] S. Mundody e S. K. "Article: Evaluating the Impact of Android Best Practices on Energy Consumption". Em: *IJCA Proceedings on International Conference on Information and Communication Technologies* ICICT.8 (2014). Full text available, pp. 1–4.

[48] K. Nagata, S. Yamaguchi e H. Ogawa. "A Power Saving Method with Consideration of Performance in Android Terminals." Em: *UIC/ATC*. Ed. por B. O. Apduhan, C.-H. Hsu, T. Dohi, K. Ishida, L. T. Yang e J. Ma. IEEE Computer Society, 2012, pp. 578–585. ISBN: 978-1-4673-3084-8. URL: http://dblp.uni-trier.de/db/conf/uic/uic2012.html#NagataYO12.

[49] D. D. Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman e A. D. Lucia. "Software-based energy profiling of Android apps: Simple, efficient and reliable?" Em: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Vol. 00. 2017, pp. 103–114. DOI: 10.1109/SANER.2017.7884613. URL: doi.ieeecomputersociety.org/10.1109/SANER.2017.7884613.

[50] *Number of android apps downloads from google play.* URL: https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/ (acedido em 23/06/2018).

[51] *Number of available applications in the Google Play Store.* URL: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ (acedido em 23/06/2018).

[52] *ODROID-XU4.* URL: http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825 (acedido em 23/06/2018).

[53] R. Pereira, M. Couto, J. Cunha, J. P. Fernandes e J. Saraiva. "The Influence of the Java Collection Framework on Overall Energy Consumption". Em: *Proc. of 5th Int. Workshop on Green and Sustainable Software*. GREENS '16. ACM, 2016, pp. 15–21.

[54] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes e J. a. Saraiva. "Energy Efficiency Across Programming Languages: How Do Energy, Time, and Memory Relate?" Em: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: ACM, 2017, pp. 256–267. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136031. URL: http://doi.acm.org/10.1145/3136014.3136031.

[55] R. Pereira, T. Carção, M. Couto, J. Cunha, J. a. P. Fernandes e J. a. Saraiva. "Helping Programmers Improve the Energy Efficiency of Source Code". Em: *Proceedings of the 39th International Conference on Software Engineering Companion*. ICSE-C '17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 238–240. ISBN: 978-1-5386-1589-8. DOI: 10.1109/ICSE-C.2017.80. URL: https://doi.org/10.1109/ICSE-C.2017.80.

[56] G. Pinto e F. Castor. "Characterizing the Energy Efficiency of Java's Thread-Safe Collections in a Multi-Core Environment". Em: *Proc. of SPLASH'2014 workshop on Software Engineering for Parallel Systems (SEPS), SEPS*. Vol. 14. 2014.

[57] G. Pinto, F. Castor e Y. D. Liu. "Mining Questions About Software Energy Consumption". Em: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 22–31. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597110. URL: http://doi.acm.org/10.1145/2597073.2597110.

[58] *PixelateFreestyle app's FDroid page*. URL: https://github.com/Pixate/pixate-freestyle-android (acedido em 23/06/2018).

[59] Qualcomm. *Trepn Power Profiler - Qualcomm Developer Network*. URL: https://developer.qualcomm.com/software/trepn-power-profiler (acedido em 23/06/2018).

[60] R. Rosenthal. *Meta-analytic procedures for social research*. Vol. 6. Sage, 1991.

[61] R. Rosenthal, H Cooper e L. Hedges. "Parametric measures of effect size". Em: *The handbook of research synthesis* (1994), pp. 231–244.

[62] C. Sahin, L. Pollock e J. Clause. "How Do Code Refactorings Affect Energy Usage?" Em: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '14. Torino, Italy: ACM, 2014, 36:1–36:10. ISBN: 978-1-4503-2774-9. DOI: 10.1145/2652524.2652538. URL: http://doi.acm.org/10.1145/2652524.2652538.

[63] C. Sahin, L. L. Pollock e J. Clause. "From benchmarks to real apps: Exploring the energy impacts of performance-directed changes". Em: *Journal of Systems and Software* 117 (2016), pp. 307–316. DOI: 10.1016/j.jss.2016.03.031. URL: http://dx.doi.org/10.1016/j.jss.2016.03.031.

[64] S. Sawilowsky. "New Effect Size Rules of Thumb". Em: *Journal of Modern Applied Statistical Methods* 8 (nov. de 2009), pp. 597–599.

[65] *Smartphone Platform Market Share*. URL: https://www.businessinsider.com/smartphone-market-share-android-ios-windows-blackberry-2016-8 (acedido em 23/06/2018).

[66] A. R. Tonini, L. M. Fischer, J. C. B. de Mattos e L. B. de Brisolara. "Analysis and Evaluation of the Android Best Practices Impact on the Efficiency of Mobile Applications". Em: *2013 III Brazilian Symposium on Computing Systems Engineering (SBESC)* 00 (2013), pp. 157–158. ISSN: 2324-7894. DOI: doi.ieeecomputersociety.org/10.1109/SBESC.2013.39.

[67] A. Vieira, D. Debastiani, L. V. Agostini, F. Marques e J. C. B. de Mattos. "Performance and Energy Consumption Analysis of Embedded Applications Based on Android Platform". Em: *2012 Brazilian Symposium on Computing System Engineering*. 2012, pp. 59–64. DOI: 10.1109/SBESC.2012.20.

[68] J. Yang, K. Hotta, Y. Higo e S. Kusumoto. "Towards purity-guided refactoring in Java". Em: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 521–525. DOI: 10.1109/ICSM.2015.7332506.

[69] *Yocto-Amp*. URL: http://www.yoctopuce.com/EN/products/usb-electrical-sensors/yocto-amp (acedido em 23/06/2018).