Pós-Graduação em Ciência da Computação

Wellington de Oliveira Júnior

NATIVO OU WEB? UM ESTUDO SOBRE O CONSUMO DE ENERGIA DOS MODELOS DE DESENVOLVIMENTO PARA ANDROID

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2016

Wellington de Oliveira Júnior

NATIVO OU WEB? UM ESTUDO SOBRE O CONSUMO DE ENERGIA DOS MODELOS DE DESENVOLVIMENTO PARA ANDROID

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Fernando José Castor de Lima Filho*

RECIFE
2016

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

O48n Oliveira Júnior, Wellington de
Nativo ou Web? um estudo sobre o consumo de energia dos modelos de desenvolvimento para android / Wellington de Oliveira Júnior. – 2016.
82 f.:il., fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2016.
Inclui referências e apêndices.

1. Engenharia de software. 2. Consumo de energia. I. Lima Filho, Fernando José Castor de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2017-227

Wellington de Oliveira Junior

Nativo ou Web? Um Estudo sobre o Consumo de Energia dos Modelos de Desenvolvimento para Android

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 26/02/2016.

BANCA EXAMINADORA

_____

Prof. Dr. Kiev Santos da Gama
Centro de Informática / UFPE

_____

Prof. Dr. Marco Tulio de Oliveira Valente
Departamento de Ciência da Computação / UFMG

_____

Prof. Dr. Fernando José Castor de Lima Filho
Centro de Informática / UFPE
(**Orientador**)

Eu dedico essa dissertação a minha família, amigos,
professores e colegas, que me ajudaram a torná-la possível.

Agradecimentos

Escrever é sempre uma dificuldade tremenda. Menos, neste caso, é mais.

Gostaria de agradecer primeiro aos meus pais. Sem eles, nada seria possível.

Preciso agradecer enormemente ao meu orientador, Fernando Castor, pela gigantesca paciência, pelas palavras sábias e por ser capaz de me empurrar sempre para frente. Empurrões sempre muito bem vindos.

Gostaria de agradecer especialmente a Bianca Ximenes, por suas noites de sono perdidas me ajudando a escrever, aguentando minhas loucuras e me fazendo ser melhor, e a Weslley Torres, por estar sempre disponível para ajudar um colega em apuros.

A meus familiares,
pelo suporte fornecido.

A meus amigos,
pela diversão proporcionada.

Aos meus colegas e professores,
pelo conhecimento adquirido.

A todos que fazem parte da minha vida,
por existirem.

Obrigado.

If you want to find the secrets of the universe, think in terms of energy, frequency and vibration.

—NIKOLA TESLA

Resumo

Consumo de energia vem se tornando um tópico importante no desenvolvimento de *software*, especialmente dado a ubiquidade dos aparelhos móveis e o fato de a escolha da linguagem de programação influenciar diretamente no consumo de bateria. Este trabalho apresenta um estudo com informações sobre o consumo de energia na plataforma Android. Foram comparados o desempenho e o consumo de energia de 33 *benchmarks* diferentes nas duas principais linguagens usadas para desenvolver aplicativos para Android: Java e JavaScript. Os resultados mostram que aplicações Java podem consumir até 36.27x mais energia, com uma mediana de 2,28x, que as versões em JavaScript, principalmente para os casos que são mais intensos computacionalmente. Em alguns cenários entretanto, os *benchmarks* escritos em Java apresentam uma eficiência energética melhor, com JavaScript chegando a consumir 2,27x mais energia. Baseado nestes resultados, três aplicações escritas em Java foram modificadas para incluir funções em JavaScript que emulem o comportamento de um método equivalente em Java, produzindo aplicações híbridas. Em todas as aplicações modificadas foi possível obter ganho em eficiência energética, contudo, fazer muito uso de invocações entre linguagens pode ser prejudicial, levando os aplicativos a consumir até 1,85x mais energia. Considerando que aplicativos para Android são normalmente desenvolvidos usando Java, os resultados deste estudo indicam que a combinação de JavaScript e Java, usando uma abordagem adequada, pode levar a um ganho de eficiência energética não desprezível.

Palavras-chave: Android. Consumo de Energia. Análise de Desempenho. Benchmarks. Java. JavaScript

Abstract

Energy consumption has become an increasingly important topic in software development, especially due to the ubiquity of mobile devices, and the choice of programming language can directly impact battery life. This dissertation presents a study aiming to shed some light on the issue of energy efficiency on the Android platform, comparing the performance and energy consumption of 33 different benchmarks in the two main programming languages employed in Android development: Java and JavaScript. The results of this work show that Java benchmarks may consume up to 36.27x more energy, with a median of 2.28x, than their JavaScript counterparts, in benchmarks that are mostly CPU-intensive. In some scenarios, though, the Java benchmarks exhibited better energy efficiency, with JavaScript consuming up to 2.27x more energy. Based on these results, three Java applications were re-engineered, and through the insertion of JavaScript functions, hybrid applications were produced. In this three modified applications, improvements in energy efficiency were obtained, but using too many cross-language invocations resulted in more energy being consumed, leading the apps to consume up to 1.85x more energy. Considering that Android apps written in Java are the norm, the results from this study indicate that using a combination of JavaScript and Java may lead to a non-negligible improvement in energy efficiency.

Keywords: Android. Energy Consumption. Performance Analysis. Benchmarks. Java. JavaScript

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

ADB	Android Debug Bridge
API	Application Programming Interface
app	Aplicativo Móvel
CSS	Cascading Style Sheets
CPU	Central Processing Unit
DAQ	Data acquisition systems
GPS	Global Positioning System
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JS	JavaScript
LoC	Lines of Code
kLoC	1000 Lines of Code
mAh	miliampere.hora
NDK	Native Development Kit
QP	Questão de Pesquisa
RAPL	Running Average Power Limit
SANER	IEEE International Conference on Software Analysis, Evolution, and Reengineering
SDK	Software Development Kit
TCLBG	The Computer Language Benchmark Game
tCPU	Tempo de uso da CPU
tExec	Tempo de Execução

Sumário

1

Introdução

Nos últimos anos, computadores têm sofrido modificações de hardware significativas, sendo comum fazerem uso de telas de alta definição e processadores com múltiplos núcleos. Isso ocorreu em paralelo com a crescente necessidade das pessoas de se manterem conectadas o tempo todo, o que motivou o surgimento de novos tipos e padrões de uso de aparelhos. Entretanto, isto vem acompanhado de um custo crescente no consumo de energia. O problema na minimização de gastos com energia já é abordado por várias companhias, por exemplo, o Facebook moveu parte de seus *data centers* para o círculo ártico, fazendo uso do clima para refrigerar seus servidores. Já o Google lançou o Projeto Volta[1] com o Android 5.0, tentando aumentar a eficiência da bateria dos celulares e facilitar a análise do consumo de energia dos aplicativos. A eficiência energética pode parecer uma questão de menor importância se consideramos aparelhos que estão sempre conectados à rede elétrica; contudo, com aparelhos móveis, a autonomia e controle de gasto de energia ganha grande importância.

Smartphones têm se tornado bastante populares, ultrapassando computadores em penetração no mercado[2] e há diversas plataformas disponíveis (por exemplo, Android, iOS, Windows Phone). Desenvolvedores devem escolher se querem desenvolver aplicativos usando a linguagem nativa do sistema operacional móvel ou usar tecnologias para desenvolvimento Web (HTML, CSS e JavaScript), subsequentemente portando os seus apps através de um framework específico (Cordova[3], Ionic[4], Titanium[5]). Complementando à dificuldade de escolher um modelo para seus aplicativos, desenvolvedores têm pouca informação sobre as diferenças entre desempenho e consumo de bateria entre os dois modelos (nativo e web), tornando a sua decisão ainda mais difícil.

Atualmente, o Android é o sistema operacional mais popular para smartphones e tablets. Graças à sua versatilidade, ele é utilizado também em diversos outros tipos de equipamentos como câmeras, relógios, videogames e sistemas inteligentes em carros. Trabalhos anteriores (PATHAK;

[1] http://developer.android.com/about/versions/android-5.0.html#Power
[2] http://www.gartner.com/newsroom/id/2996817
[3] https://cordova.apache.org/
[4] http://ionicframework.com/
[5] http://www.appcelerator.org/#titanium

HU; ZHANG, 2012; COUTO et al., 2014; WILKE et al., 2013; HAO et al., 2013; LI et al., 2013; LINARES-VáSQUEZ et al., 2014; WILKE et al., 2013; ZHANG; HINDLE; GERMÁN, 2014; WILKE et al., 2013; HINDLE, 2012) analisaram o consumo de energia de Android a partir de diversas perspectivas, por exemplo o consumo de métodos, linhas de código, uso de bibliotecas e diferentes formas de utilizar o aplicativo. Contudo, uma perspectiva que ainda não havia sido analisada, se refere ao impacto de diferentes modelos de desenvolvimento para Android no consumo de energia das aplicações.

Aplicativos para Android podem ser desenvolvidos utilizando Java (também chamados de aplicativos nativos) ou utilizando majoritariamente JavaScript (chamados de aplicativos híbridos). Outra opção ainda é o Native Development Kit (NDK)[6], que utiliza C/C++; contudo, aplicativos que usam o NDK não são exclusivamente compostos por código em C/C++, contendo parte de sua lógica em Java. Através de uma amostragem aleatória de 109 aplicativos dos 1600 presentes no F-Droid[7], um repositório online de aplicativos de código aberto, averiguamos que somente cinco foram desenvolvidos fazendo uso de JavaScript.

A tabela 1.1 mostra a parcela dos aplicativos analisados do total de aplicativos presentes no F-Droid. 104 dos 109 aplicativos, foram desenvolvidos em Java, com apenas cinco (4%) destes 104 apresentando parte da sua lógica em C ou C++ através do uso do NDK, e todos sendo aplicativos de jogos. Cinco aplicativos do total de 109 foram escritos quase que completamente usando JavaScript. Ainda não há conclusões definitivas sobre a eficiência energética dessa abordagem em aplicativos Android.

Com esta dissertação, busca-se evidenciar as diferenças entre os dois modelos de desenvolvimento para Android em se tratando de consumo de energia. O NDK é um recurso usado para otimizações e não desenvolvimento de aplicativos, uma vez que não é possível desenvolver somente usando ele. Foram comparados o consumo de energia e o desempenho de 33 diferentes *benchmarks* desenvolvidos por diversos autores do Rosetta Code[8] e do The Computer Language Benchmark Game[9]. Todos os *benchmarks* tinham versões escritas em Java e JavaScript. Eles foram desenvolvidos com a função de solucionar um problema específico ou para comparar as linguagens. Nos casos em que as diferenças de implementação causavam uma diferença muito grande de consumo de energia ou desempenho, os *benchmarks* foram modificados para manter a comparação tão justa e realista quanto possível.

Como forma de medir o consumo de energia e tempo de execução dos aplicativos e *benchmarks* foi utilizado o Projeto Volta do Android. O objetivo deste estudo é fornecer uma resposta para as seguintes questões de pesquisa:

- **QP1. Há um modelo de desenvolvimento, entre os dois mais comumente usados em Android, que seja mais eficiente energeticamente?**

[6] `http://developer.android.com/tools/sdk/ndk/index.html`
[7] `https://f-droid.org/`
[8] `http://rosettacode.org/wiki/Rosetta_Code`
[9] `http://benchmarksgame.alioth.debian.org`

Tabela 1.1: Aplicativos analisados pelo F-Droid. Do total de 109 aplicativos, 104 aplicativos foram desenvolvidos usando Java dos quais 5 fazem uso do Native Development Kit (NDK) e 5 deles foram desenvolvidos usando JavaScript.

Linguagem	Aplicativos
Somente Java	1x1 clock, Anagram Solver, Allsimon/Alldebrid, AndroidRun, android-obd-reader, thialfihar/apg, bpear96/ARChon-Packager, applocker, google/google-authenticator-android, Sash0k/bluetooth-spp-terminal, boardgamegeek, jchmrt/clean-calculator, bitfireAT/cadroid, callmeter, Car Report, CineCat, Clover, CountdownTimer, Cowsay-android, CricketsAlarm,DeepScratch, derandom, dotty, Drinks, DroidBeard, Earmouse, esms, EasyDice, EnigmAndroid, external-ip,falling for reddit, Fish,Flashlight, FreeOTP, frostwire-android, GetBack GPS, Gobandroid, HandyNotes, Hash It!, HeartRateMonitor, HeaterRC,HUD, ICSdroid, IntentRadio, JAWS, Matrix Calc, MAXS Module LocationFine, MAXS Module Ringermode, Migraine Tracker, Movian Remote, MobileOrg, MyOwnNotes, MultiPing, NetMBuddy, Network Discovery, Number Guesser, No Stranger SMS, OI About, OI Notepad, OpenMensa, Page Plus Balance, Photo Bookmark, Permissions, Pocket Talk, PocketSphinx Demo, Prism, Quest Player, RedScreenActivity, ReLaunch, S Tools, sanity, Search Light, SecDroid, Send to SD card, ShoppingList, Simply Do, Sky Map, Sokoban, SparkleShare, Speedo, StockTicker, Sudowars, TaigIME, Temaki, Timber, Toe, Torch, Tri Rose, Twister, Visualizer, Voodoo,CarrierIQ Detector, WebSMS Connector: GMX, weechat, WiFi Warning, WiGLE,Wifi Wardriving, WWWJDIC for Android, Yaacc, YubiClip, YubNub Command Line.
Java usando NDK	24 Game, OpenWnn Legacy, PrBoom For Android, Lumicall, Mitzuli
JavaScript	ankidroid/Anki-Android, clipcaster, Overchan, RainTime, smeir/berlin-vegan-guide

Nós descobrimos que apesar de não haver um vencedor absoluto, JavaScript apresenta uma vantagem significativa em relação aos *benchmarks*. Dos 33 *benchmarks* analisados, JavaScript obteve um menor consumo de energia em 26 deles; para estes *benchmarks*, a mediana do consumo de energia das versões Java foi 1,96x maior. JavaScript também teve um melhor desempenho na maioria dos casos. Houve, entretanto, exceções. Para 6 destes *benchmarks* (*Fannkuch, Knucleotide, Spectral, Happy Numbers, QuickSort e Tower of Hanoi*), Java obteve um desempenho superior aos *benchmarks* em JavaScript, apesar de acabar consumindo mais energia. Em 4 destes 6 (*Fannkuch, Knucleotide, Spectral e QuickSort*), o desempenho melhor se deve ao uso de paralelismo. Isto indica que, pelo menos para aplicativos que utilizam a CPU intensamente ou que façam exclusivamente operações aritméticas, Java pode não ser a melhor opção. Apesar disso, aplicativos podem apresentar um comportamento muito diferente de *benchmarks* (RATANAWORABHAN; LIVSHITS; ZORN, 2010), uma vez que boa parte de seu tempo de execução é gasto esperando entradas do usuário ou fazendo uso de sensores, como *GPS*, *Wi-fi* ou *3G*. Isto levanta a questão de que talvez seja possível economizar energia utilizando um modo de desenvolvimento híbrido, i.e., adotando JavaScript para executar parte das aplicações que sejam mais intensas em uso de CPU. Por conseguinte, apresentamos uma resposta inicial para a seguinte questão de pesquisa adicional:

- **QP2. É possível reduzir o consumo de energia de um aplicativo que foi desenvolvido utilizando linguagem nativa ao torná-lo híbrido?**

Nós modificamos três aplicativos, disponíveis no F-Droid, escritos totalmente em Java,

e incluímos trechos de JavaScript na sua lógica. Analisamos diferentes modelos para que os aplicativos em Java invocassem as funções necessárias em JavaScript e medimos o consumo e desempenho de todos os casos. Nossos resultados indicam que é possível economizar energia fazendo uso de um modo de desenvolvimento híbrido. Em um dos aplicativos, TriRose, a versão híbrida chegou a consumir 30% menos energia que a versão Java, ao agrupar invocações em JavaScript. Isto é uma indicação que agrupar é uma maneira de reduzir o overhead de comunicação entre as linguagens. Agrupar as mesmas operações em Java não acarreta nenhum ganho significativo de desempenho ou redução no consumo de energia. Entretanto, as aplicações que foram adaptadas com uma granularidade muito baixa tiveram um desempenho inferior ao da aplicação original em Java.

As mudanças nos aplicativos foram relativamente pequenas, chegando a 3% do código no caso do EngimAndroid, e resultaram em uma diminuição significativa no consumo de energia. Saber disso dá aos desenvolvedores a possibilidade de escolher o modo de desenvolvimento que eles preferem ou até mesmo optar por desenvolver uma solução usando mais de uma linguagem, fazendo uso ótimo das características de cada uma delas. Ademais, desenvolvedores de ferramentas podem introduzir refatorações entre linguagens para dar suporte à modificações em aplicações existentes, quando torná-las híbridas for benéfico. Análise de programas pode ser utilizado nessas ferramentas para ajudar os desenvolvedores a identificarem estes casos. Os dados relativos a este trabalho estão disponíveis em `http://nativeorweb.github.io/`.

Esta dissertação está organizada segundo a seguinte estrutura: No Capítulo 2, fazemos a fundamentação teórica com o objetivo de fornecer o conhecimento necessário para acompanhar o resto da dissertação. No Capítulo 3 apresentamos a metodologia utilizada no desenvolvimento deste trabalho, como foram feitas as modificações nos *benchmarks* onde isto foi necessário e o ambiente de execução do experimento. No Capítulo 4, apresentamos os resultados coletados e realizamos a análise dos experimentos realizados. No Capítulo 5, relacionamos outros trabalhos que tratam sobre o consumo de energia, com foco nos trabalhos que lidam com o consumo de energia em aparelhos móveis. Finalmente, no Capítulo 6, sintetizamos as principais contribuições científicas e aplicadas deste trabalho.

2

Fundamentação

Este capítulo tem como objetivo detalhar os pilares nos quais este trabalho foram fundamentados. O capítulo conta com uma seção sobre o consumo de energia de aparelhos móveis (Seção 2.1), uma seção sobre o desenvolvimento de aplicativos Android (Seção 2.2) e uma seção sobre *frameworks* de aplicações híbridas (Seção 2.3).

2.1 Consumo de energia de aparelhos móveis

Por muito tempo, o consumo de energia de computadores era creditado aos componentes de hardware em conjunto com o sistema operacional. TIWARI; MALIK; WOLFE (1994) mostrou, entretanto, que software também é parte fundamental do consumo de energia de computadores. Desde então, diversas pesquisas focaram em tentar reduzir o consumo de energia através de modificações de software, conseguindo resultados relevantes (PINTO; CASTOR; LIU, 2014; HINDLE, 2012; COHEN et al., 2012; LI; HALFOND, 2014; MANOTAS; POLLOCK; CLAUSE, 2014). A descrença na importância do software fomentava a falta de preocupação dos desenvolvedores com o consumo de energia de seus programas, havendo consequentemente uma escalada constante no consumo de energia de computadores no decorrer dos anos (ASAFU-ADJAYE, 2000).

Mais recentemente, viu-se um florescer das tecnologias móveis e o uso constante de baterias para fornecer energia aos aparelhos. Neste contexto, onde não há uma fonte de energia constante, o consumo de energia é crítico e qualquer melhora em relação a eficiência energética pode influenciar positivamente a experiência do usuário.

As seções seguintes relacionam as diversas técnicas utilizadas para medir o consumo de energia de aparelho móveis, foco desta pesquisa. As formas de medir o consumo serão divididas em duas seções: **2.1.1 Medição de granularidade grossa** e **2.1.2 Medição de granularidade fina**

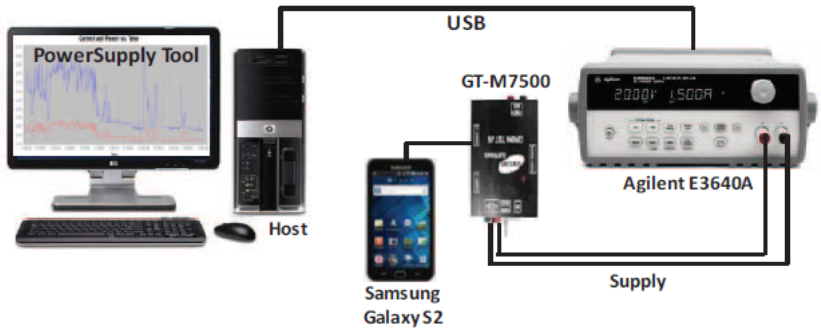

Figura 2.1: Esquemático de como fazer uso de uma bateria programável para realizar medições de energia. Figura retirada de SILVA-FILHO et al. (2012).

2.1.1 *Medição de granularidade grossa*

Medições de granularidade grossa analisam o consumo de energia de forma geral, não fazendo distinção entre os diversos fatores ou componentes que podem influenciar o consumo. Normalmente se fazem valer de aparelhos físicos para realizar as medições, recorrendo a software somente para analisar os dados coletados.

Uma solução é utilizar uma bateria externa programável para medir o consumo, como ilustrado na figura 2.1. A bateria programável permite o ajuste de corrente e potência, o que torna o consumo de energia mensurável dentro de um determinado intervalo de tempo. Os dados refentes ao consumo de energia pela bateria programável são então transferidos para o computador para serem analisados (SILVA-FILHO et al., 2012). Uma grande vantagem desta abordagem é o controle sobre a quantidade de energia que o aparelho estará consumindo, variando de acordo com o grau de precisão da bateria, mas como desvantagem nos temos o fato de que o uso de uma bateria externa não representa um cenário de uso padrão.

Outra técnica é deixar uma aplicação ou *benchmark* sendo executado até que o smartphone consuma toda sua bateria. Ao isolar o comportamento do aparelho, deixando somente o aplicativo ou *benchmark* que se deseja analisar executando, a quantidade de energia consumida pelo aplicativo fixa, restando somente o tempo como variável, tornando a mensuração de energia pelo aplicativo trivial. (COUTO et al., 2014). Como vantagem desta abordagem está o fato de que não é necessário o uso de aparelhos externos para realizar a medição, mas como desvantagem nos temos o fato que a idade da bateria, e quanto de energia ela consegue manter, influencia na medição.

Alguns estudos fazem uso de uma pinça de corrente com o objetivo de medir a diferença de corrente que está sendo transmitida para o aparelho, juntamente a um *Data Acquisition System* (DAQ) para coletar os sinais do que está sendo consumido no aparelho e coletado pela pinça (LI et al., 2013; SABORIDO et al., 2015; PETERSON et al., 2011). A figura 2.2 ilustra um ambiente utilizado para este tipo de abordagem. Esta abordagem apresenta como vantagem o fato de não interferir com a bateria padrão do aparelho e por lidar com a corrente, não está sujeita a idade da bateria. A grande desvantagem está principalmente na dificuldade em realizar as medições, que

Figura 2.2: Ambiente utilizado para medir o consumo utilizando um DAQ (*Data Acquisition System*). Figura retirada de PETERSON et al. (2011).

necessita de diversos aparelhos externos.

No caso específico de computadores que façam uso de processadores Intel, há a possibilidade de utilizar o Intel RAPL (*Running Average Power Limit*) para realizar as medições. O Intel RAPL é composto por um conjunto de contadores que fornecem informações sobre consumo de energia, e não é um medidor de potência analógico, ao invés disso, usando um modelo de potência de software. Este modelo de potência de software calcula o consumo de energia usando contadores de desempenho de hardware e modelos de entrada e saída. Diversos trabalhos fazem uso do Intel RAPL para medir o consumo de energia (SUBRAMANIAM; FENG, 2013; LIU; PINTO; LIU, 2015; KAMBADUR; KIM, 2014). Seria possível realizar medidas utilizando o Intel RAPL neste trabalho através do uso do emulador do Android, porém esta opção foi descartada uma vez que não simula o uso real de um aparelho móvel.

Como grande vantagem das medições de granularidade grossa nós temos a confiabilidade nos dados. Uma vez que a medição está sendo realizada através de aparelhos externos ao celular, não há o *overhead* de coleta dos dados na aplicação. Estes métodos de medir também são usados em outras áreas acadêmicas e na indústria, tendo sua confiabilidade averiguada (HäHNEL et al., 2012);

Uma das barreiras encontradas para o uso de soluções de granularidade grossa é que, uma vez que elas medem o consumo do aparelho como um todo, elas não detectam comportamentos específicos de um aplicativo particular, não sendo possível discriminar o consumo de sensores (*GPS*, *WI-FI*, Acelerômetro) ou outros aplicativos ao mesmo tempo que medem o aplicativo de teste. Uma vez que nosso objetivo é focar no consumo de *benchmarks* e aplicativos, preferimos focar em abordagens de granularidade fina.

2.1.2 *Medição de granularidade fina*

As medições de granularidade fina podem apontar de forma mais precisa onde a energia está sendo gasta. Para este estudo, partimos do pressuposto que meios de granularidade fina são aqueles que realizem medições obtendo o consumo de energia de um aplicativo ou algo mais específico (métodos, linhas de código).

Uma forma comumente usada de realizar a medição de granularidade fina é a instrumentação de código em aplicativos para medir o desempenho ou consumo, permitindo diversas opções em relação ao nível de granularidade. Métodos (COUTO et al., 2014), chamadas de API, (LINARES-VáSQUEZ et al., 2014) nas aplicações como um todo (WILKE et al., 2013) e até linhas de código específicas (HAO et al., 2013; LI et al., 2013). Há, entretanto, um *trade-off* quando se opta por um nível de granularidade muito fino. A instrumentação do código consome energia (HäHNEL et al., 2012), e o overhead imposto à aplicação para mensurar o consumo pode acabar por elevar de forma significativa o consumo da mesma, tornando os dados coletados menos precisos. Se a instrumentação não é feita pelo próprio time desenvolvedor, há também a necessidade do aplicativo ser de código aberto, uma vez que é necessário inserir trechos de código para realizar as medições. A opção para aplicativos de código fechado é utilizar um desmontador (*disassembler*), revertendo o *bytecode* para código Java. Esta opção só é possível em aplicações Java. Por conta destas limitações, a instrumentação do código se torna uma solução de baixa abrangência.

Duas soluções que fazem uso de granularidade fina sem instrumentação usadas para medir o consumo de energia e desempenho das aplicações e utilizadas na prática são: o Trepn Profiler da Qualcomm e o Projeto Volta da Google.

O Trepn Profiler[1] é um *profiler* de energia e desempenho para dispositivos móveis desenvolvido pela Qualcomm. Ele é projetado para ajudar a identificar o uso de CPU, consumo de dados ou bateria em excesso. Embora site do Trepn Profiler informe que ele funciona na maioria dos dispositivos Android, em nossos testes ele não funcionou de forma adequada no Moto G3 nem em um Samsung J7. O site informa ainda que recursos adicionais estão disponíveis ao utilizar aparelhos com processadores Qualcomm Snapdragon, como por exemplo LG G4, Moto X Pure Edition, Microsoft Lumia 950 XL, HTC One M9 e Nexus 5.

O Projeto Volta é um conjunto de ferramentas disponibilizada pela Google que, dentre outras coisas, fornece APIs e mais acesso à informação de consumo de bateria e desempenho dos aplicativos, tendo como objetivo fornecer dados e meios para que os desenvolvedores possam reduzir o consumo de energia de seus apps. O Projeto Volta está disponível a partir da versão 5.0 do Android, Lollipop, lançada em 2014. Pouca informação foi divulgada sobre seu funcionamento, por exemplo, a taxa de amostragem que é usada para coletar os dados de consumo de bateria.

Ambas as soluções têm as vantagens de utilizar um método de medição de granularidade

[1] `https://developer.qualcomm.com/software/trepn-power-profiler`

fina, sendo possível identificar exatamente onde o aparelho está gastando energia. Uma vez que nenhuma delas depende do código fonte da aplicação para realizar suas medições, ambas apresentam uma alta abrangência, sendo possível medir o consumo de energia de aplicações de código aberto ou não. Neste trabalho, entretanto, optamos por utilizar somente o Projeto Volta, pelas seguintes razões:

- O Projeto Volta está disponível em todos os aparelhos Android, a partir da versão 5.0 ou maior. Isto nos dá a liberdade de rodar os experimentos em qualquer tipo de aparelho, com todos os recursos disponíveis pela Google. Utilizar o Trepn nos limitaria em questão de recursos, caso desejássemos utilizar aparelhos que não fizessem uso do processadores Snapdragon, e limitaria que aparelhos poderíamos usar, uma vez que nem todos os aparelhos no mercado fazem uso dos processadores Snapdragon. Essa limitação prejudicaria a generalidade dos resultados deste trabalho.

- O Trepn Profiler funciona como um aplicativo. Apesar de não realizar instrumentação do código, o fato de haver dois aplicativos rodando ao mesmo tempo, o Trepn Profiler e o aplicativo do qual se deseja analisar o consumo, poderia influenciar na medição de energia. O Projeto Volta faz sua análise sem a necessidade de rodar um aplicativo separado.

- O fato de o Projeto Volta ser disponibilizado pela Google, empresa responsável pelo desenvolvimento do sistema operacional Android, torna a futura replicação dos experimentos mais fácil. Por ser apoiado pela Google e vir incluso em todos os aparelhos, há uma probabilidade menor que este recurso não esteja disponível no futuro para outros estudos quando comparado ao Trepn Profiler.

Detalhes sobre o uso das ferramentas disponíveis através do Projeto Volta serão apresentados no capítulo 3.

2.2 Desenvolvimento de Aplicativos Android

O Android é um sistema operacional móvel de código aberto desenvolvido pela Google. Atualmente, é o sistema operacional mais utilizado do mundo, superando inclusive sistemas mais tradicionais como o Windows da Microsoft[2]. Apesar da popularidade, a plataforma Android ainda não é tão madura, passando por mudanças profundas na sua infraestrutura que podem afetar radicalmente o desempenho de diversos aplicativos (exemplo: mudança da máquina virtual Dalvik para Android Runtime[3]).

Inicialmente, para desenvolver aplicações Android é preciso determinar qual será a forma de construção da aplicação. Aplicações nativas, i.e., desenvolvidas utilizando a linguagem nativa

[2] `http://www.gartner.com/newsroom/id/2996817`
[3] `http://developer.android.com/about/versions/lollipop.html#Perf`

do sistema operacional, no caso Android, Java, tem sido vistas como mais rápidas, mais seguras e se adaptam melhor às mudanças de sistema operacional[4]. Aplicações híbridas, isto é, aplicações usando tecnologias da web, são vistas como mais portáveis, com menor custo de manutenção e desenvolvimento e são mais rápidas para serem lançadas no mercado[4].

Normalmente os aplicativos são desenvolvidos em Java por ser a linguagem nativa do sistema operacional, com diversos tutoriais, incluindo o da própria Google. É mais simples começar a desenvolver por esse caminho pois o programador tem acesso a todas as APIs e bibliotecas, além da documentação e suporte da Google. Para desenvolver utilizando Java, só é preciso utilizar sua IDE(*Integrated Development Environment*) favorita (a Google recomenda o uso da sua IDE própria, o Android Studio). Com base nos conhecimentos prévios de Java, o desenvolvedor por criar aplicativos simples, aplicativos mais complexos entretanto, podem demandar o uso das APIs específicas e um conhecimento maior da infraestrutura do Android.

Desenvolvedores que optem por desenvolver aplicações híbridas sem experiência prévia em desenvolvimento móvel podem encontrar mais dificuldade inicialmente em relação a tutoriais ou suporte de IDE. O próprio Android Studio oferece suporte limitado ao uso de JavaScript. Para conseguir desenvolver aplicações híbridas também é necessário o uso de um ***framework* de desenvolvimento móvel** para encapsular a aplicação. Esses *frameworks* serão discutidos na Seção 2.3. Entretanto, se o desenvolvedor tiver experiência com o desenvolvimento para *web*, poderá então desenvolver sua aplicação de forma muito mais fácil, uma vez que as APIs do Android são acessíveis através de simples chamadas de plugins do *framework*, e o desenvolvimento *per se* não é muito diferente do desenvolvimento de um *WebApp*. Aplicações mais complexas podem necessitar de um maior conhecimento do *framework* de desenvolvimento.

No caso específico do Android há ainda a possibilidade de desenvolver aplicativos fazendo uso do NDK, utilizando C/C++ como forma melhorar a performance do aplicativo. Apesar da existência do NDK, aplicativos em Android são comumente escritos usando Java, JavaScript ou uma combinação de ambos. Isto se deve às complicações intrínsecas do NDK, como apontado no próprio site da Google: "*(...) has little value for many types of Android apps. It is often not worth the additional complexity it inevitably brings to the development process*"[5]. Não é possível criar um aplicativo usando somente o NDK, uma vez que parte dele tem que estar escrito em Java. Na análise dos aplicativos no F-Droid, foram encontrados aplicativos que faziam uso do NDK, dando suporte às aplicações em Java, como forma de melhorar seu desempenho. Estas aplicações representavam uma parcela pequena da nossa amostra, somente 4% dos aplicativos, e todos eles eram jogos, o que pode indicar que o uso do NDK está restrito a um nicho bastante específico.

[4] `https://www.ibm.com/developerworks/community/blogs/mobileblog/entry/swot_analysis_hybrid_versus_native_development_in_ibm_worklight?lang=en`

[5] `http://developer.android.com/ndk/guides/index.html`

Figura 2.3: Diversos níveis da infraestrutura do Android. Figura retirada de `http://mihasoftware.com/2014/02/android-new-runtime/`.

.

2.2.1 *Infraestrutura Android*

A infraestrutura do Android é composta de vários níveis: O nível de aplicações; *Framework* de Aplicação; Bibliotecas e *Runtime*; Camada de Abstração de *Hardware* e finalmente o *Kernel*, como ilustrado na figura 2.3. Todas as aplicações Android, sejam elas nativas ou híbridas, rodam no nível de aplicação. A camada de aplicação é composta de diversos elementos que permitem acesso do sistema ao aplicativo. A seguir vamos descrever os quatro principais componentes, cada um com uma finalidade específica. Como descritos no site da Google[6]:

Atividades: são componentes que representam as telas da aplicação. Cada atividade é independente da outra e pode representar qualquer elemento relevante dentro da aplicação. É possível para um aplicativo chamar uma atividade de outro aplicativo.

Serviços: são componentes que realizam um trabalho externo à aplicação, invocando algum tipo de recurso disponível no aparelho, em *software* ou *hardware*. Por exemplo, um serviço pode tocar música em segundo plano enquanto o usuário está em um aplicativo diferente, ou buscar dados na rede sem bloquear a interação do usuário com uma atividade.

Provedores de conteúdo: gerenciam um conjunto compartilhado de dados do aplicativo e são comumente usados para gerenciar bancos de dados como SQLite. Fazendo uso do provedor de conteúdo, outras aplicações podem aproveitar os dados coletados, por exemplo, acessar a lista de contatos do usuário para adicioná-los em um aplicativo de rede social.

Receptores de transmissão: são componentes que respondem a anúncios de transmissão

[6] `http://developer.android.com/guide/components/fundamentals.html`

por todo o sistema. Desligar a tela ou informar ao usuário que a bateria esta baixa são exemplos de eventos que enviam transmissões para tela. Normalmente o receptor de transmissão é usado para fazer a interface entre outros componentes, como inicializar um serviço baseado num evento que aconteceu numa atividade.

Nesta pesquisa, dos quatro componentes básicos de uma aplicação Android, só foram usados intensamente os recursos do componente de Atividades. Todos os outros componentes foram utilizados de forma mínima, devido a natureza dos *benchmarks* e apps analisados.

2.3 Framework de aplicações Híbridas

Para tornar possível o desenvolvimento de aplicativos para smartphones usando as tecnologias *web*, como HTML5, CSS3 e JavaScript, é comum utilizar *frameworks* de desenvolvimento móvel. Estes *frameworks* tem como objetivo permitir a utilização das tecnologias padrão de desenvolvimento para *web* para desenvolvimento entre linguagens, isto é, a aplicação é desenvolvida uma única vez e portada para diversos sistemas operacionais móveis diferentes (iOS, Android, BlackBerry, Windows Phone, Firefox OS), evitando que o desenvolvedor tenha que lidar com a linguagem nativa de cada um dos sistemas operacionais. Estes aplicativos são então envolvidos por um *wrapper*, fazendo com que o sistema operacional possa identificá-lo como um aplicativo tradicional e o mesmo tenha acesso a APIs nativas.

Este trabalho foi desenvolvido utilizando o Apache Cordova como um *framework* de desenvolvimento móvel. O Apache Cordova é de código aberto e é utilizado como base para outros *frameworks* bastante populares como Ionic, Intel XDK, Monaca, TACO, e Telerik. Entre os contribuidores do projeto Apache Cordova estão a Adobe, IBM, Google, Microsoft, Intel, Blackberry, Mozilla e outros[7]. A escolha do Apache Cordova se deve a sua estabilidade e pela possibilidade de de portar estes aplicativos para outros *frameworks* caso fosse necessário, uma vez que diversos outros *frameworks* usam o Cordova como parte de sua infraestrutura.

As aplicações desenvolvidas usando o Apache Cordova dependem de um arquivo comum config.xml que fornece informações sobre o aplicativo e sobre como ele deve funcionar, como por exemplo, se ele deve ou não responder à mudança de orientação da tela. Utilizando o Cordova, é possível ter acesso à API nativa do sistema operacional, tornando possível o acesso a recursos fundamentais no desenvolvimento de um aplicativo como o uso de *GPS*, funções de vibração do aparelho, e acesso a outros aplicativos como a localização no mapa através do Google Maps.

O aplicativo é desenvolvido como uma página web, podendo referenciar quaisquer arquivos CSS, JavaScript, imagens, arquivos de mídia ou outros recursos necessários para executá-lo. Este aplicativo será executado por cima de uma *WebView* encapsulada dentro de uma aplicação nativa, tornando possível distribui-lo como um aplicativo. No caso específico do Android, o aplicativo será compilado dentro de um arquivo .apk, que é o formato padrão de distribuição de aplicativos Android.

[7] `http://wiki.apache.org/cordova/who`

Os aplicativos gerados pelo Cordova fazem uso de *WebViews* para executar o código e fazer a renderização, o que constitui um comportamento diferente do habitual dos aplicativos nativos. As *WebViews* são nativas e não podem ser modificadas pelo usuário.

Uma interface de plugin está disponível, tornando possível a comunicação entre o aplicativo no Cordova e os componentes nativos. Isto possibilita a invocação de trechos nativos dentro do código JavaScript. Idealmente, as APIs de JavaScript para código nativo são consistentes através de múltiplas plataformas. Há também plugins disponíveis externamente que permitem uso de recursos não disponíveis em todas as plataformas. Os plugins utilizados durante o desenvolvimento deste trabalho foram: *vibration*, para tornar possível usar a função de vibrar do aparelho; *file*, para ter acesso a arquivos .txt; e *whitelist* para controle de URLs.

Pela quantidade de recursos disponíveis no Cordova, o seu uso pode ir do extremamente simples, como foi o caso dos recursos utilizados nesta pesquisa, aos extremamente complexos, como no caso do Halo Waypoint da Microsoft[8]. Como forma de minimizar o impacto do *framework* na comparação entre os *benchmarks*, foram usados os recursos mínimos para tornar a execução possível.

[8] `http://halo-waypoint.en.softonic.com/android`

3

Metodologia

O objetivo deste estudo é analisar os modelos de desenvolvimento de aplicativos para Android mais populares e verificar se eles diferem em relação ao consumo de energia e desempenho. Nesta seção, são descritos os *benchmarks* analisados, o processo de hibridização dos aplicativos e o nosso procedimento experimental. Todo o desenvolvimento relacionado a Android desta pesquisa foi realizado usando o Android Studio.

A decisão de executar experimentos usando *benchmarks* e apps se deve ao fato de que, no caso do primeiro, podemos focar em partes específicas da aplicação que desejamos testar, como intensa atividade de CPU, uso de memória ou diversas estruturas de dados, já os apps, tem como objetivo simular um uso real do aparelho. Os apps usados nesta pesquisa são aplicativos reais coletados através do portal F-Droid.

Este capítulo tem como objetivo detalhar o procedimento utilizado para realizar os experimentos. O capítulo conta com uma seção realizando uma análise sobre os *benchmarks* (Seção 3.1), uma seção realizando a análise dos aplicativos modificados (Seção 3.2) e uma seção sobre a execução dos experimentos (Seção 3.3).

3.1 Benchmarks

Nesta seção, apresentamos os *benchmarks* estudados para responder a QP1. Os três aplicativos modificados para utilizar a abordagem híbrida serão discutidos no capítulo 4. Dois repositórios serviram como base para a coleta de *benchmarks* nesta pesquisa: O Rosetta Code e o The Computer Language Benchmarks Game.

O Rosetta Code é um site que apresenta um compêndio de soluções em diversas linguagens de programação para um mesmo problema proposto. O site inclui centenas de problemas e soluções em cerca de 600 linguagens de programação diferentes. Além disso, para alguns problemas, diversas soluções viáveis são propostas. Este trabalho selecionou as soluções identificadas como mais eficientes. O objetivo do Rosetta Code é ensinar a programar através do ato da programação em si, tornando possível para os desenvolvedores validar soluções, tanto analisando a saída esperada do programa quanto o algoritmo.

Tabela 3.1: O conjunto de *benchmarks* e aplicativos selecionados. Todos incluem versões em Java e JavaScript.

Fonte	***Benchmark* ou Aplicativo**
Rosetta Code	BubbleSort, Combinations, Count in factors, CountingSort, GnomeSort, Happy Numbers, HeapSort, HofstadterQ, InsertSort, Knapsack Bounded, Knapsack Unbounded, Man or Boy, Matrix Multiplication, MergeSort, nQueens, PancakeSort , Perfect Number, QuickSort, SeqNonSquares, ShellSort , Sieve of Eratosthenes, Tower of Hanoi e Zero-One Knapsack.
The Computer Language Benchmark Game	BinaryTree, Fannkuch, Fasta M Core, Fasta S Core, Nbody, RegexDna M Core, RegexDna S Core, RevComp, Knucleotide e Spectral
F-Droid	anDOF, EnigmAndroid e Tri Rose

O The Computer Language Benchmarks Game (TCLBG) é um site onde o principal objetivo é comparar o desempenho de diversas linguagens de programação. Inicialmente, o site era um projeto pessoal de um desenvolvedor que tinha como objetivo comparar linguagens que julgava interessantes de maneira informal, mas a ideia se tornou atraente para outros programadores que ajudaram a expandir o escopo das linguagens disponíveis e otimizaram as respostas.

A decisão de usar estes sites foi consequência da quantidade de *benchmarks* diferentes de que eles dispõem em ambas as linguagens. Os *benchmarks* do Rosetta foram selecionados dentro da gama de *benchmarks* previamente utilizados em outros artigos (NANZ; FURIA, 2015). Todos os *benchmarks* disponíveis no TCLBG foram utilizados.

É importante ressaltar que os códigos dos *benchmarks* não são escritos por uma única pessoa, sendo refinado através do tempo por diversos desenvolvedores, reduzindo assim a probabilidade de enviesamento dos experimentos. Apesar disso, nos tivemos que modificar alguns *benchmarks*, como será discutido na Seção **3.1.1**.

Ambos os sites já foram utilizados para comparar o desempenho de linguagens de programação (NANZ; FURIA, 2015) e análise de consumo de energia (LIMA et al., 2016)

O conjunto de *benchmarks* deste trabalho contempla 23 provenientes do Rosetta Code e 10 do TCLBG, todos possuem versões originalmente escritas tanto em Java quanto em JavaScript. A Tabela 3.1 mostra a lista de todos os *benchmarks* utilizados, e no Apêndice há uma descrição resumida dos *benchmarks*. Como critério para seleção dos *benchmarks*, eles deveriam estar presentes em ambas as linguagens. Alguns *benchmarks* do Rosetta Code presentes em outros estudos foram excluídos (por exemplo, SleepSort) embora satisfizessem esse critério, pois apresentavam um tempo de execução muito pequeno em uma das linguagens (inferior a 5 segundos de execução) ao passo que era extremamente lento na outra, o que culminava no travamento da aplicação. Todas as implementações presentes no TCLGB em ambas as linguagens foram utilizadas neste estudo. Uma vez que os *benchmarks* presentes no Rosetta Code têm como principal objetivo solucionar um problema, e não garantir desempenho. É importante notar que,

por causa disso, o desempenho pode variar de forma significativa entre os diferentes *benchmarks*. Como o objetivo da pesquisa é analisar as diferenças no consumo de energia e desempenho entre as linguagens, e não entre as implementações dos algoritmos, todos os *benchmarks* do Rosetta Code foram verificados e, quando necessário, foram feitas modificações para torná-los mais compatíveis com suas contrapartes escritas na outra linguagem, i.e., deixando os algoritmos em Java e JavaScript num nível similar de eficiência. Um exemplo desse tipo de modificação é apresentado na Seção 3.1.1.

No caso do TCLBG, há uma preocupação explícita com a otimização dos algoritmos. Os programadores envolvidos buscam alcançar o melhor desempenho possível na linguagem em que estão codificando a solução do *benchmark*, uma vez que o principal objetivo deste site é comparar qual linguagem é a mais rápida num caso específico. As únicas modificações realizadas nos *benchmarks* presentes no TCLBG foram feitas para que fosse possível executar no ambiente Android, i.e., executar o *benchmark* através de um aplicativo, sem utilizar o console ou importar os arquivos necessários através da API do Android. Diversas implementações de *benchmarks* em Java utilizam paralelismo para solucionar os problemas, promovendo um aumento do desempenho. Algumas vezes isso implica em um aumento de consumo de energia, sacrificando eficiência energética (PINTO; CASTOR; LIU, 2014). Implementações dos mesmos *benchmarks* otimizados para um único núcleo também foram analisadas, como forma de verificar se uma execução mais lenta poderia render uma eficiência energética superior às soluções em Java. Todos os *benchmarks* em JavaScript foram otimizados para um único núcleo, uma vez que a linguagem possui suporte limitado a paralelismo e nenhuma das soluções presentes no repositório fazia uso deste recurso.

Todos os *benchmarks* foram executados usando uma carga de trabalho predefinida de forma individual para cada *benchmark*. O tamanho da carga de trabalho foi determinado para que o *benchmark* executasse por pelo menos 20s e o processo foi repetido 10 vezes em cada caso. Não foi realizado *warmup* dos *benchmarks* ou aplicativos uma vez que testes preliminares mostraram que o uso de *warmup* não alterava os resultados. Todas as compilações foram feitas usando o modo *release* do Android Studio.

Considerando as duas versões de cada um dos 33 *benchmarks*, somente cinco (dos 66) tiveram um desvio padrão relativo maior do que 18% para a média dos desvios de consumo de energia. Isto indica que, para a configuração experimental que utilizamos, os resultados foram estáveis. Todos os casos em que um *benchmark* teve um desvio padrão relativo maior do que 18% foram executados em menos de 20s. Para aumentar o tempo de execução destes casos seria necessário aumentar a carga de trabalho, o que não foi possível uma vez isto incorria em travamento do benchmark em uma das linguagens.

Os *benchmarks* foram executados seguindo dois fluxos distintos, um para Java e outro para JavaScript. O modelo usado para Java foi de executar todos os *benchmarks* dentro de uma única aplicação Android. Esta aplicação roda somente uma Atividade, *MainActivity*. Esta Atividade apresentava uma caixa de texto em branco que ao final da execução do *benchmark*

mostrava o texto "OVER!". O modelo usado para JavaScript foi o de executar todos *benchmarks* dentro de uma aplicação Android criada pelo Cordova. Esta aplicação consistia em apenas uma tela com dois elementos: um texto maior contendo "Apache Cordova"e um texto menor "AA", que ao final da execução era trocado por "IT'S OVER"no maior e branco no menor. O segundo texto era usado para depuração durante o adaptação dos *benchmarks*. As telas dos aplicativos podem ser visualizadas no Apêndice . Em todos os casos em que foi necessário usar algum arquivo externo, como um arquivo texto de entrada, os arquivos foram alocados na pasta "assets".

Durante a execução dos *benchmarks*, foram desenvolvidos dois aplicativos base, um para executar todos os *benchmarks* em Java e outro para executá-los em JavaScript. No caso de Java, cada *benchmark* era representado por uma classe e a classe principal, *MainActivity*, só chamava as classes referentes ao *benchmark* que deveria ser executado; todas as demais classes não eram sequer importadas. No caso de JavaScript, foram utilizados dois arquivos .js, um com todos os *benchmarks* do Rosetta Code e outro com todos os *benchmarks* do TCLBG.

A utilização de aplicativos criados através do Cordova vem da motivação de simular um cenário de utilização real de aplicativos híbridos. Aplicativos utilizando Javascript são comumente desenvolvidos usando um *framework* para portá-los. Um motivador importante para utilizar o Cordova foi a tentativa de também estimar o tamanho do *overhead* que estaria sendo imposto pelo uso do *framework*. Durante a pesquisa, não se encontrou um mesmo aplicativo desenvolvido pelo mesmo grupo de pessoas nos dois métodos distintos (Java para Android e web). Portanto, usar o Cordova nos permite, dentro de certas restrições, comparar dois aplicativos exatamente iguais, um deles usando puramente Android e o outro usando Cordova.

Para minimizar a interferência do *framework* nos testes, o arquivo *html*, responsável por gerar a tela inicial da aplicação, foi mantido somente com duas caixas de texto: uma delas avisa o término da execução da aplicação e a outra é usada para realizar testes durante as adaptações dos *benchmarks*. Toda a lógica da aplicação nos *benchmarks* executados através do Cordova estava contida num arquivo JavaScript e a compilação da aplicação foi realizada através do Android Studio.

No caso dos aplicativos convertidos, foi adotada uma abordagem diferente da dos *benchmarks*. O aplicativo não foi convertido completamente para um aplicativo híbrido utilizando o Cordova. Isto se deve a dois motivos principais:

- Aplicativos criados através do Cordova geram sua interface gráfica não através dos componentes de atividades do Android, mas usando recursos naturais de desenvolvimento móvel. Para efetuar esta mudança, seria necessário bastante esforço por parte do desenvolvedor, tornando o trabalho de adaptar um aplicativo basicamente o trabalho de reconstruí-lo inteiro, o que não é viável.

- A ideia por trás das adaptações é mostrar que modificações simples podem trazer benefícios para a aplicação, desde que façam uso inteligente das vantagens de cada linguagem. Ao converter toda a aplicação através do Cordova, não seria possível

mostrar este fato.

3.1.1 *Modificações nos Benchmarks*

Para realizar as modificações, os *benchmarks* foram executados com o código original seguindo procedimento padrão, 10 vezes em cada linguagem. Caso uma implementações do *benchmark* tivesse uma diferença muito grande, i.e., o consumo do *benchmark* em uma das linguagens fosse mais do que cinco vezes o da outra, o algoritmo da linguagem que apresentava maior consumo era modificado para que ficasse o mais parecido possível com a implementação do *benchmark* mais eficiente, com a intenção de aumentar a eficiência energética da linguagem que estava apresentando maior consumo de energia. Pela similaridade semântica entre as duas linguagens, Java e JavaScript, o trabalho não era de alta complexidade.

Um exemplo simples de modificação ocorreu no caso do *benchmark* "sequence of non-squares". Apesar da modificação ter influenciado no resultado, a diferença ainda foi bastante relevante. Quando comparada às implementações retiradas do Rosetta Code, a implementação em Java consome em média 65,72x mais energia do que a implementação em JavaScript. A implementação adaptada, que faz com que a solução em Java rode uma implementação similar à apresentada no algoritmo em a solução em JavaScript, consume 56,71x mais energia do que JavaScript, o que resulta em uma economia de 15% de energia com a relação a solução original em Java. As modificações realizadas podem ser encontradas na Figura 3.1.

Todos os dados presentes neste trabalho levam em conta a melhor solução para um determinado *benchmark* ou aplicativo. Os resultados das soluções originais que foram considerados piores do que as soluções adaptadas foram descartados.

Outro exemplo é o caso do *benchmark* nQueens, que envolve colocar n rainhas em um tabuleiro de xadrez $N \times N$, onde $N > 3$. Nos testes, foram utilizadas 13 rainhas. O tempo de execução de Java foi de 20s e o de JavaScript 69s, com JavaScript consumindo 5,17x mais energia do que Java. Ao converter o *benchmark* em JavaScript para utilizar o mesmo algoritmo presente em Java, o tempo de execução em JavaScript caiu para 12s e Java passou a consumir 4.75x mais energia. Este é um caso interessante, pois demonstra que Java estava tendo um melhor desempenho inicialmente não por causa da linguagem *per se*, mas pela maneira ineficiente como havia sido construído o algoritmo em JavaScript. As imagens referentes a esta modificação podem ser encontradas no Apêndice .

Em outros casos foram feitas modificações nos *benchmarks* modificando o sistema de uso de arquivos, uma vez que os *benchmarks* foram feitos inicialmente levando em conta o sistema de arquivos de um computador. Estas modificações, que são muito mais extensas e provavelmente não influenciariam diretamente no consumo de energia e desempenho, foram omitidas deste trabalho. As modificações podem ser analisadas através do código fonte.

```java
//Java from Rosetta:
public static void execute(long number) {
    for (long i = 1; i < number; i++) {
    long args = i + (long)Math.round(Math.sqrt(i));
  }
}
```

```javascript
//JavaScript from Rosetta:
function seqnonsquares(number){
  for (var i = 1; i < number; i++) {
    var args = i + Math.floor(1/2 + Math.sqrt(i))
  }
}
```

```java
//Java modified:
public static void execute(long number) {
  for (long i = 1; i < number; i++) {
    long args = i + (long) Math.floor(1/2 + Math.sqrt(i));
  }
}
```

Figura 3.1: Exemplo de modificação no caso do *benchmark* "sequence of non-squares". O primeiro trecho é o original em Java. O segundo é o original em JavaScript. O terceiro é o código adaptado em Java, de acordo com a solução em JavaScript. O consumo de energia usando o terceiro trecho é 15% menor do que o primeiro, mas ainda é 56,71x mais lento que a solução em JavaScript.

3.2 Hibridização dos Aplicativos

Como é mostrado na Seção 4.2, os resultados sugerem que as duas abordagens de desenvolvimento têm *trade-offs* diferentes com relação a consumo de energia. Contudo, aplicações para Android são majoritariamente escritas usando somente Java, ignorando o impacto que esta abordagem pode ter no consumo de energia. Entretanto, é possível fazer uso de código Java através de uma função em JavaScript e vice-versa. Uma vez que Java é a linguagem predominante no desenvolvimento de aplicativos Android, pode ser possível economizar energia ao adaptar aplicativos em Java para fazer uso de funções em JavaScript, onde este uso possa apresentar um ganho de eficiência energética e vice-versa. O código das aplicações, bem como os dados desta pesquisa estão disponíveis em `http://nativeorweb.github.io/`. O maior obstáculo ao utilizar esta abordagem de desenvolvimento com duas linguagens é o custo adicional por invocações entre linguagens (GRIMMER et al., 2013).

Benchmarks não podem ser utilizados para responder a QP2, uma vez que aplicativos funcionam de uma maneira diferente (RATANAWORABHAN; LIVSHITS; ZORN, 2010). Foram utilizados três aplicativos reais, desenvolvidos em código aberto e coletados através do F-Droid, para responder esta questão de pesquisa. Os aplicativos selecionados aparecem na última linha da Tabela 3.1. Tendo como principal objetivo das modificações ganhar eficiência energética, os aplicativos foram selecionados com foco nos seguintes fatores:

1. Os aplicativos precisavam ser escritos completamente em Java

2. O aplicativo precisava fazer uso intensivo de CPU. Boa parte dos aplicativos faz uso intenso da tela ou sensores, sendo difícil avaliar o consumo de energia do seu uso padrão.

3. Os aplicativos precisavam ter uma função principal que fizesse boa parte do trabalho do aplicativo. Com boa parte da execução dependendo de uma única função, ao modificar esta única função estaria assegurado que não haveria alteração no comportamento do aplicativo e estaríamos medindo o consumo de energia de um uso padrão.

4. O aplicativo não poderia fazer uso de uma biblioteca nativa em Android para realizar o seu comportamento. Ao fazer grande uso de uma biblioteca padrão, o aplicativo pode mudar drasticamente o consumo de energia e desempenho conforme o sistema operacional seja atualizado. Para garantir que as modificações seriam válidas independentemente da versão do sistema instalada no aparelho e a viabilidade da modificação por parte do desenvolvedor, o método que seria modificado deveria ter sido feito por um desenvolvedor.

Ao modificar partes do código do aplicativo em Java para que ele use JavaScript, é importante determinar a frequência com que o trecho em Java irá invocar o trecho em JavaScript. Se a invocação for muito frequente, boa parte do tempo de execução e consumo de energia será dominado pelo custo adicional referente ao uso de invocações entre linguagens. Se estas invocações forem muito pouco frequentes, dependendo da natureza da aplicação, o funcionamento da aplicação pode ficar comprometido. Como exemplo, temos o aplicativo Tri Rose, ao tentar diminuir a frequência das chamadas, o desempenho da aplicação foi severamente prejudicado, reduzindo pela metade a quantidade de traços desenhados em um intervalo de tempo. Neste trabalho optamos por utilizar três tipos diferentes de abordagens com relação à frequência de invocações entre linguagens: ***Stepwise***, ***Batch*** e ***Export***.

No caso da abordagem ***Stepwise***, um método em Java é mapeado diretamente para uma função em JavaScript e toda vez que o método for chamado, o fluxo é redirecionado e a função em JavaScript é chamada no lugar do método.

No caso da abordagem ***Batch***, um método em Java é mapeado para uma função em JavaScript, agrupando várias chamadas deste método. Esta função retorna o conjunto de resultados de várias execuções do método convertido. Este conjunto de resultados é então utilizado em Java para atualizar os valores correspondentes ao uso do método em Java. Esta abordagem reduz o custo adicional de comunicação ao dividir parte do trabalho entre Java e JavaScript.

No caso da abordagem ***Export***, todo o trabalho é realizado em JavaScript, uma vez que uma sequência de métodos é agrupada e mapeada em uma única função. Esta abordagem procura minimizar o custo adicional de comunicação.

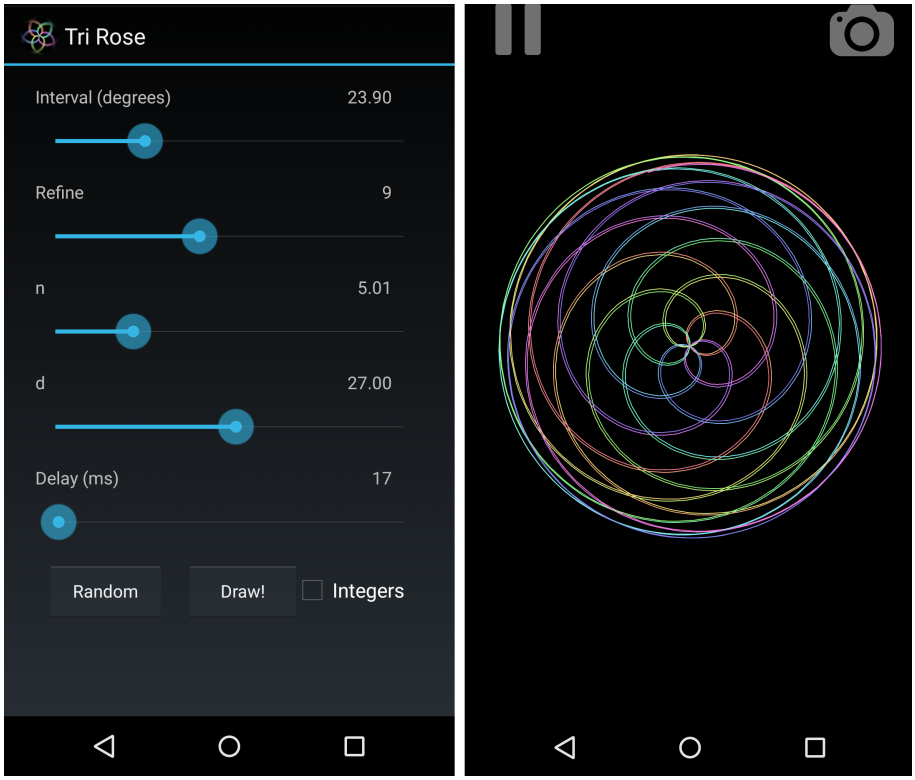

Figura 3.2: Telas do aplicativo Tri Rose. A tela da esquerda é onde o usuário pode selecionar os parâmetros para definir o desenho. Ao apertar o botão "Draw!", a aplicação muda para a tela da direita, onde o desenho é montado até o usuário decidir parar.

3.2.1 *Descrição dos aplicativos*

Os aplicativos apresentam comportamentos diferentes. A seguir vamos explicar quais foram as abordagens utilizadas em cada um dos aplicativos, o motivo dessas abordagens serem utilizadas, suas cargas de trabalho e a quantidade de linhas modificadas adicionadas ao código em JavaScript em cada aplicação.

Como no caso dos *benchmarks*, as cargas de trabalho foram definidas tentando manter o tempo de execução dentro de um limite preestabelecido. Tentou-se manter um mínimo de 30s de tempo de execução para que aplicação pudesse executar de forma regular e manter um desvio padrão baixo.

Tri Rose: Este aplicativo realiza desenhos na tela segundo fórmulas matemáticas e parâmetros estabelecidos pelo usuário. Neste aplicativo foram utilizadas as abordagens ***Stepwise*** e ***Batch***. Não foi possível utilizar a abordagem ***Export***, uma vez que desenhar na tela ocupa uma parte significante da execução e optou-se por utilizar o trecho do código em Java para realizar esses desenhos, não sendo viável a transferência da execução para JavaScript por completo, como já foi discutido na Seção 3.1.1. A carga de trabalho foi de 1.500 traços desenhados em cada uma das execuções usando sempre o mesmo conjunto de parâmetros. Na versão ***Batch***, um buffer de 11×10^4 posições foi utilizado para armazenar todos os dados. O buffer é utilizado

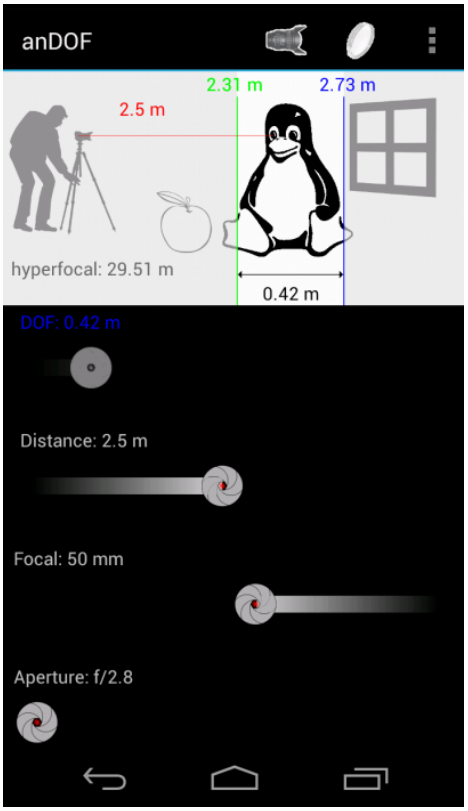

Figura 3.3: Tela do aplicativo anDOF. Modificações nas barras chamam um método que atualiza os valores no canto superior do aplicativo com base nos novos parâmetros.

tanto na versão ***Batch*** em Java quanto em JavaScript. O aplicativo tem cerca de 1kLoC e o arquivo contendo a função adaptada e todas as funções auxiliares em JavaScript tem 101 linhas de código, o que representa cerca de 10% da aplicação. A Figura 3.2 mostra as duas telas do aplicativo.

anDOF: Este aplicativo calcula a profundidade de campo usando parâmetros estabelecidos pelo usuário. Neste aplicativo foram usadas as abordagens ***Stepwise*** e ***Export***. Para simular a execução, o aplicativo foi executado, modificando a barra que altera os parâmetros necessários para realizar o cálculo, fazendo o aplicativo calcular a profundidade de campo. Como as execuções são feitas em tempo real e só era necessário um único valor para modificar os cálculos (o novo valor referente à barra), não foi utilizada a abordagem ***Batch***. A carga de trabalho foi de 3×10^3 valores diferentes de modificação na barra para a abordagem ***Stepwise*** e 18×10^6 valores diferentes de modificação da barra para a abordagem ***Export***. O aplicativo tem cerca de 1,7kLoC e o arquivo contendo a função adaptada e todas as funções auxiliares em JavaScript tem 166 linhas de código, o que representa cerca de 9,7% da aplicação. A Figura 3.3 mostra a tela do aplicativo.

EnigmAndroid: Este aplicativo procura simular o comportamento da máquina codificadora de mensagens Enigma, utilizada pelos nazistas na segunda guerra mundial. Neste aplicativo foi utilizada apenas a abordagem ***Export***. Uma vez que o aplicativo funciona perfeitamente

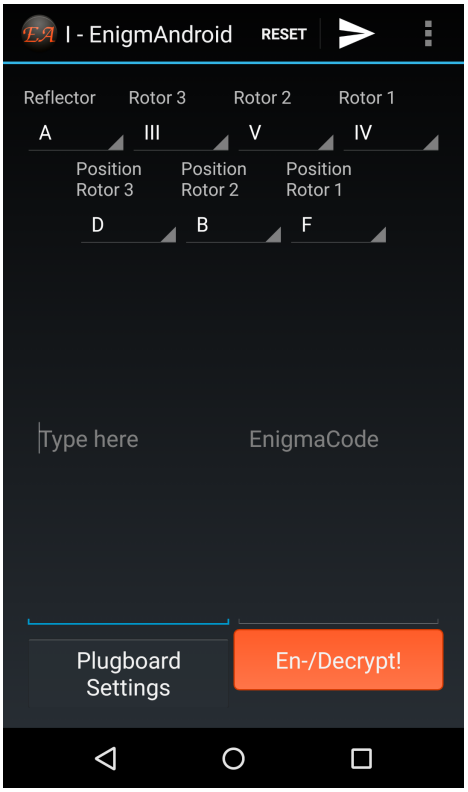

Figura 3.4: Tela do aplicativo EnigmAndroid. A parte superior representa os parâmetros da máquina. O usuário insere o texto a ser codificado na caixa de texto "Type here"e ao apertar em "En-/Decrypt!"o texto vai aparecer codificado na caixa de texto "EnigmaCode".

.

dentro do contexto do ***Export***, isto é, toda a execução é realizada por uma única função e esta retorna um único valor, a String inicial codificada, fazer adaptações do aplicativo para ***Stepwise*** ou ***Batch*** seria contra-intuitivo. A carga de trabalho utilizada foi uma única string padrão para todas as execuções. A String padrão era composta de frases aleatórias compostas por 60×10^3 caracteres. O aplicativo tem cerca de 4,6kLoC e o arquivo contendo a função adaptada e todas as funções auxiliares em JavaScript tem 173 linhas de código, o que representa cerca de 3% da aplicação. A Figura 3.4 mostra a tela do aplicativo.

3.3 Executando os experimentos

Para realizar as medições é necessário que o desenvolvedor tenha acesso ao Android Debug Bridge (ADB), que pode ser instalado junto ao Software Development Kit (SDK) do Android.

A Figura 3.5 mostra o comando utilizado no console do computador que dá ao desenvolvedor diversas informações sobre o uso de bateria do sistema, como a quantidade que cada aplicativo gastou de bateria, medido em mAh, e a quantidade de tempo que ele passou executando, desde sua última medição.

```
1       $ adb shell dumpsys batterystats --charged <package-name>
```

Figura 3.5: Comando utilizado para coletar dados através do Projeto Volta.

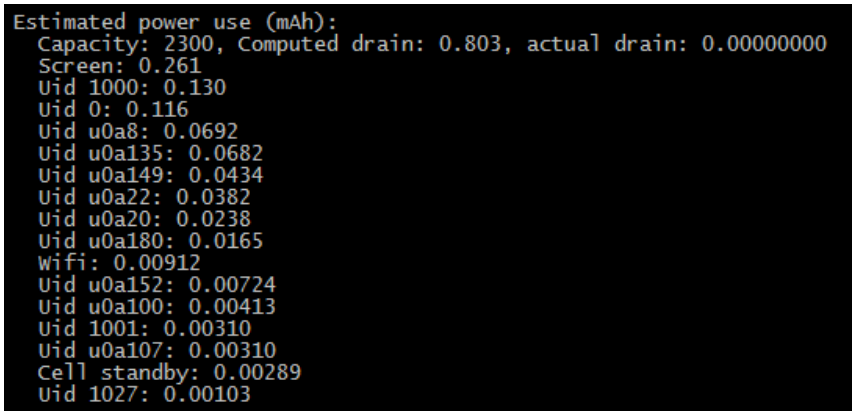

Figura 3.6: Como os dados de uso de bateria são exibidos através do Projeto Volta. Os aplicativos de usuário seguem o padrão "Uid u0aXX : Y"onde XX é o id da aplicação e Y é o consumo de bateria em mAh.

A Figura 3.6 ilustra a forma como os dados são apresentados. Os aplicativos são identificados através do seu número de ID. Como ilustrado na figura, aplicativos do usuário seguem o formato "Uid u0aXX"onde XX representa o ID do aplicativo. Os aplicativos recebem seus IDs de acordo com a ordem em que foram instalados no aparelho. Os aplicativos do sistema seguem o formato "Uid ####", onde #### representa o ID fixo do aplicativo. Recursos do sistema são identificados pelo seu nome (Screen, Wifi, Bluetooth).

A medida usada para medir o consumo de bateria é *miliampere*$\times$*hora*. Por não se tratar de uma medida de energia mas sim de transferência de energia, todos os dados foram convertidos. Foi considerado que a voltagem da bateria se manteve constante em 4 Volts, como será discutido na sessão de resultados. Todos os valores foram convertidos segundo a seguinte igualdade: 1aHV $=$ 3600Joules.

Para os testes, foi utilizado um aparelho Nexus 5 (2013), rodando Android 5.1, com 16GB de flash memory, 2GB de memória RAM, chipset Qualcomm MSM8974 Snapdragon 800, CPU Quad-core 2.3 GHz Krait 400 e bateria Li-Po 2300 mAh. Cada *benchmark* foi executado pelo menos 10 vezes para cada linguagem de programação, totalizando mais de 600 testes, com um tempo médio de 80s de execução. Todos os dados sobre consumo de energia e tempo de execução foram coletados utilizando os recursos do Projeto Volta. Todos os *benchmarks* e aplicativos vibravam por 1.5s e finalizavam sua própria execução ao terminar de executar a carga de trabalho.

Todos os experimentos foram executados observando os seguintes passos:

1. Verificar se a bateria tinha pelo menos 80% de sua carga, evitando que o aparelho entre em modo de economia de energia e mantendo a voltagem sempre em pelo menos 4V.

2. Fechar todos os aplicativos que não tivessem relação com os testes, ativar o modo avião e imediatamente reiniciar o aparelho. Este passo tenta isolar o comportamento da aplicação, impedindo que outros aplicativos continuem executando, tais como sensores Wi-Fi ou GPS que poderiam interferir nos resultados da medição.

3. Conectar o aparelho ao computador e limpar todos os dados relacionados a consumo de bateria. Desconectá-lo depois disso.

4. Executar a aplicação ou *benchmark*.

5. Impedir que o aplicativo execute no *background*, não travar a tela nem deixá-la apagar ou mudar para outro aplicativo. Ao travar a tela, o aplicativo obrigatoriamente rodaria no *background* e ao rodar no *background* ele não faz uso ótimo da CPU, enviesando o experimento.

6. Conectar o aparelho ao computador e verificar o consumo de bateria e tempo de execução.

4

Resultados do estudo

Este capítulo tem como objetivo detalhar a análise dos resultados coletados nos experimentos. O capítulo conta com uma seção de visão geral dos resultados (Seção 4.1), uma seção que busca responder a primeira questão de pesquisa (Seção 4.2), uma seção que busca responder a segunda questão de pesquisa (Seção 4.3), uma análise sobre os dados coletados (Seção 4.4) e uma seção de ameaças a validade da pesquisa (Seção 4.5).

4.1 Visão geral dos resultados

A princípio serão exibidos os dados relativos à execução dos *benchmarks* e dos aplicativos modificados. As Tabelas 4.1 e 4.2 contêm os dados relativos aos experimentos com aplicativos modificados e as Tabelas 4.3 e 4.4 os dados relativos aos experimentos com *benchmarks*.

Algumas tabelas apresentam a coluna *Workload*, que representa a carga de trabalho do *benchmark* ou aplicativo. No caso dos *benchmarks*, cada um deles pode apresentar um tipo diferente de entrada. O dado consta na tabela como forma de permitir a replicação do experimento. No caso dos aplicativos, o que cada valor na coluna *Workload* representa foi detalhado na Seção 3.1.1.

Na Tabela 4.1 há um resumo da execução dos aplicativos. Esta tabela tem como foco principal as duas últimas colunas, $\frac{Java}{JS}$ tempo e $\frac{Java}{JS}$ energia, que apresentam a relação de desempenho e consumo de energia das duas linguagens. É facilmente visto através da tabela que com a abordagem ***Export*** é possível obter um desempenho e eficiência energética muito superior às outras abordagens. A abordagem ***Batch*** ainda apresenta melhora relevante com relação à energia, mas não com relação ao tempo. ***Stepwise*** é uma abordagem que provavelmente não deve ser realizada, uma vez que não encontramos casos onde ela fosse benéfica para o desempenho ou para o consumo de energia da aplicação.

Na Tabela 4.2 temos dois conjuntos de dados relevantes: As linhas de código e as médias relacionadas à execução do teste. Além disso, uma coluna identifica as linhas de código totais do aplicativo e as linhas de código adicionadas para efetuar as modificações (LoC ; LoC+). É possível notar que a quantidade de linhas adicionadas é bastante pequena em relação ao

Tabela 4.1: Dados referentes à execução dos aplicativos modificados. Nesta tabela são sumarizadas a carga de trabalho (*Workload*), a razão entre o tempo de execução de Java sobre o tempo de execução de JavaScript ($\frac{Java}{JS}$ tempo) e a razão entre o consumo de energia de Java sobre o consumo de energia de JavaScript($\frac{Java}{JS}$ energia). Nos dois últimos casos, se a relação $\frac{Java}{JS}$ for menor do que 1, ela é marcada de negrito.

Aplicativo	Abordagem	Workload	$\frac{Java}{JS}$ tempo	$\frac{Java}{JS}$ energia
anDof	Stepwise	3×10^3	1.00	**0.79**
	Export	18×10^6	19.29	28.08
TriRose	Stepwise	1,500	**0.41**	**0.54**
	Batch	1,500	1.01	1.30
EnigmAndroid	Export	60×10^3	2.57	2.96

total da aplicação, evidenciando que até pequenas modificações em trechos que são muito utilizados podem trazer uma mudança não negligenciável ao consumo de energia e desempenho da aplicação. Outras 3 colunas sintetizam às médias das execuções dos aplicativos: tempo de execução (tExec), tempo de CPU (tCPU) e o consumo de energia da aplicação durante a realização do teste (Energia). Há também uma coluna auxiliar com o desvio padrão das execuções em relação a energia (σ Energia). É possível notar que o desvio padrão é relativamente baixo para todas as execuções, com exceção da execução do TriRose usando a abordagem ***Batch*** em JavaScript. Como será explicado mais adiante, o desvio padrão costuma ser menor para as execuções em JavaScript, tornando este caso atípico.

Nas Tabelas 4.3 e 4.4 os dados são muito similares, uma vez que eles são referentes a *benchmarks* que lidam majoritariamente com problemas que fazem uso intenso de CPU. Na Tabela 4.4, há dois *benchmarks* que foram testados com sua implementação otimizada para um único núcleo e para vários núcleos. Eles são marcados respectivamente com SC, referente a *singlecore*, e MC, referente a *multicore*, depois do nome do *benchmark*. Assim como nos caso dos aplicativos, os dados mais relevantes estão nas duas últimas colunas, $\frac{Java}{JS}$ tempo e $\frac{Java}{JS}$ energia, onde consta a relação de desempenho e consumo de energia das duas linguagens. É possível notar também que o desvio padrão costuma ficar abaixo dos 18% e somente 5 das 66 execuções tiveram um desvio padrão maior. Cabe enfatizar na Tabela 4.3 os *benchmarks* **SeqNonSquares**, como caso em que a execução em JavaScript tem menor consumo de energia, com a execução em Java consumindo 36,27x mais energia, e **ShellSort**, como caso em que a execução em Java tem menor consumo de energia, com a execução em JavaScript consumindo 42% mais energia. Na Tabela 4.4, destacamos o *benchmark* **RegexDna MC** que consome 23,9x mais energia na execução em Java e **RevComp** que consome 2,27x mais energia em JavaScript.

Tabela 4.2: Dados referentes às médias dos testes com os aplicativos modificados. As linhas de código total do aplicativo e as linhas adicionadas para modificá-lo (LoC ; LoC+), as médias do tempo de execução (tExec), tempo de CPU (tCPU), consumo de energia (Energia) e o desvio padrão referente a energia (σ Energia).

Aplicativo	Abordagem	LoC ; Loc+	Linguagem	tExec (s)	tCPU (s)	Energia (J)	σ Energia
anDof	Stepwise	1700 ; 166	Java	33.76	10.72	13.08	8.67%
			JavaScript	33.78	14.21	16.64	15.60%
	Export		Java	632.81	672.49	715.68	2.38%
			JavaScript	32.80	30.19	25.49	12.99%
TriRose	Stepwise	1000 ; 101	Java	30.73	14.07	10.32	10.10%
			JavaScript	74.52	36.46	19.21	5.23%
	Batch		Java	34.33	13.37	9.60	10.36%
			JavaScript	34.09	10.77	7.40	24.32%
EnigmAndroid	Export	4661 ; 173	Java	89.56	98.51	79.96	5.17%
			JavaScript	34.89	31.49	26.97	17.93%

4.2 Há um modelo de desenvolvimento que seja mais eficiente energeticamente?

Nos nossos experimentos, as versões de JavaScript da maioria dos *benchmarks* consumiu menos energia (82% dos casos) e obteve um melhor desempenho (64% dos casos). Contudo, este resultado não é universal e, em alguns casos, as versões em Java dos *benchmarks* tiveram um melhor resultado, tanto em desempenho (36% dos casos) quanto em consumo de energia (18% dos casos).

A Figura 4.1 mostra o tempo de execução (linhas) e a energia consumida (barras) dos *benchmarks* originados do Rosetta Code. De forma geral, os *benchmarks* em JavaScript exibiram um consumo menor de energia e tempo de execução. As versões em Java destes *benchmarks* consumiram uma mediana de 2,09x mais energia que as versões em JavaScript. As versões em Java terminaram uma mediana 1,52x mais tempo para terminar a execução. A figura mostra que em 18 dos 22 *benchmarks* do Rosetta, as versões em JavaScript consumiram menos energia e em 16 dos 22, tiveram um tempo de execução menor.

Os resultados do *benchmark* "sequence of non-squares"não estão presentes na figura 4.1. Este *benchmark* usa a fórmula $n + \lfloor \frac{1}{2} + \sqrt{n} \rfloor$ para dar a sequência dos números naturais não-quadrados. O valor n utilizado no teste foi de 35×10^8. Ele foi omitido do gráfico para melhorar a legibilidade do mesmo. O tempo médio de execução e consumo médio em Java foram 538s e 378.14J e em JavaScript foram 16.83s e 10.43J, respectivamente. Apesar desse resultado ter sido omitido do gráfico, ele foi considerado em todos os cálculos de consolidação de resultados junto aos outros *benchmarks*. As tabelas relativas aos testes e o gráfico com todos os *benchmarks* pode ser encontrado no Apêndice e Apêndice . Finalmente, em 3 dos 7 *benchmarks* onde Java foi mais rápido, ele também teve um consumo maior de energia que JavaScript, o que sugere uma relação não linear entre desempenho e consumo de energia.

Ao se comparar os resultados deste trabalho com os encontrados em sites que contêm os

Tabela 4.3: Dados referentes às médias do testes de *benchmarks* do Rosetta Code. Os dados são referentes às médias do tempo de execução (tExec), tempo de CPU (tCPU), consumo de energia (Energia) e o desvio padrão referente à energia, (σ Energia) e as razões entre tempo de execução e consumo de energia de Java sobre JavaScript. Nos casos em que a relação $\frac{Java}{JS}$ for menor do que 1, ela é marcada de negrito.

Benchmark	Workload	Linguagem	tExec(s)	tCPU(s)	Energia(J)	σEnergia	$\frac{Java}{JS}$ t	$\frac{Java}{JS}$ e
Matrix Mult	650x650	Java	53.65	48.64	41.76	17.89%	2.75	3.63
		JavaScript	19.51	13.08	11.51	14.75%		
Sieve of Eratosthenes	80×10^6	Java	70.12	69.24	83.10	8.32%	1.52	2.18
		JavaScript	46.06	40.05	38.19	15.95%		
Tower of Hanoi	32	Java	91.16	81.72	82.43	11.39%	**0.94**	1.04
		JavaScript	97.39	91.59	79.29	13.98%		
Happy Numbers	50×10^4	Java	62.72	58.46	68.04	7.32%	**0.94**	1.48
		JavaScript	66.50	60.88	46.02	13.74%		
Combinations	(75,30)	Java	56.03	49.89	45.89	15.46%	**0.75**	**0.86**
		JavaScript	74.99	68.89	53.31	11.05%		
Count in factors	32×10^4	Java	73.49	67.64	62.81	12.87%	**0.80**	**0.92**
		JavaScript	92.09	86.28	67.92	13.21%		
Knapsack Unbounded	12	Java	57.30	50.88	39.64	12.39%	4.18	5.38
		JavaScript	13.70	7.39	7.37	17.74%		
Zero-One Knapsack	10	Java	46.51	64.70	76.36	5.31%	2.35	5.90
		JavaScript	19.78	13.15	12.94	16.43%		
Knapsack Bounded	13	Java	63.24	81.16	89.50	5.15%	1.00	1.48
		JavaScript	63.46	58.16	60.34	10.45%		
Man or Boy	35×10^3	Java	96.32	101.59	102.07	5.88%	1.43	2.12
		JavaScript	67.23	61.30	48.11	9.09%		
NQueens	(4,13)	Java	19.29	13.60	10.94	14.69%	1.54	4.80
		JavaScript	12.53	2.51	2.28	29.98%		
Perfect Number	15×10^5	Java	163.56	156.93	147.95	15.46%	7.87	10.06
		JavaScript	20.79	16.09	14.71	13.51%		
SeqNonSquares	35×10^8	Java	538.01	536.77	378.14	2.28%	31.98	36.27
		JavaScript	16.83	11.56	10.43	12.78%		
BubbleSort	35×10^3	Java	101.41	88.71	76.61	16.97%	2.36	2.16
		JavaScript	43.00	39.93	35.50	13.22%		
CountingSort	45×10^6	Java	37.67	29.18	26.97	13.90%	1.90	1.97
		JavaScript	19.82	14.37	13.70	14.19%		
GnomeSort	18×10^4	Java	151.13	140.13	118.04	14.11%	2.45	2.85
		JavaScript	61.59	56.12	41.39	8.43%		
HeapSort	25×10^6	Java	64.17	58.15	44.91	10.28%	1.22	1.14
		JavaScript	52.58	47.18	39.40	13.55%		
InsertSort	22×10^4	Java	92.04	86.44	69.55	11.13%	1.08	1.10
		JavaScript	85.35	79.95	63.01	12.52%		
MergeSort	50×10^4	Java	53.87	57.34	62.61	7.08%	2.00	3.50
		JavaScript	26.96	20.27	17.87	11.87%		
PancakeSort	50×10^3	Java	43.88	35.66	29.32	15.32%	1.78	1.52
		JavaScript	24.71	22.57	19.32	14.63%		
QuickSort	18×10^5	Java	58.87	84.20	90.30	9.46%	**0.83**	1.62
		JavaScript	71.26	64.53	55.79	14.12%		
ShellSort	40×10^6	Java	55.17	50.51	43.29	14.40%	**0.71**	**0.70**
		JavaScript	77.81	71.98	62.18	14.21%		
HofstadterQ	25×10^5	Java	81.94	75.98	78.64	15.97%	4.95	8.93
		JavaScript	16.55	10.52	8.81	23.61%		

Tabela 4.4: Dados referentes às médias do testes de *benchmarks* do The Computer Language Benchmark Game. Os dados são referentes às médias do tempo de execução (tExec), tempo de CPU (tCPU), consumo de energia (Energia), o desvio padrão referente à energia (σ Energia), e as razões entre tempo de execução e consumo de energia de Java sobre JavaScript. Nos dois últimos casos, se a relação $\frac{Java}{JS}$ for menor do que 1, ela é marcada de negrito.

Benchmark	**Workload**	**Linguagem**	**tExec(s)**	**tCPU(s)**	**Energia(J)**	**σ Energia**	$\frac{Java}{JS}$ t	$\frac{Java}{JS}$ e
BinaryTree	19	Java	65.01	77.63	76.71	3.08%	1.55	2.39
		JavaScript	42.06	37.17	32.07	11.11%		
Fannkuch	12	Java	104.59	371.14	279.79	2.96%	**0.71**	2.67
		JavaScript	146.98	140.95	104.93	11.83%		
Fasta SC	50×10^6	Java	36.75	30.69	32.43	19.19%	**0.57**	**0.56**
		JavaScript	64.08	56.80	58.16	17.68%		
Fasta MC	50×10^6	Java	28.79	37.65	35.04	11.99%	**0.45**	**0.60**
		JavaScript	64.08	56.80	58.16	17.68%		
Nbody	50×10^6	Java	81.66	74.24	60.74	12.27%	1.40	1.41
		JavaScript	58.23	51.82	43.20	16.54%		
Spectral	50×10^6	Java	45.25	154.50	122.03	6.85%	**0.50**	1.69
		JavaScript	90.75	85.77	72.27	16.69%		
RegexDna SC	5×10^6	Java	145.80	139.27	120.30	10.25%	14.18	22.09
		JavaScript	10.28	5.18	5.44	27.99%		
RegexDna MC	5×10^6	Java	109.18	143.19	130.12	6.24%	10.62	23.90
		JavaScript	10.28	5.18	5.44	27.99%		
RevComp	10×10^6	Java	62.79	63.15	59.72	4.44%	**0.37**	**0.44**
		JavaScript	171.52	162.29	136.09	6.08%		
Knucleotide	5×10^6	Java	58.29	190.95	144.576	2.20%	**0.63**	1.96
		JavaScript	92.68	86.64	73.71	14.11%		

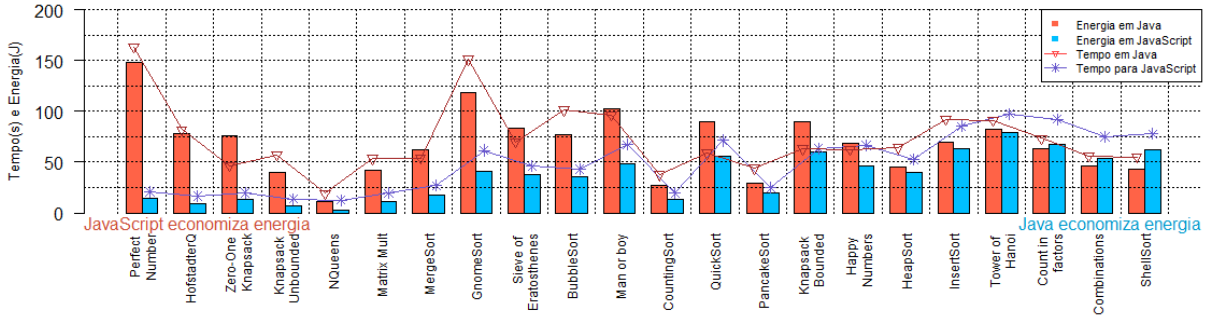

Figura 4.1: Resultados dos *benchmarks* do Rosetta Code. Um dos resultados não está no gráfico; o *benchmark* "sequence of non-squares". O tempo médio e o consumo médio deste *benchmark* foi de 16s e 10J em JavaScript e 670s e 570J para Java. As barras são organizadas de acordo com o ganho relativo em consumo de energia segundo a fórmula (*Consumo de energia em Java* ÷ *Consumo de energia em JavaScript*) para cada um dos benchmarks.

benchmarks selecionados, percebe-se uma disparidade nos dados de desempenho e eficiência energética. Isto pode indicar que os dados advindos de comparações de desempenho em desktops podem não ter correspondência significativa com dados extraídos de experimentos similares em dispositivos móveis.

Para validar os dados coletados dos *benchmarks* do TCLGB, buscando evitar enviesamento dos dados, um segundo aparelho foi utilizado. As especificações e demais detalhes deste aparelho serão discutidos na Seção 4.5.

A Figura 4.2 representa os resultados dos *benchmarks* do TCLBG. As versões de Java destes *benchmarks* consumiram 1,82x mais energia (mediana) do que as versões em Javascript. Esta figura mostra que 7 de 10 dos *benchmarks* consumiram menos energia em JavaScript. Entretanto, de forma diferente dos *benchmarks* do Rosetta Code, a mediana dos tempos de execução em Java para estes *benchmarks* é de 0,67x o tempo da execução em JavaScript. A principal razão para este comportamento é o uso de paralelismo. Todas as versões de JavaScript são executadas de forma sequencial, enquanto 8 dos 10 *benchmarks* em Java fazem uso de programação paralela para melhorar o seu desempenho. 5 de 10 *benchmarks* tem um desempenho superior em Java quando comparados a JavaScript, mas somente 3 deles exibem um consumo de energia menor. Este comportamento é consistente com trabalhos anteriores (PINTO; CASTOR; LIU, 2014), adicionando evidências substanciais de que para programas capazes de se beneficiar com processadores de múltiplos núcleos, melhora de performance não necessariamente indica uma redução no consumo de energia.

Analisando todos os *benchmarks* do Rosetta e TCLGB, nos casos onde JavaScript obteve um resultado superior, Java consumiu uma mediana de 2,28x mais energia e suas execuções levaram uma mediana de 1,59x mais tempo do que as versões em JavaScript. Quando Java obteve um desempenho melhor, JavaScript consumiu uma mediana de 1,40x mais energia e uma mediana de 1,40x mais tempo.

Através de uma análise do comportamento dos aplicativos, foi observado que *benchmarks*

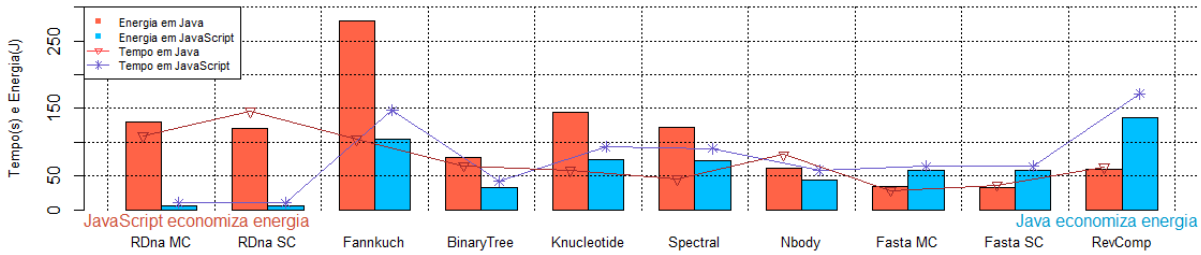

Figura 4.2: Resultados dos *benchmarks* do The Computer Language Benchmark Game (TCLBG). As barras são organizadas de acordo com o ganho relativo em consumo de energia segundo a fórmula (*Consumo de energia em Java ÷ Consumo de energia em JavaScript*) para cada um dos benchmarks.

que dependiam intensamente da CPU ou faziam milhares de operações aritméticas (*SeqNonSquares, nQueens, Perfect Number*)foram os que obtiveram um maior ganho de desempenho e eficiência energética, favorecendo JavaScript em detrimento de Java. Contudo, é fácil fazer uso de paralelismo em Java, o que contribui positivamente para o desempenho dos *benchmarks* escritos em Java. Apesar desta facilidade, o ganho em desempenho não parece reduzir de forma geral o consumo de energia, por vezes até aumentando o mesmo quando comparado a versões otimizadas para um único núcleo. Finalmente, para os *benchmarks* que faziam uso intenso de memória (*Knapsack Bounded, Zero-One Knapsack, HofstadterQ*) ou arquivos (*Knucleotide,RevComp, RegexDna*) não foi identificada nenhuma vantagem para nenhuma das duas linguagens.

Sintetizando os resultados da análise dos benchmarks, JavaScript obteve desempenho e eficiência energética superiores na maioria dos casos. JavaScript consumiu menos energia em 27 dos 33 *benchmarks* analisados, com Java gastando uma mediana de 97% mais energia no conjunto total de *benchmarks*. JavaScript também teve um desempenho melhor em 21 dos 33 *benchmarks* analisados, com Java gastando uma mediana de 43% mais energia no conjunto total de *benchmarks*.

Para aplicações que fazem uso intenso de CPU e operações aritméticas, como é o caso de uma parte considerável dos *benchmarks*, JavaScript tem um desempenho superior ao de Java. Para aplicações que seguem este perfil, o modelo de desenvolvimento usando linguagem nativa pode incorrer em um consumo maior de energia.

4.3 É possível reduzir o consumo de energia de um aplicativo ao torná-lo híbrido?

Através dos testes, identificamos que a hibridização pode ser benéfica para o aplicativo dependendo do seu grau de acoplamento. Nesta seção vamos examinar se é possível modificar os aplicativos de forma que o ganho em eficiência energética supere o custo adicional das invocações.

A Figura 4.3 mostra os resultados da análise das modificações nos aplicativos. Em

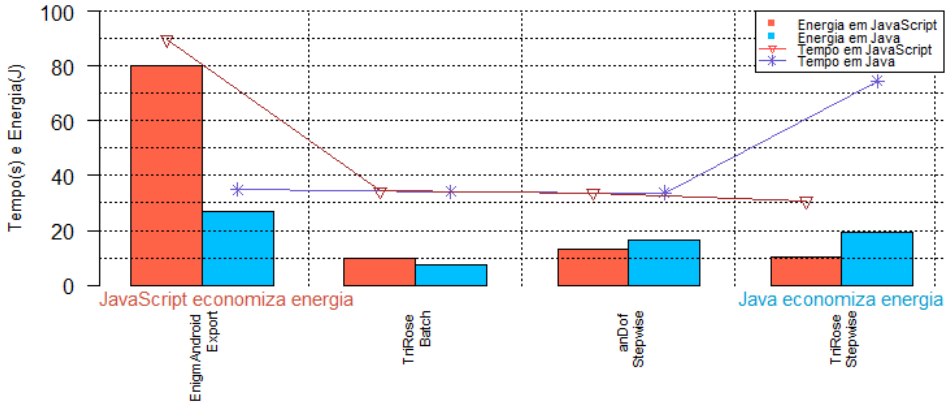

Figura 4.3: Resultados dos aplicativos modificados. As três abordagens diferem na adaptação do método em Java para JavaScript. ***Export***: executa o método várias vezes e retorna um único resultado. ***Batch***: executa o método várias vezes e retorna um conjunto de resultados. ***Stepwise***: executa o método uma única vez, diversas vezes, retornando o resultado cada uma das vezes. O abordagem ***Export*** do aplicativo anDOF foi omitido pela legibilidade. As barras são organizadas de acordo com o ganho relativo em consumo de energia segundo a fórmula (*Consumo de energia em Java* ÷ *Consumo de energia em JavaScript*) para cada um dos benchmarks.

todos os casos em que foram aplicadas modificações na abordagem ***Stepwise***, houve perda de performance e o consumo de energia elevou-se. Este resultado pode indicar que o custo adicional de trabalho gerado por milhares de requisições a JavaScript aumenta o consumo a um ponto onde não é possível reverter a situação com um possível ganho de energia pela execução ser realizada em JavaScript. Caso não seja possível agrupar um conjunto de requisições ou até mesmo realizar todo o trabalho computacional em JavaScript, aplicações híbridas podem não gerar nenhuma melhora de eficiência energética. Nos nossos experimentos, todos os casos usando ***Stepwise***, a versão original em Java teve melhor desempenho e um consumo de energia menor.

Usando a abordagem ***Batch***, com novas aplicações criadas em Java, usando as originais, modificando-as para fazer uso de um sistema de ***Batch*** assim como a nova versão em JavaScript, as aplicações em Java usando ***Batch*** consumiram 30% mais energia que as aplicações usando código JavaScript e 40% quando comparadas à versão ***Batch*** híbrida com a versão original em Java. Um detalhe sobre o caso do Tri-Rose é que, apesar da computação dos pontos ser realizada em JavaScript, boa parte da execução é feita desenhando as curvas na tela, usando código Java. Para a abordagem ***Batch***, JavaScript apresentou um consumo menor de energia, mas Java teve um desempenho melhor.

Nos casos utilizando ***Export***, o caso do anDOF representa um exemplo não realista de uso do aplicativo, uma vez que a execução é feita sem o toque do usuário na barra que controla o parâmetro, mas representa um caso onde é possível fazer uso de um loop para executar boa parte do comportamento do aplicativo, diluindo o custo de invocações entre linguagens. O caso do EnigmAndroid é perfeitamente realista e não altera em nada o comportamento da aplicação quando comparado com a aplicação original. Os resultados desta abordagem indicam que quando

a aplicação realizar uma grande quantidade de cálculos, o uso de JavaScript pode resultar em ganhos significativos para o desempenho e eficiência energética da aplicação. Em todos os casos, a aplicação híbrida utilizando a abordagem ***Export*** teve um desempenho superior e um consumo de energia menor.

Apesar das modificações levarem a melhoras significativas na eficiência energética dos aplicativos, elas não foram modificações em larga escala. Os arquivos JavaScript para cada um dos aplicativos continham menos de 10% do total de linhas de código da aplicação, e as modificações representaram apenas 3% do código no caso do EnigmAndroid (173 linhas adicionadas e 4660 linhas na aplicação).

Através da análise dos dados, identificamos que é possível usar JavaScript, transformando um aplicativo nativo em híbrido, para obter ganhos tanto em desempenho quanto em eficiência energética em casos específicos. Para que seja possível obter benefícios da hibridização é necessário que sejam usadas abordagens similares a ***Batch*** e ***Export***, tentando minimizar o *overhead* de comunicação entre as linguagens. Abordagens que façam muito uso da comunicação entre Java e JavaScript, como é o caso da abordagem ***Stepwise***, podem acarretar no prejuízo do desempenho e eficiência energética.

4.4 Análise dos dados

Como forma de se tentar obter uma observação geral sobre o comportamento de aplicativos em ambas as linguagens e em todos os cenários analisados i.e., *benchmarks* do Rosetta Code, *benchmarks* do TCLBG e aplicativos *open-source*, foram estabelecidas certas relações entre os dados coletados. Vale salientar que o perfil dos aplicativos é diferente dos *benchmarks*, tendo um padrão distinto de uso, uma vez que boa parte de sua execução conta com elementos de IO, como desenhar na tela. Isto se torna aparente na grande diferença entre os valores das relações quando se compara o cenário dos aplicativos ao dos *benchmarks*.

Tempo de CPU ÷ Tempo de execução : Na Tabela 4.5 estão os dados referentes a esta relação. Ela busca verificar o quanto a aplicação faz uso da CPU e representa esse uso em valores percentuais. É possível notar que, apesar de ser um dado que representa o percentual de uso de CPU dentro de uma aplicação, ele pode ser maior do que 1 no caso de uso de paralelismo. Os resultados variam enormemente mesmo dentro de um mesmo cenário, chegando a ter um desvio padrão relativo de 64,36% no caso dos *benchmarks* do TCLGB em Java. Intuitivamente, as aplicações gastam muito menos tempo fazendo uso da CPU em relação ao seu tempo de execução, levando a uma diferença significativa entre os valores dos *benchmarks* e dos aplicativos (Rosetta = 91% ; TCLBG = 95% ; Aplicativos = 44%). O tempo que uma aplicação passa usando a CPU influencia diretamente no seu consumo de energia. Ao se comparar aplicações com alto percentual de uso de CPU com seu tempo de execução, as mesmas também apresentavam uma grande quantidade de gasto de energia por tempo de execução.

Energia ÷ Tempo de execução (J÷s ou W): Na Tabela 4.6 constam os dados referentes

Tabela 4.5: Dados referentes a divisão do tempo gasto utilizando a CPU e o tempo de execução do *benchmark* ou aplicativo.

Tempo de CPU ÷ Tempo de execução				
		Rosetta Code	**TCLBG**	**Aplicativos**
Java	Media:	0.97	1.78	0.67
	Mediana:	0.93	1.25	0.46
	Desvio Padrão:	18.12%	64.36%	57.56%
JavaScript	Media:	0.80	0.83	0.61
	Mediana:	0.91	0.89	0.49
	Desvio Padrão:	22.28%	21.13%	46.29%
Total	Media:	0.89	1.30	0.64
	Mediana:	0.91	0.95	0.47
	Desvio Padrão:	22.02%	71.35%	49.96%

a esta relação. Ela busca verificar se o consumo de energia varia de forma consistente de acordo com o tempo levado para um aplicativo terminar a sua execução. Assim como no caso da relação entre tempo de uso de CPU e tempo de execução, há uma grande variação dentro de um mesmo cenário, fazendo com que o desvio padrão relativo chegue a 63,17% no caso dos aplicativos modificados em Java. O consumo relacionado à mediana de JavaScript é inferior ao de Java no caso dos *benchmarks*, tanto para o Rosetta (Java = 0,82W, JavaScript = 0,74W) como para o TCLBG (Java = 1,19W, JavaScript = 0,78W), mas apresenta uma mediana superior no caso dos aplicativos (Java = 0,39W, JavaScript = 0,49W). A relação de *Energia ÷ Tempo de execução* apresenta dados muito similares aos de *Tempo de uso de CPU ÷ Tempo de execução*. Por esse motivo, dividindo essas duas relações para obter a razão *Energia ÷ Tempo de CPU*, chega-se a um número estável. Este resultado seria esperado no caso dos *benchmarks*, que apresentam um perfil de uso similar e intensos em CPU, e estudos anteriores (TIWARI; MALIK; WOLFE, 1994) mostram que o consumo de energia está atrelado ao uso de CPU, para aplicações que são limitadas por CPU. Vale a pena lembrar que PATHAK; HU; ZHANG (2012) identificaram que uma parte considerável da energia em aplicativos é gasta com operações de entrada e saída. Pela quantidade de uso de instruções de entrada e saída, o cenário poderia ser diferente no caso dos aplicativos. A análise das medianas do total de *benchmarks* nas duas linguagens aponta os resultados de mediana dos *benchmarks* do Rosetta como 0,88W, os *benchmarks* do TCLBG como 0,86W, e o dos aplicativos como 0,83W. Este resultado leva a crer que o consumo de energia relacionado a CPU é um valor que tende a variar pouco e independentemente da aplicação, se aproximando de 0,85W. Para validar esta hipótese, são necessários mais experimentos com diferentes tipos de aplicativos e aparelhos.

Energia × Tempo de execução (J×s ou EDP): Na Tabela 4.7 constam os dados referentes a esta relação. Uma vez que ambas as grandezas, tempo e energia, são relevantes e temos interesse em que ambas sejam o menor possível, o EDP dá uma ideia intuitiva sobre o *trade-off* entre energia e tempo quando comparando diferentes abordagens. Levando em consideração o desvio padrão encontrado, chegando a 288,94% no Rosetta em Java, a análise

Tabela 4.6: Dados referentes à divisão da energia pelo tempo de execução. Os dados estão em J÷s ou watt.

Energia ÷ Tempo de execução (J÷s ou W)				
		Rosetta Code	**TCLBG**	**Aplicativos**
Java	Media:	0.94	1.48	0.61
	Mediana:	0.82	1.19	0.39
	Desvio Padrão:	30.94%	53.89%	63.17%
JavaScript	Media:	0.74	0.75	0.50
	Mediana:	0.74	0.78	0.49
	Desvio Padrão:	26.82%	17.52%	53.47%
Total	Media:	0.84	1.12	0.55
	Mediana:	0.78	0.85	0.44
	Desvio Padrão:	31.59%	60.37%	57.06%

Tabela 4.7: Dados referentes a multiplicação da energia pelo tempo de execução. Os dados estão em J×s também conhecido como EDP.

Energia × Tempo de execução (J×s)				
		Rosetta Code	**TCLBG**	**Aplicativos**
Java	Media:	14387.81	9085.76	92228.29
	Mediana:	4616.21	5254.39	441.62
	Desvio Padrão:	288.94%	97.61%	218.63%
JavaScript	Media:	2289.74	6358.76	804.63
	Mediana:	1759.01	3727.16	836.10
	Desvio Padrão:	99.32%	117.69%	54.76%
Total	Media:	8338.77	7722.26	46,516.46
	Mediana:	3147.58	4973.62	699.08
	Desvio Padrão:	356.72%	105.00%	306.99%

das medianas é mais adequada. A análise mostra que Java costuma ter um valor mais alto de EDP. O único caso em que Java apresenta uma mediana inferior a JavaScript é no caso dos aplicativos, apesar da média ser duas ordens de grandeza maior. Isto se deve aos casos que fazem uso da abordagem Export, que apesar de serem poucos, apresentam um grande ganho de desempenho para JavaScript. A Tabela 4.8 mostra os casos específicos onde os aplicativos tem melhor desempenho e consomem mais energia em Java. Dos 6 *benchmarks*, somente 2 tiveram um EDP inferior em Java (*Spectral, Tower of Hanoi*).

De forma geral, no caso dos *benchmarks*, JavaScript apresenta resultados melhores, tanto para a média quanto para a mediana, em relação às duas relações analisadas (*Energia ÷ Tempo de execução* e *Energia × Tempo de execução*), o que está de acordo com os dados encontrados nesta pesquisa uma vez que os *benchmarks* são fortemente vinculados ao uso de CPU. Analisando a mediana, Java apresenta dados melhores para os aplicativos, que fazem menos uso de CPU. Os dados coletados em JavaScript apresentam também um desvio padrão sempre inferior aos dados de Java, o que pode representar uma estabilidade maior no consumo de energia e desempenho ao utilizar aplicações que façam uso de JavaScript como linguagem principal.

Tabela 4.8: Benchmarks que apresentam melhor desempenho e pior eficiência energia em Java. Os 3 primeiros *benchmarks* são do TCLBG e os 3 últimos do Rosetta Code. As duas últimas colunas mostram a relação de energia dividido pelo tempo de execução (e ÷ t) e energia multiplicada pelo tempo de execução (e × t).

Benchmark	**Linguagem**	$\frac{Java}{JS}$ **tempo**	$\frac{Java}{JS}$ **energia**	**e ÷ t (J÷s)**	**e × t (J×s)**
Fannkuch	Java	0.71	2.67	2.68	29264.45
	JavaScript			0.71	15423.51
Spectral	Java	0.50	1.69	2.70	5521.68
	JavaScript			0.80	6558.80
Knucleotide	Java	0.63	1.96	2.48	8427.34
	JavaScript			0.80	6831.86
QuickSort	Java	0.83	1.62	1.53	5315.72
	JavaScript			0.78	3975.42
Tower of Hanoi	Java	0.94	1.04	0.90	7513.79
	JavaScript			0.81	7721.79
Happy Numbers	Java	0.94	1.48	1.08	4267.57
	JavaScript			0.69	3060.68

4.5 Ameaças à validade

Para minimizar os riscos de utilizar um aparelho com defeito e prejudicar os resultados desta pesquisa, um segundo aparelho com especificações similares, mas manufaturado por outra empresa e rodando uma versão diferente de Android, foi utilizado para executar os *benchmarks* do TCLBG. Os valores brutos de tempo e consumo de energia foram similares e a relação entre as linguagens, que linguagem obteve melhor desempenho e eficiência energética, se mantiveram. Este dados podem ser encontrados no Apêndice . O segundo aparelho foi um Moto G (2015) rodando Android 5.1.1, com 8GB de flash memory, 1GB de memória RAM, chipset Qualcomm MSM8916 Snapdragon 410, CPU Quad-core 1.4 GHz Cortex-A53 e bateria Li-Ion 2470 mAh.

O Projeto Volta é uma nova ferramenta que visa a melhorar o consumo de energia de aplicações Android e conta com serviços que permitem medir o consumo de energia de uma aplicação; contudo, a sua precisão ainda não foi verificada. Apesar disso, mesmo que as medições através do Projeto Volta não sejam precisas, os resultados ainda são relevantes, uma vez que o dado mais importante desta pesquisa é a relação entre as linguagens sobre o consumo de energia e desempenho, e não o valor absoluto do consumo. Uma vez que o Projeto Volta foi utilizado em todas as execuções e em ambas as linguagens, mesmo que ele apresente falhas, todas as execuções estão sujeitas às mesmas condições.

Benchmarks não representam o comportamento de um aplicativo usando JavaScript como linguagem principal (RATANAWORABHAN; LIVSHITS; ZORN, 2010), e por esta razão não é possível extrapolar os resultados para todas as aplicações, uma vez que aplicações Android costumam ser mais intensas em uso de toques na tela e uso de sensores e acesso à rede. Apesar disso ser verdade, através de *benchmarks* é possível avaliar um cenário específico e mensurar

onde há um ganho de desempenho, isolando um comportamento padrão. No nosso caso, o foco foi em aplicativos que faziam uso intenso de CPU, uma vez que esses poderiam apresentar maior ganho de desempenho e eficiência energética. As modificações feitas mostram que mesmo pequenas modificações podem levar a ganhos de eficiência energética significativos.

É possível escrever partes do aplicativo usando C/C++ através do Native Development Kit (NDK), que visa precisamente a melhorar a performance do aplicativo. Apesar da existência do NDK, aplicativos em Android são comumente escritos usando Java, JavaScript ou uma combinação de ambos. Isto se deve as complicações intrínsecas do NDK. Além disso, como já foi citado em capítulos anteriores, não é possível criar um aplicativo usando somente o NDK, pois parte dele tem que estar escrito em Java. Nossa escolha, portanto, foi devido ao padrão utilizado na indústria. Apesar disso, reconhecemos a importância de avaliar os benefícios que utilizar C/C++ podem gerar ao aplicativo em termos de desempenho e eficiência energética, e testes usando o NDK serão conduzidos em trabalhos futuros.

A adaptação dos algoritmos foi necessária em alguns casos para os *benchmarks* do Rosetta Code, como forma de minimizar o impacto que uma implementação do *benchmark* poderia ter na comparação das linguagens. Esta adaptação foi feita de forma a manter o algoritmo o mais similar possível com o original e graças a similaridade sintática das linguagens de programação, não acarretou muito esforço. Algumas adaptações foram checadas por mais pesquisadores para reforçar a sua validade. Exemplos de modificações podem ser encontradas na Seção 3.1.1 e no Apêndice .

Todos os testes foram compilados utilizando o modo *release* do Android Studio, que conta com otimizações. Testes usando o modo *debug* podem apresentar resultados diferentes. O uso do modo *release* se mostrou mais adequado uma vez que queríamos testar o uso real de aplicativos.

Todos os testes foram executados com as baterias com pelo menos 80% da carga completa. Isto garante que a voltagem não baixe para menos de 4V. Isto é importante para minimizar o efeito da variação de voltagem nos resultados das medições. No principal aparelho utilizado neste estudo, a bateria máxima é de 4.2V, sendo possível uma variação de no máximo 5% nos resultados de consumo de energia.

5

Trabalhos Relacionados

Este capítulo trata de trabalhos do estado da arte que influenciaram no desenvolvimento e construção desta pesquisa. O capítulo está divido em duas seções: Seção 5.1, sobre trabalhos relacionados ao consumo de energia e Seção 5.2, sobre pesquisas que realizaram comparações de desempenho ou energia que influenciaram este trabalho.

5.1 Consumo de Energia

O problema do consumo de energia em desenvolvimento de software como um todo tem atraído a atenção de pesquisadores em várias áreas. Esta seção discute os trabalhos que analisam o consumo de energia em dispositivos móveis (Seção 5.1.1) e outros tipos de dispositivos (Seção 5.1.2).

5.1.1 *Sistemas Móveis*

Nos últimos anos, a proliferação de sistemas móveis em suas várias formas realçou a necessidade de se fazer uso ótimo da energia disponível. PATHAK; HU; ZHANG (2012) propuseram o primeiro *profiler* de granularidade fina para investigar onde a energia era gasta dentro de um aplicativo. Usando também meios de granularidade fina, outros artigos buscaram identificar quais parte do aplicativo gastavam mais energia, focando em métodos (COUTO et al., 2014), chamadas de API (LINARES-VáSQUEZ et al., 2014), ou quais tipos de aplicações são mais propensos a consumir mais energia (WILKE et al., 2013). Há também trabalhos que buscam especificar exatamente as linhas do código fonte que consomem mais energia (HAO et al., 2013; LI et al., 2013). Assim como nos trabalhos ligados ao consumo de energia de computadores, pesquisadores notam a importância de apontar exatamente os pontos de maior consumo de energia dentro de uma aplicação.

ZHANG; HINDLE; GERMÁN (2014); WILKE et al. (2013) tiveram como foco de sua pesquisa o padrão de uso do aplicativo pelo usuário, não focando na parte de desenvolvimento dos aplicativos. WILKE et al. (2013) realizaram uma análise do consumo de energia dos aplicativos do Google Play, os correlacionado com o seu ranking e preço. Essa abordagem é bem distinta da

utilizada nessa pesquisa, pois a avaliação de padrão de uso do usuário não está no nosso escopo. Ao focar na parte de desenvolvimento, espera-se fornecer ao desenvolvedor mais informações sobre as opções que ele dispõe para otimizar.

HINDLE (2012) propôs um *framework* para ser usado no desenvolvimento de aplicativos móveis como forma de maximizar a eficiência energética dos aplicativos. Isto é um passo na direção de tentar munir desenvolvedores com maior poder de decisão na hora de fazer escolhas sobre o consumo de energia e desempenho de seus aplicativos, mas também difere do nosso trabalho porque não são feitos testes exaustivos com *benchmarks* e aplicativos. Nosso estudo apresenta também comparação de desempenho e consumo de energia entre as linguagens usadas no desenvolvimento de aplicativos, assuntos não tratados em trabalhos anteriores.

Durante o período dessa pesquisa, não encontramos outros trabalhos que busquem analisar a diferença de consumo de energia entre diferentes abordagens de desenvolvimento de aplicativos.

A maior parte dos trabalhos anteriores que mediram o consumo de energia de aplicativos Android, buscou medir o consumo de energia através de instrumentação do código (PATHAK; HU; ZHANG, 2012; LINARES-VáSQUEZ et al., 2014; LI et al., 2013) ou usando um dispositivo físico para conseguir medir o consumo de energia (COUTO et al., 2014; SABORIDO et al., 2015; SILVA-FILHO et al., 2012; LI et al., 2013; PETERSON et al., 2011). Conforme já explicitado em capítulos anteriores, este trabalho usou principalmente os recursos disponíveis através do Projeto Volta para coletar os dados.

5.1.2 *Outros tipos de dispositivos*

Naturalmente, os primeiros passos com relação ao estudo de desempenho e consumo de energia são anteriores à existência em massa e ubiqüidade de dispositivos móveis. É possível encontrar diversos artigos que tem como foco otimização de consumo de energia a nível de software (PINTO, 2013; FARKAS et al., 2000; PINTO; CASTOR; LIU, 2014), otimizações a nível de hardware (HOROWITZ; INDERMAUR; GONZALEZ, 1994; DAVID et al., 2010) ou a nível de sistema operacional (GE et al., 2007; MERKEL; BELLOSA, 2006; YUAN; NAHRSTEDT, 2003). TIWARI; MALIK; WOLFE (1994) correlacionou o consumo de energia do software ao uso da CPU, no contexto de sistemas embarcados, os precursores dos dispositivos móveis atuais. Essa relação foi confirmada também para sistemas móveis durante esta pesquisa.

Tentando identificar o gasto de energia, diversos trabalhos buscaram analisar o consumo de energia em vários níveis diferentes, indo de chamadas individuais de instrução (TIWARI et al., 1996) até a análise do código fonte (LIU; PINTO; LIU, 2015). Diversos experimentos foram conduzidos utilizando o Intel RAPL (SUBRAMANIAM; FENG, 2013; LIU; PINTO; LIU, 2015; KAMBADUR; KIM, 2014) como ferramenta para medir o consumo de energia.

5.2 Comparações de desempenho e energia

Não foram encontrados estudos realizando comparação do uso de Java e JavaScript nos aplicativos desenvolvidos para Android. Houve dificuldade em encontrar outros trabalhos que comparassem aplicativos desenvolvidos usando Java e JavaScript em qualquer critério. CHARLAND; LEROUX (2011) faz diversas comparações, contudo, as comparações são feitas entre aplicativos nativos e aplicações *web* remotas, i.e., aplicações que tem sua lógica executada num servidor remoto. Mesmo dentro desta comparação, os tópicos que são comparados são a facilidade de uso, código de interface com o usuário, experiência de uso e desempenho. Em particular, este trabalho não apresenta nenhum dado sobre o consumo de bateria ou desempenho quando comparados aplicativos nativos e aplicativos web locais.

NANZ; FURIA (2015) usaram as tarefas presentes no Rosetta Code para comparar oito diferente linguagens de programação (C, Go, C#, Java, F#, Haskell, Python e Ruby) em termos de desempenho e uso de memória, entre outros fatores. LIMA et al. (2016) fizeram uso dos *benchmarks* disponíveis no The Computer Language Benchmark Game para comparar o desempenho e consumo de energia, tendo como foco a linguagem Haskell. Não há um foco nestes trabalho em relação a aplicativos Android ou mesmo qualquer dado sobre o consumo de energia das linguagens supracitadas. Essa referências são importantes contudo por validar a escolha do Rosetta Code e do TCLBG como repositório de *benchmarks*, e terem feito uma seleção de parte dos *benchmarks* que foram usados nessa pesquisa.

6

Conclusão

Este trabalho investigou se há diferença no consumo de energia e desempenho entre as duas abordagens mais comuns de desenvolvimento para aplicativos Android: usando Java como principal linguagem de programação, desenvolvendo o que chamamos de aplicativos nativos, e usando JavaScript como principal linguagem de programação, desenvolvendo o que chamamos de aplicativos híbridos. Além disso, tentamos mostrar algumas das vantagens e desvantagens de usar cada uma das abordagens, usando como base *benchmarks* coletados de repositórios que já haviam sido utilizados em outros estudos, o Rosetta Code e o The Computer Language Benchmark Game e aplicativos de código aberto coletados através do F-Droid.

Os resultados desta pesquisa, científicos e aplicados, serão discutidos nas Seções 6.1 e 6.2, respectivamente. A Seção 6.3 trata sobre os trabalhos futuros.

6.1 Resultados Científicos

Este trabalho apresenta um conjunto de passos a serem seguidos para a execução de trabalhos futuros utilizando o Projeto Volta como forma de analisar o consumo de energia e desempenho de aplicações. A metodologia empregada neste trabalho resultou em dados que se mostraram consistentes no uso de mais de um aparelho móvel e apresentando médias relativamente consistentes nos diversos *benchmarks* e aplicativos testados.

Este trabalho encontrou evidências de que o consumo de energia relacionado a CPU é estável, independente do tipo de uso da aplicação. Corroborando com trabalhos anteriores, encontramos evidência de que, em aplicativos que fazem uso intensivo de CPU, o consumo de energia de um aplicativo está fortemente vinculado ao uso de CPU, trazendo esta informação ao ambiente móvel.

Como contribuição menor do trabalho, temos o fato de que os algoritmos presentes no Rosetta Code não apresentam o mesmo desempenho e refinamento em todas as linguagens. Outros trabalhos que tenham usado o Rosetta Code como fonte de seus *benchmarks* podem precisar revisar os resultados, apresentando soluções para esta desigualdade de desempenho. A solução adotada neste trabalho foi a de adaptar os algoritmos para utilizar somente a melhor

abordagem. Neste trabalho são utilizadas duas linguagens de programação sintaticamente similares, tornando o esforço e a complexidade da adaptação baixos.

O conteúdo resumido deste trabalho foi aceito para publicação na conferência 23ª IEEE Conferência Internacional sobre Análise de Software, Evolução e Reengenharia (SANER-2016), Qualis A2.

6.2 Resultados Aplicados

Em relação a análise dos *benchmarks*, JavaScript obteve desempenho e eficiência energética superiores na maioria dos casos. JavaScript consumiu menos energia em 27 dos 33 *benchmarks* analisados, com Java gastando uma mediana de 97% mais energia no conjunto total de *benchmarks*. JavaScript também teve um desempenho melhor em 21 dos 33 *benchmarks* analisados, com Java gastando uma mediana de 43% mais energia no conjunto total de *benchmarks*. Os dados coletados nesta pesquisa demostram que para aplicações que fazem uso intenso de CPU e operações aritméticas, como é o caso de uma parte considerável dos *benchmarks*, JavaScript tem um desempenho superior ao de Java.

Em relação à análise dos aplicativos modificados, este estudo mostra evidências que é possível obter ganhos significativos em desempenho e eficiência energética com uma quantidade de esforço relativamente pequena em alguns casos. Para obter benefícios da hibridização é necessário que seja possível agrupar um conjunto de execuções de um método ou utilizar uma abordagem onde uma parcela não-negligenciável da lógica do programa possa ser executada em JavaScript. Entretanto, para aplicações onde seria necessária uma comunicação constante entre Java e JavaScript, a hibridação é desaconselhada devido ao *overhead* de comunicação. Não encontramos nenhum caso utilizando este padrão de execução que venha a obter algum benefício em desempenho ou eficiência energética graças à hibridização da aplicação. Entretanto, nossa análise foi restrita a três apps, o que pode prejudicar a generalidade desta conclusão.

Como parte das contribuições aplicadas deste trabalho, apresentamos o conjunto de *benchmarks* adaptados em Java e JavaScript para serem usados no sistema operacional Android, bem como os aplicativos modificados utilizados neste trabalho. Estes conjuntos de *benchmarks* e aplicativos podem também ser usados novamente em pesquisas futuras, tornando possível comparar dados de novos experimentos com os dados apresentados neste trabalho, e permitindo uma comparação justa dos dados.

6.3 Trabalhos Futuros

Como trabalho futuro é necessário levantar mais dados para rejeitar ou validar a hipótese de que JavaScript pode ser mais eficiente energicamente do que Java. Como forma de extrapolar os dados do trabalho e generalizá-los, mais aplicativos devem ser analisados, potencialmente transformando-os em aplicativos híbridos para mensurar a diferença do uso de JavaScript. Estes

aplicativos devem seguir diversos perfis, não somente aplicativos intensos no uso de CPU, como foi feito na presente pesquisa, mas com outros padrões de uso onde JavaScript possa ser mais eficiente do que Java. Como forma de generalizar os resultados, também é necessário que os testes sejam realizados numa gama maior de aparelhos, de diversos fabricantes distintos, com processadores e arquiteturas diferentes e utilizando versões diferentes do sistema operacional.

É necessário avaliar a precisão dos dados coletados pelo Projeto Volta, comparando-os com outras formas de mensurar energia, utilizando tanto métodos de mensuração de granularidade alta, por exemplo, instrumentação do código, como de granularidade baixa, por exemplo, deixar o aparelho descarregar usando o aplicativo. No melhor do nosso conhecimento, este é o primeiro trabalho utilizando o Projeto Volta. Até onde conseguimos averiguar, é necessário validar o grau de similaridade dos dados coletados pelo Projeto Volta com os outros meios de mensuração.

Durante essa pesquisa, demonstramos que JavaScript apresenta vantagem tanto em eficiência energética quanto em desempenho na maioria dos *benchmarks* e algumas versões dos aplicativos. É necessário investigar as razões por trás do fato de JavaScript obter melhor desempenho que Java nos casos onde isto ocorreu e entender os motivos de JavaScript obter um melhor desempenho. As razões para tais vantagens, que parecem contra-intuitivas, são desconhecidas e podem ser a chave para modificações no sistema operacional Android de forma a torná-lo mais rápido e eficiente, e representar uma contribuição vital e abrangente para a indústria.

Também é necessário investigar a melhora em eficiência energética e desempenho que pode ser provida pelo uso do NDK, como forma de avaliar qual seria a melhor forma de realizar modificações no código para obter o máximo de eficiência energética e desempenho nos aplicativos para Android caso seja do interesse do desenvolver realizar algum tipo de modificação no código.

Seria relevante a criação de um conjunto de boas práticas, fornecendo aos desenvolvedores uma forma fácil de saber se é benéfico para o seu aplicativo fazer uso de hibridização. Entretanto, para um melhor desenvolvimento deste conjunto é necessário entender as razões pelas quais JavaScript apresenta uma eficiência energética superior a Java e em que casos exatamente isto ocorre.

Todos os dados referentes a pesquisa, o conjunto de *benchmarks* utilizados e os aplicativos modificados estão disponíveis em `http://nativeorweb.github.io/`.

Referências

ASAFU-ADJAYE, J. The relationship between energy consumption, energy prices and economic growth: time series evidence from asian developing countries. **Energy Economics**, [S.l.], v.22, n.6, p.615 – 625, 2000.

CHARLAND, A.; LEROUX, B. Mobile Application Development: web vs. native. **Commun. ACM**, New York, NY, USA, v.54, n.5, p.49–53, May 2011.

COHEN, M. et al. Energy types. In: ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA 2012, PART OF SPLASH 2012, TUCSON, AZ, USA, OCTOBER 21-25, 2012, 27. **Proceedings...** [S.l.: s.n.], 2012. p.831–850.

COUTO, M. et al. Detecting Anomalous Energy Consumption in Android Applications. In: PROGRAMMING LANGUAGES - 18TH BRAZILIAN SYMPOSIUM, SBLP 2014, MACEIO, BRAZIL, OCTOBER 2-3, 2014. PROCEEDINGS. **Anais...** [S.l.: s.n.], 2014. p.77–91.

DAVID, H. et al. RAPL: memory power estimation and capping. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 16. **Proceedings...** [S.l.: s.n.], 2010. p.189–194. (ISLPED '10).

FARKAS, K. I. et al. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. In: ACM SIGMETRICS INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, 2000. **Proceedings...** [S.l.: s.n.], 2000. p.252–263. (SIGMETRICS '00).

GE, R. et al. CPU MISER: a performance-directed, run-time system for power-aware clusters. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP 2007), SEPTEMBER 10-14, 2007, XI-AN, CHINA, 2007. **Anais...** [S.l.: s.n.], 2007. p.18.

GRIMMER, M. et al. An Efficient Native Function Interface for Java. In: INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICES OF PROGRAMMING ON THE JAVA PLATFORM: VIRTUAL MACHINES, LANGUAGES, AND TOOLS, 2013. **Proceedings...** ACM, 2013. p.35–44.

HäHNEL, M. et al. Measuring Energy Consumption for Short Code Paths Using RAPL. **SIGMETRICS Perform. Eval. Rev.**, [S.l.], v.40, n.3, p.13–17, Jan. 2012.

HAO, S. et al. Estimating Mobile Application Energy Consumption Using Program Analysis. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2013., Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2013. p.92–101. (ICSE '13).

HINDLE, A. Green mining: a methodology of relating software change to power consumption. In: MINING SOFTWARE REPOSITORIES (MSR), 2012 9TH IEEE WORKING CONFERENCE ON. **Anais...** [S.l.: s.n.], 2012. p.78–87.

HOROWITZ, M.; INDERMAUR, T.; GONZALEZ, R. Low-power digital design. In: LOW POWER ELECTRONICS, 1994. IEEE SYMPOSIUM. **Anais...** [S.l.: s.n.], 1994.

KAMBADUR, M.; KIM, M. A. An experimental survey of energy management across the stack. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, OOPSLA 2014, PART OF SPLASH 2014, PORTLAND, OR, USA, OCTOBER 20-24, 2014, 2014. **Proceedings...** [S.l.: s.n.], 2014. p.329–344.

LI, D. et al. Calculating Source Line Level Energy Information for Android Applications. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2013., New York, NY, USA. **Proceedings...** ACM, 2013. p.78–89. (ISSTA 2013).

LI, D.; HALFOND, W. G. J. An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. In: INTERNATIONAL WORKSHOP ON GREEN AND SUSTAINABLE SOFTWARE, 3. **Proceedings...** [S.l.: s.n.], 2014. p.46–53. (GREENS 2014).

LIMA, L. G. et al. Haskell in Green Land: analyzing the energy behavior of a purely functional language. In: SUBMITTED TO SANER'2016. **Anais...** [S.l.: s.n.], 2016.

LINARES-VáSQUEZ, M. et al. Mining Energy-greedy API Usage Patterns in Android Apps: an empirical study. In: WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 11., New York, NY, USA. **Proceedings...** ACM, 2014. p.2–11. (MSR 2014).

LIU, K.; PINTO, G.; LIU, D. Data-Oriented Characterization of Application-Level Energy Optimization. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, 18. **Proceedings...** [S.l.: s.n.], 2015. (FASE'15).

MANOTAS, I.; POLLOCK, L.; CLAUSE, J. SEEDS: a software engineer's energy-optimization decision support framework. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** [S.l.: s.n.], 2014. p.503–514. (ICSE 2014).

MERKEL, A.; BELLOSA, F. Balancing Power Consumption in Multiprocessor Systems. In: ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS 2006, 1. **Proceedings...** [S.l.: s.n.], 2006. p.403–414. (EuroSys '06).

NANZ, S.; FURIA, C. A. A Comparative Study of Programming Languages in Rosetta Code. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - VOLUME 1, 37., Piscataway, NJ, USA. **Proceedings...** IEEE Press, 2015. p.778–788. (ICSE '15).

PATHAK, A.; HU, Y. C.; ZHANG, M. Where is the Energy Spent Inside My App?: fine grained energy accounting on smartphones with eprof. In: ACM EUROPEAN CONFERENCE ON COMPUTER SYSTEMS, 7., New York, NY, USA. **Proceedings...** ACM, 2012. p.29–42. (EuroSys '12).

PETERSON, P. A. H. et al. Investigating Energy and Security Trade-offs in the Classroom with the Atom LEAP Testbed. In: CONFERENCE ON CYBER SECURITY EXPERIMENTATION AND TEST, 4. **Proceedings...** [S.l.: s.n.], 2011. p.11–11. (CSET'11).

PINTO, G. Refactoring Multicore Applications Towards Energy Efficiency. In: COMPANION PUBLICATION FOR CONFERENCE ON SYSTEMS, PROGRAMMING, & APPLICATIONS: SOFTWARE FOR HUMANITY, 2013., New York, NY, USA. **Proceedings...** ACM, 2013. p.61–64. (SPLASH '13).

PINTO, G.; CASTOR, F.; LIU, Y. D. Understanding Energy Behaviors of Thread Management Constructs. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, 2014. **Proceedings...** [S.l.: s.n.], 2014. p.345–360. (OOPSLA '14).

RATANAWORABHAN, P.; LIVSHITS, B.; ZORN, B. G. JSMeter: comparing the behavior of javascript benchmarks with real web applications. In: USENIX CONFERENCE ON WEB APPLICATION DEVELOPMENT, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.3–3.

SABORIDO, R. et al. **On the impact of sampling frequency on software energy measurements**. [S.l.]: PeerJ PrePrints, 2015.

SILVA-FILHO, A. et al. Energy-aware technology-based DVFS mechanism for the android operating system. In: COMPUTING SYSTEM ENGINEERING (SBESC), 2012 BRAZILIAN SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2012. p.184–187.

SUBRAMANIAM, B.; FENG, W.-c. Towards Energy-proportional Computing for Enterprise-class Server Workloads. In: ACM/SPEC INTERNATIONAL CONFERENCE ON PERFORMANCE ENGINEERING, 4. **Proceedings...** [S.l.: s.n.], 2013. p.15–26. (ICPE '13).

TIWARI, V. et al. Instruction Level Power Analysis and Optimization of Software. **Journal of VLSI Signal Processing**, [S.l.], v.13, p.1–18, 1996.

TIWARI, V.; MALIK, S.; WOLFE, A. Power Analysis of Embedded Software: a first step towards software power minimization. **IEEE Transactions on VLSI Systems**, [S.l.], v.2, p.437–445, 1994.

WILKE, C. et al. Comparing Mobile Applications' Energy Consumption. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 28., New York, NY, USA. **Proceedings...** ACM, 2013. p.1177–1179. (SAC '13).

WILKE, C. et al. Energy Consumption and Efficiency in Mobile Applications: a user feedback study. In: GREEN COMPUTING AND COMMUNICATIONS (GREENCOM), 2013 IEEE AND INTERNET OF THINGS (ITHINGS/CPSCOM), IEEE INTERNATIONAL CONFERENCE ON AND IEEE CYBER, PHYSICAL AND SOCIAL COMPUTING. **Anais...** [S.l.: s.n.], 2013. p.134–141.

YUAN, W.; NAHRSTEDT, K. Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems. In: NINETEENTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES. **Proceedings...** [S.l.: s.n.], 2003. p.149–163. (SOSP '03).

ZHANG, C.; HINDLE, A.; GERMÁN, D. M. The Impact of User Choice on Energy Consumption. **IEEE Software**, [S.l.], v.31, n.3, p.69–75, 2014.

Apêndices

Apêndice A - Descrição dos Benchmarks do Rosetta Code

As descrições a seguir foram retiradas do site do Rosetta Code e traduzidas.

Matrix Mult: Multiplicar duas matrizes. Eles podem ser de quaisquer dimensões, desde que o número de colunas da primeira matriz seja igual ao número de linhas da segunda matriz.

The Sieve of Eratosthenes: O Crivo de Eratóstenes é um algoritmo simples que encontra os números primos até um determinado número inteiro.

Tower of Hanoi: A Torre de Hanoi é composto de três hastes, e uma série de discos de diferentes tamanhos que podem deslizar em qualquer hastes. O quebra-cabeça começa com os discos em uma pilha limpa por ordem de tamanho ascendente sobre uma haste, o menor na parte superior, montando assim uma figura cônica. O objetivo do quebra-cabeça é mover toda a pilha para outra haste, obedecendo a seguintes regras: 1) Apenas um disco pode ser movido de cada vez. 2) Cada movimento consiste em levar o disco superior a partir de uma das pilhas e colocando-o em cima de uma outra pilha, i.e., um disco apenas pode ser movido, se for o disco superior numa pilha. 3) Nenhuma disco pode ser colocado em cima de um disco menor.

Com três discos, o enigma pode ser resolvido em sete movimentos. O número mínimo de movimentos necessários para resolver um quebra-cabeça Torre de Hanói é $2n - 1$, onde n é o número de discos.

Happy Number: Um Happy Number é definido pelo seguinte processo: Começando com qualquer número inteiro positivo, substituir o número pela soma dos quadrados dos seus dígitos, repetindo o processo até que o número seja igual a 1 (onde permanecerá), ou que circula num laço infinito, que não inclui 1. Esses números para que este processo termina em 1 são Happy Numbers, enquanto que aqueles que não terminam em 1 Unhappy Numbers.

Combinations: Dado números não negativos inteiros m e n, gerar todas as combinações de tamanho m dos inteiros de 0 a $n - 1$ de forma ordenada (cada combinação é ordenada e toda a tabela é ordenada).

Count in factors: Escreva um programa que conta a partir de 1, exibindo cada número como a multiplicação de seus fatores primos.

Knapsack Unbounded: Um viajante está autorizado a levar tanto quanto ele gosta dos seguintes itens, contanto que consiga fazer caber em sua mochila, e dentro do limite de no máximo, 25 *pesos* no total;. e que a capacidade da sua mochila é de 0,25 m^3. O viajante sabe o peso, volume e preço de cada de item. Ele pode levar quantas cópias quiser de cada item mas somente itens inteiros. Maximize o valor dos itens que o viajante leva com ele.

Zero-One Knapsack: Um turista pode transportar no máximo 4kg nele. Ele cria uma lista do que ele quer trazer para a viagem, mas o peso total de todos os itens é maior do que 4kg. Ele decide, então, adicionar colunas à sua lista inicial detalhando seus pesos e um valor numérico representando o quão importante o item é para a viagem. O turista pode optar por fazer qualquer combinação de itens da lista, mas apenas um de cada item está disponível. Ele só deve levar itens inteiros. Maximize o valor da mochila de forma que não exceda 4kg.

Knapsack Bounded: Um turista pode transportar no máximo 4kg nele. Ele cria uma lista do que ele quer trazer para a viagem, mas o peso total de todos os itens é maior do que 4kg. Ele decide, então, adicionar colunas à sua lista inicial detalhando seus pesos e um valor numérico representando o quão importante o item é para a viagem. O turista pode optar por fazer qualquer combinação de itens da lista, e um número infinito de itens está disponível. Ele só deve levar itens inteiros. Maximize o valor da mochila de forma que não exceda 4kg.

Man or Boy: Imitar o exemplo de Knuth em Algol 60 em outro idioma, na medida do possível, i.e., Cria uma árvore de B quadros de chamada que se referem uns aos outros e ao quadro de chamadas que às contem, A, cada um com sua própria cópia de k que muda toda vez que o quadro B associado é chamado.

NQueens: O enigma *N* rainhas é o problema de colocar *N* rainhas, para $N > 3$ em um tabuleiro de xadrez *NxN* de modo que não há duas rainhas ameaçam uns aos outros. Assim, a solução exige que não há duas rainhas compartilham a mesma linha, coluna ou diagonal.

Perfect Number: Escreva uma função que diz se um número é perfeito. Um número perfeito é um inteiro positivo que é a soma de seus divisores positivos adequados excluindo o número em si. De modo equivalente, um número perfeito é um número que é metade da soma de todos os seus divisores positivos (incluindo o próprio).

SeqNonSquares: Mostre que a seguinte fórmula dá a sequência de números naturais não-quadrado: $n + \lfloor(\frac{1}{2} + \sqrt{n})\rfloor$

BubbleSort: O Bubble Sort funciona por meio sequencialmente sobre uma lista, comparando cada valor ao que imediatamente após ele. Se o primeiro valor é maior do que o segundo, as suas posições são trocadas. Ao longo de um número de passagens, no máximo igual ao número de elementos na lista, todos os valores de derivar para as suas posições corretas (valores grandes "bolha"rapidamente para o fim, empurrando os outros para baixo em torno deles). Porque cada passagem encontra o item máximo e o coloca na extremidade, a porção da lista a ser classificado pode ser reduzida em cada passagem. Uma variável booleana é usado para controlar se as alterações foram feitas no passe atual; depois de uma completa sem mudar nada, o algoritmo termina.

CountingSort: Counting Sort é um algoritmo para classificar uma coleção de objetos de acordo com as chaves, sendo estas pequenos números inteiros; isto é, é um ordenador de números inteiros. Ele funciona contando o número de objetos com chaves distintas, e usando aritmética nestes contadores para determinar a posição de cada chave na sequencia.

GnomeSort: Gnome Sort é um algoritmo de ordenação que é semelhante ao Insert Sort,

exceto que mover um elemento para o seu devido lugar é realizado por uma série de trocas, como no Bubble Sort.

HeapSort: A ideia básica é tornar o array em uma estrutura de pilha binária, o qual tem a propriedade que permite a recuperação eficiente e remoção do elemento superior. Heap Sort requer acesso aleatório, portanto, apenas pode ser utilizado em estruturas similares a arrays.

InsertSort: Insert sort é um simples algoritmo de ordenação, eficiente quando aplicado a um pequeno número de elementos. Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados.

MergeSort: A ideia básica consiste em dividir o problema em vários sub-problemas e resolver esses sub-problemas através da recursividade e após todos os sub-problemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos sub-problemas.Como o algoritmo do Merge Sort usa a recursividade em alguns problemas esta técnica não é muito eficiente devido ao alto consumo de memória e tempo de execução.

PancakeSort: Panqueca Sort se baseia numa pilha desordenada de panquecas na ordem de tamanho quando uma espátula pode ser inserida em qualquer ponto na pilha e usada para virar todas as panquecas acima dela.

QuickSort: A estratégia consiste em rearranjar as chaves de modo que as chaves "menores"precedam as chaves "maiores". Em seguida o Quicksort ordena as duas sublistas de chaves menores e maiores recursivamente até que a lista completa se encontre ordenada. Os passos são: 1) Escolha um elemento da lista, denominado pivô 2) Rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada partição 3) Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;

ShellSort: O algorimo é um refinamento do Insert Sort, contudo diferete dele pelo fato de no lugar de considerar o array a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles. Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores. Nos grupos menores é aplicado o método da ordenação por inserção.

HofstadterQ: A sequência Hofstadter Q é definido como: Q (1) = Q (2) = 1, Q (n) = Q (N - Q (n - 1)) + Q (N - Q (n - 2)), N > 2. É similar a sequência de Fibonacci. Contudo, na sequência de Fibonacci o próximo termo é a soma dos dois termos anteriores, na sequência Q os dois termos anteriores dizer o quão longe se deve voltar na sequência Q para encontrar os dois números que se deve somar para fazer o próximo termo da sequência.

Apêndice B - Descrição dos Benchmarks do TCLBG

As descrições a seguir foram retiradas do site do The Computer Language Benchmark Game do que cada um dos benchmarks deve fazer e traduzidas.

BinaryTree: 1) Definir uma classe nó de árvore e métodos, um recorde árvore de nós e procedimentos, ou um tipo de dados algébrica e funções, ou ... 2) Alocar uma árvore binária para a "esticar"a memória, verifique que ela existe, e ele desalocar. 3) Alocar uma árvore binária de longa duração que vai continuar viva, enquanto outras árvores são alocados e desalocados. 4) Alocar, percorrer e desalocar muitas árvores binárias bottom-up. 5) Alocar uma árvore. 6) Percorrer os nós de árvore, verificando os itens de nós (e talvez desalocar o nó). 7) Desalocar a árvore. 8) Verifique se a árvore binária de longa duração continua a existir.

Fannkuch: 1) Tendo uma permutação de 1, ..., n, por exemplo: 4,2,1,5,3. 2) Usar o primeiro elemento, no exemplo 4, e inverter a ordem das primeiras 4 elementos: 5,1,2,4,3. 3) Repita este procedimento até o primeiro elemento seja 1, de forma que qualquer inversão não mude nada: 3,4,2,1,5, 2,4,3,1,5, 4,2,3, 1,5, 1,3,2,4,5. 4) Contar o número de inversões, aqui 5. 5) Mantenha um *checksum*. 6) *checksum* = *checksum* + (caso índice_permutação seja par então contador_inversão senão -contador_inversão). 7) *checksum* = *checksum* + (sinal_chavamento_-1_1 * contador_inversão). 8) Faça isso para todas as $n!$ permutações, e grave o número máximo de inversões necessários para qualquer permutação.

Fasta: 1) Gerar seqüências de DNA, copiando a partir de uma determinada sequência. 2) Gerar seqüências de DNA, por seleção aleatória ponderada a partir de 2 alfabetos. 3) Converter a probabilidade esperada da seleção de cada nucleótido em probabilidades cumulativas. 4) Fazer a correspondência entre um número aleatório e essas probabilidades cumulativas para selecionar cada nucleotídeo (usar a pesquisa linear ou busca binária). 5) Usar este gerador de congruência linear para calcular um número aleatório cada vez que um nucleótido precisa ser selecionado (não armazenar em cache a sequência de números aleatórios).

Nbody: Modelar as órbitas dos planetas Júpiter, usando o mesmo simples *symplectic-integrator*.

Spectral: 1) Calcular a norma espectral de uma matriz infinita A, com entradas a11 = 1, a12 = $\frac{1}{2}$, a21 = $\frac{1}{3}$, a13 = $\frac{1}{4}$, a22 = $\frac{1}{5}$, a31 = $\frac{1}{6}$, etc 2) Implementar 4 diferentes funções, ou procedimentos, ou métodos como o programa C#

RegexDna: 1) Ler tudo de um arquivo no formato FASTA e gravar o comprimento da sequência. 2) Usar o mesmo simples padrão de expressão regular casar-trocar para remover as

descrições da sequência FASTA e registrar o comprimento da sequência. 3) Usar os mesmos simples padrões de expressão regular, representando DNA 8-meros e seu complemento inverso (com um curinga em uma posição), e (um padrão de cada vez) contam correspondências no arquivo FASTA. 4) Escrever o padrão de expressão regular e conte. 5) Usar os mesmos simples padrões de expressão regular para fazer as alternativas de código IUB explícito, e (um padrão de cada vez) case e troque o padrão no arquivo FASTA e registre o comprimento da sequência. 6) Escrever os 3 comprimentos sequência gravada.

RevComp: 1) Ler linha por linha um arquivo no formato FASTA. 2) para cada sequência: 2.a) escrever o id, a descrição e a sequência inversa do complemento no formato FASTA.

Knucleotide: 1) Ler linha por linha um arquivo no formato FASTA. 2) Extrair as sequências de DNA TRÊS 3) Definir um procedimento/função para atualizar uma tabela hash de chaves k-nucleotide e contar seus valores, para um quadro de leitura particular - ainda que nos combinemos todas as contagem de k-nucleotide para todos os quadros de leitura (o tabela hash crescer a partir de um tamanho padrão pequeno) 4) Aplicar esse procedimento / função e para hashtable para: 4.a) Contar todas as sequências 1-nucleótidos e 2-nucleotídeos. Escrever o código e frequência percentual, ordenado de forma descendente pela frequência e, em seguida ascendente pela chave k-nucleotídeo. 4.b) contar todas as sequências 3- 4- 6- 12- e 18 nucleótidos. Escrever a contagem e código para as sequências específicas GGT GGTA GGTATT GGTATTTTAATT GGTATTTTAATTTATAGT.

Apêndice C - Modificações nos benchmarks

```
1   //Trecho original em JavaScript:
2   function queenPuzzle(rows, columns) {
3     if (rows <= 0) {
4       return [[]];
5     } else {
6       return adxdQueen(rows - 1, columns);
7     }
8   }
9   function addQueen(newRow, columns, prevSolution) {
10    var newSolutions = [];
11    var prev = queenPuzzle(newRow, columns);
12    for (var i = 0; i < prev.length; i++) {
13      var solution = prev[i];
14      for (var newColumn = 0; newColumn < columns; newColumn++) {
15        if (!hasConflict(newRow, newColumn, solution))
16          newSolutions.push(solution.concat([newColumn]))
17      }
18    }
19    return newSolutions;
20  }
21  function hasConflict(newRow, newColumn, solution) {
22    for (var i = 0; i < newRow; i++) {
23      if (solution[i] == newColumn ||
24      solution[i] + i == newColumn + newRow ||
25      solution[i] - i == newColumn - newRow) {
26        return true;
27      }
28    }
29    return false;
30  }
31  function queenPuzzleExecute(loop,size){
32    for (var i = 0; i < loop; i++) {
33      queenPuzzle(size,size)
34    }
35  }
```

Figura C.1: *Benchmark* nQueens. Trecho Original em JavaScript.

```java
//Trecho original em Java:
public class NQueens {

  public static int DEFAULT_LOOP = 4;
  public static int DEFAULT_SIZE = 13;
  private static int[] b;

  static boolean unsafe(int y) {
    int x = b[y];
    for (int i = 1; i <= y; i++) {
      int t = b[y - i];
      if (t == x || t == x - i || t == x + i) {
        return true;
      }
    }
    return false;
  }
  public static void putboard(int number) {
    for (int y = 0; y < number; y++) {
      for (int x = 0; x < number; x++) {
        //System.out.print((b[y] == x) ? "|Q" : "|_");
      }
    }
  }
  public static void execute(int loop, int size){
    for (int i = 0; i < loop ; i++) {
      run(size);
    }
  }
  public static void run(int number) {
    b = new int[number];
    int y = 0;
    b[0] = -1;
    while (y >= 0) {
      do {
        b[y]++;
      } while ((b[y] < number) && unsafe(y));
      if (b[y] < number) {
        if (y < number-1) {
          b[++y] = -1;
        } else {
          putboard(number);
      }
      } else {
        y--;
      }
    }
  }
}
```

Figura C.2: *Benchmark* nQueens. Trecho Original em Java.

```
1    //Trecho em JavaScript modificado, baseado em Java:
2  var b = [];
3  function unsafe(y){
4    var x = b[y];
5    for(var i = 1; i <= y; i++){
6      var t = b[y - i];
7      if (t == x ||
8      t == x - i ||
9      t == x + i) {
10       return true;
11     }
12   }
13   return false;
14 }
15 function putboard(number){
16   for (var y = 0; y < number; y++) {
17     for (var x = 0; x < number; x++) {
18       //print((b[y] == x) ? "|Q" : "|_");
19     }
20   }
21 }
22 function executeNQueens(loop,size){
23   for (var i = 0; i < loop; i++) {
24     run(size);
25   }
26 }
27 function run(number){
28   b = [];
29   var y = 0;
30   b[0] = -1;
31   while (y >= 0) {
32     do {
33       b[y]++;
34     } while ((b[y] < number) && unsafe(y));
35     if (b[y] < number) {
36       if (y < number-1) {
37         b[++y] = -1;
38       } else {
39         putboard(number);
40       }
41     } else {
42       y--;
43     }
44   }
45 }
```

Figura C.3: *Benchamrk* nQueens. Trecho em JavaScript modificiado levando em conta a solução em Java. Esta solução consome 23x menos energia que a solução original em JavaScript.

Apêndice D - Tela dos benchmarks

Figura D.1: Telas do aplicativo usado para rodar os benchmarks escritos em Java. A tela da esquerda é a tela inicial e a tela da direita a tela ao término da execução.

Figura D.2: Telas do aplicativo usado para rodar os benchmarks escritos em JavaScript. A tela da esquerda é a tela inicial e a tela da direita a tela ao término da execução. A caixa de texto com "AA"era utilizada para depuração.

Apêndice E - Tabelas

Tabela E.1: Dados sobre todas as execuções dos aplicativos modificados.

Aplicativo	Lg	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Media	σ
anDOF Stepwise	Java	tExec (s)	35.844	31.478	32.005	31.326	40.364	33.415	31.248	33.517	36.034	32.34	33.76	8.32%
		tCPU (s)	14.62	9.98	10.07	9.91	11.07	12.22	9.51	10.01	9.28	10.49	10.72	7.95%
		energia (J)	15.12	12.60	12.63	12.43	13.62	14.83	11.94	12.63	11.79	13.23	13.08	6.74%
	JS	tExec (s)	36.623	35.726	28.136	35.312	37.497	37.635	29.608	31.256	33.064	32.907	33.78	9.41%
		tCPU (s)	13.67	12.97	12.13	13.68	18.73	16.41	12.43	12.47	14.71	14.94	14.21	14.52%
		energia (J)	16.27	15.84	12.77	19.58	19.30	19.87	13.09	15.55	18.58	15.55	16.64	15.58%
anDOF Export	Java	tExec (s)	575.109	602.337	649.335	670.058	713.331	653.122	602.266	602.037	600.401	660.129	632.81	5.97%
		tCPU (s)	612.91	641.48	689.38	686.37	756.27	691.89	640.64	640.96	634.31	730.66	672.49	6.14%
		energia (J)	717.12	704.16	709.92	712.80	715.68	748.80	741.60	711.36	696.96	698.40	715.68	2.38%
	JS	tExec (s)	29.109	29.05	26.982	30.49	37.384	34.439	35.652	36.028	33.25	35.652	32.80	10.28%
		tCPU (s)	25.41	26.95	24.96	30.04	30.45	32.02	35.08	35.92	29.42	31.65	30.19	10.96%
		energia (J)	19.58	25.63	27.79	24.34	23.62	24.34	27.07	32.26	26.50	23.76	25.49	10.16%
TriRose Stepwise	Java	tExec (s)	35.205	30.85	31.092	29.351	30.458	28.927	29.672	30.562	30.715	30.483	30.73	2.32%
		tCPU (s)	17.18	12.09	14.33	13.47	13.32	12.5	13.37	15.46	14.52	14.5	14.07	7.20%
		energia (J)	11.33	8.27	10.22	9.63	10.20	9.26	10.67	11.65	11.23	10.77	10.32	9.51%
	JS	tExec (s)	77.721	75.688	74.906	73.14	72.789	73.513	76.22	75.613	74.02	71.631	74.52	1.96%
		tCPU (s)	38.75	34.02	35.75	36.43	36.12	35.74	39.49	36.22	37.25	34.81	36.46	4.00%
		energia (J)	20.30	16.99	18.00	19.01	19.73	19.44	20.02	19.44	19.87	19.30	19.21	4.84%
Tri Rose Batch	Java	tExec (s)	34.587	34.224	32.881	32.176	33.2	41.368	34.546	32.318	33.199	34.76	34.33	7.71%
		tCPU (s)	13.24	13.2	14.17	12.05	12.89	15.46	14.04	13.26	14.47	10.89	13.37	9.60%
		energia (J)	9.29	10.74	10.31	8.99	9.36	10.18	9.53	9.85	10.50	7.29	9.60	10.29%
	JS	tExec (s)	34.126	31.928	37.149	31.296	30.159	36.605	37.02	38.78	32.598	31.265	34.09	8.98%
		tCPU (s)	13.92	6.12	5.83	13.5	11.17	6.4	12.55	12.89	12.94	12.35	10.77	28.81%
		energia (J)	8.63	4.44	4.95	9.23	7.59	5.28	8.76	8.76	8.41	7.96	7.40	23.62%
EnigmAndroid	Java	tExec (s)	84.112	90.74	90.479	88.829	90.546	89.751	91.114	90.03	90.678	89.34	89.56	0.81%
		tCPU (s)	90.25	96.66	97.27	103.4	97.09	97.34	103.56	96.1	97.64	105.82	98.51	3.56%
		energia (J)	75.46	79.34	77.47	86.11	77.62	77.04	82.37	75.74	81.50	86.98	79.96	4.78%
	JS	tExec (s)	33.373	35.191	32.993	35.791	34.29	34.48	35.033	36.502	35.398	35.891	34.89	2.81%
		tCPU (s)	34.69	36.37	29.76	30.94	36.27	28.85	28.27	28.66	32.38	28.67	31.49	9.64%
		energia (J)	32.40	36.86	27.65	24.77	30.24	23.76	22.75	22.75	26.06	22.46	26.97	16.49%

Tabela E.2: Tabela 1 de 4 contendo os dados da execução do Rosetta Code.

Benchmark	Lg	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Média	σ	Mediana
Matrix Mult	Java	tExec (s)	53.95	54.45	54.70	55.83	54.54	52.36	51.54	54.84	54.31	52.54	53.90	2.47%	54.38
		tCPU (s)	50.05	50.38	51.72	51.43	49.03	47.67	45.90	50.89	50.68	46.25	49.40	4.25%	50.22
		energia (J)	37.58	42.48	39.74	39.89	60.91	40.90	35.86	43.92	51.84	34.70	42.78	18.17%	40.39
	JS	tExec (s)	19.31	19.61	20.14	18.82	19.56	18.94	19.40	19.91	20.15	19.31	19.51	2.30%	19.48
		tCPU (s)	15.08	9.09	13.60	12.45	13.12	12.86	15.33	9.99	13.94	15.32	13.08	15.42%	13.36
		energia (J)	13.28	8.64	10.64	12.25	10.73	10.45	12.07	10.12	14.37	12.57	11.51	13.74%	11.40
Sieve of Eratosthenes	Java	tExec (s)	64.64	64.43	67.45	72.14	72.18	71.65	72.15	72.29	72.01	72.23	70.12	3.81%	72.08
		tCPU (s)	59.87	60.22	68.42	67.21	73.25	70.85	74.28	71.60	73.53	73.19	69.24	6.11%	71.23
		energia (J)	77.62	71.28	78.34	79.63	90.72	79.06	86.11	86.26	92.88	89.14	83.10	7.99%	82.87
	JS	tExec (s)	44.96	46.59	45.90	47.69	45.81	45.80	45.57	46.78	45.87	45.65	46.06	1.44%	45.84
		tCPU (s)	41.73	36.87	41.32	36.56	39.47	40.81	41.47	40.37	41.38	40.56	40.05	4.46%	40.69
		energia (J)	32.40	40.75	32.83	46.22	41.76	37.44	30.38	31.25	44.93	43.92	38.19	15.05%	39.10
Tower of Hanoi	Java	tExec (s)	92.36	90.46	86.60	93.06	92.54	89.99	94.71	85.93	96.23	89.71	91.16	3.58%	91.41
		tCPU (s)	80.31	81.58	76.07	83.12	82.22	77.27	80.71	81.42	91.09	83.44	81.72	4.92%	81.50
		energia (J)	81.50	86.69	102.53	78.91	85.39	71.42	89.71	70.70	76.46	80.93	82.43	11.39%	81.22
	JS	tExec (s)	97.69	99.00	106.22	96.13	96.09	96.43	96.31	95.08	95.39	95.58	97.39	3.40%	96.22
		tCPU (s)	89.75	94.31	100.09	93.36	90.86	89.45	93.39	85.05	88.69	90.98	91.59	4.34%	90.92
		energia (J)	81.79	80.06	85.97	74.16	65.66	67.68	80.93	92.02	98.50	66.10	79.29	13.94%	80.50
Happy Numbers	Java	tExec (s)	61.64	58.43	60.55	59.85	64.60	65.41	65.72	66.82	61.82	62.37	62.72	4.41%	62.10
		tCPU (s)	55.96	50.31	56.12	55.77	64.69	61.29	58.21	61.54	60.17	60.54	58.46	6.80%	59.19
		energia (J)	76.03	60.19	74.02	62.64	69.55	69.70	64.66	70.85	67.68	65.09	68.04	6.07%	68.62
	JS	tExec (s)	57.63	57.09	56.91	69.80	70.71	70.79	70.34	70.42	70.14	71.22	66.50	8.46%	70.24
		tCPU (s)	48.15	52.71	53.30	66.25	66.94	66.80	64.23	63.34	60.28	66.79	60.88	8.85%	63.79
		energia (J)	35.14	40.61	38.74	49.10	49.54	47.23	52.13	54.29	42.91	50.54	46.02	10.97%	48.17
Combinations	Java	tExec (s)	58.90	58.86	59.77	65.19	58.77	50.68	52.07	50.80	54.54	50.74	56.03	8.71%	56.65
		tCPU (s)	53.20	52.42	53.62	57.63	52.35	46.26	46.32	46.70	44.48	45.91	49.89	8.62%	49.53
		energia (J)	43.92	46.08	51.41	47.09	47.52	39.31	44.50	41.18	61.78	36.14	45.89	15.39%	45.29
	JS	tExec (s)	67.83	65.79	74.66	80.66	91.93	74.76	79.62	71.37	71.15	72.10	74.99	9.50%	73.38
		tCPU (s)	65.39	61.34	67.44	73.91	86.43	68.84	69.78	68.96	64.26	62.56	68.89	10.30%	68.14
		energia (J)	61.34	45.22	49.82	57.46	62.93	53.57	49.82	55.58	48.96	48.38	53.31	9.72%	51.70
Count in factors	Java	tExec (s)	74.53	74.75	72.29	72.51	72.17	72.47	72.32	72.14	75.73	76.00	73.49	2.09%	72.49
		tCPU (s)	67.39	65.56	68.99	69.59	65.42	61.92	67.39	69.38	70.49	70.24	67.64	3.99%	68.19
		energia (J)	60.77	76.32	63.07	66.38	50.69	65.09	53.71	58.90	59.18	74.02	62.81	12.82%	61.92
	JS	tExec (s)	91.34	91.23	90.97	90.80	90.58	91.15	91.14	93.28	90.65	99.76	92.09	3.03%	91.15
		tCPU (s)	86.63	85.02	86.70	85.61	85.38	85.84	84.85	86.70	85.50	90.61	86.28	1.92%	85.73
		energia (J)	72.29	65.81	63.22	61.49	60.62	89.71	60.77	67.68	63.50	74.16	67.92	13.02%	64.66

Tabela E.3: Tabela 2 de 4 contendo os dados da execução do Rosetta Code.

Benchmark	Lang	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Média	σ	Mediana
Knapsack Unbounded	Java	tExec (s)	56.22	56.41	56.29	56.30	54.72	57.65	64.56	53.93	57.33	59.61	57.30	5.17%	56.35
		tCPU (s)	49.19	53.43	50.41	50.23	44.73	53.65	57.97	47.43	52.14	49.62	50.88	7.11%	50.32
		energia (J)	37.58	50.98	39.60	42.77	33.84	40.46	40.75	33.98	39.46	37.01	39.64	12.25%	39.53
	JS	tExec (s)	13.98	13.58	13.83	13.90	13.80	13.70	13.46	13.59	13.23	13.94	13.70	1.58%	13.75
		tCPU (s)	7.33	7.36	7.19	7.21	7.02	8.38	8.22	6.79	6.08	8.28	7.39	9.83%	7.27
		energia (J)	6.21	8.29	5.90	7.95	6.97	9.09	7.39	8.67	5.08	8.19	7.37	16.85%	7.67
Zero-One Knapsack	Java	tExec (s)	40.46	44.02	44.69	43.76	44.28	46.10	52.36	52.46	44.67	52.27	46.51	8.01%	44.68
		tCPU (s)	57.72	58.83	59.79	65.47	64.62	62.69	66.92	76.12	62.14	72.70	64.70	8.37%	63.66
		energia (J)	79.63	73.73	73.15	82.80	81.22	74.88	69.26	77.62	75.46	75.89	76.36	5.10%	75.67
	JS	tExec (s)	20.71	19.76	21.83	20.09	18.58	18.63	19.31	19.16	19.59	20.09	19.78	4.68%	19.68
		tCPU (s)	13.52	9.91	14.10	13.24	12.82	13.20	13.00	16.93	14.20	10.54	13.15	14.73%	13.22
		energia (J)	16.27	12.44	11.65	11.22	11.71	11.76	15.41	15.26	13.78	9.85	12.94	13.73%	12.10
Knapsack Bounded	Java	tExec (s)	66.97	61.43	60.07	58.49	60.27	67.15	67.32	66.45	62.69	61.55	63.24	4.96%	62.12
		tCPU (s)	86.43	73.16	71.61	73.53	78.45	81.71	81.69	81.29	93.58	90.13	81.16	8.74%	81.49
		energia (J)	93.02	85.68	86.83	90.43	93.74	84.96	84.67	84.96	97.06	93.60	89.50	4.97%	88.63
	JS	tExec (s)	59.79	61.78	61.71	71.00	60.64	61.31	61.30	61.08	68.19	67.75	63.46	5.88%	61.51
		tCPU (s)	53.71	55.06	57.06	64.85	53.39	57.21	57.48	53.39	65.90	63.50	58.16	7.88%	57.14
		energia (J)	57.89	64.08	62.21	66.67	65.81	68.26	48.53	54.43	60.34	55.15	60.34	10.35%	61.27
Man or Boy	Java	tExec (s)	89.24	86.41	89.02	99.34	101.57	103.79	102.23	97.41	98.19	95.95	96.32	5.78%	97.80
		tCPU (s)	93.76	91.02	93.67	104.49	108.59	101.12	107.16	102.57	107.80	105.71	101.59	5.81%	103.53
		energia (J)	107.28	111.74	108.72	101.38	106.42	95.47	97.78	96.48	99.79	95.62	102.07	5.60%	100.58
	JS	tExec (s)	61.12	71.97	67.82	65.88	65.79	68.55	68.65	67.08	67.61	67.84	67.23	2.59%	67.72
		tCPU (s)	55.93	66.54	61.31	58.73	62.50	62.91	61.98	59.94	61.46	61.74	61.30	3.34%	61.60
		energia (J)	41.04	49.54	44.93	53.86	46.22	48.10	52.85	54.00	45.79	44.78	48.11	7.50%	47.16
NQueens	Java	tExec (s)	21.44	21.43	22.00	18.41	20.71	18.99	21.56	19.48	14.27	14.57	19.29	14.15%	20.10
		tCPU (s)	15.10	12.26	14.87	14.90	11.09	11.36	14.88	13.60	13.86	14.11	13.60	10.44%	13.99
		energia (J)	10.83	9.00	13.08	12.17	9.69	10.14	8.64	11.74	13.23	10.84	10.94	14.69%	10.84
	JS	tExec (s)	11.93	12.01	12.41	11.77	12.09	11.91	13.34	13.44	14.56	11.87	12.53	7.25%	12.05
		tCPU (s)	1.90	1.98	2.47	1.87	2.00	2.37	3.46	3.37	3.61	2.05	2.51	26.58%	2.21
		energia (J)	1.81	1.89	2.48	1.53	1.64	1.93	2.82	3.38	3.33	1.99	2.28	29.12%	1.96
Perfect Number	Java	tExec (s)	156.06	163.15	162.23	161.01	160.42	171.35	176.13	176.29	154.49	154.43	163.56	4.82%	161.62
		tCPU (s)	148.67	152.69	150.35	153.61	157.78	160.72	168.63	171.15	149.83	155.88	156.93	4.61%	154.75
		energia (J)	148.32	152.64	161.28	162.72	120.53	175.68	158.40	168.48	106.85	124.56	147.95	15.46%	155.52
	JS	tExec (s)	22.09	21.80	21.47	23.11	21.58	22.90	21.59	21.25	15.92	16.16	20.79	12.19%	21.58
		tCPU (s)	15.32	15.32	17.84	14.53	14.93	17.94	18.75	14.96	15.10	16.18	16.09	9.30%	15.32
		energia (J)	13.16	12.18	16.13	16.13	12.60	14.54	17.86	15.12	12.66	16.70	14.71	13.00%	14.83

Tabela E.4: Tabela 3 de 4 contendo os dados da execução do Rosetta Code.

Benchmark	Lang	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Média	σ	Mediana
SeqNonSquares	Java	tExec (s)	550.83	536.45	536.41	536.37	536.75	536.67	536.76	536.77	536.59	536.54	538.01	0.09%	536.63
		tCPU (s)	546.13	535.75	535.79	535.72	536.01	535.80	534.65	535.84	535.99	536.01	536.77	0.10%	535.82
		energia (J)	381.60	372.96	374.40	372.96	390.24	383.04	374.40	393.12	372.96	365.76	378.14	2.26%	374.40
	JS	tExec (s)	15.04	16.86	17.58	18.99	14.72	16.74	16.31	15.96	18.16	17.92	16.83	7.24%	16.80
		tCPU (s)	8.58	10.10	12.63	9.57	9.21	10.14	12.15	13.16	13.87	16.18	11.56	19.02%	11.15
		energia (J)	8.34	9.56	11.00	9.07	9.82	9.65	11.79	11.28	11.20	12.56	10.43	10.69%	10.41
BubbleSort	Java	tExec (s)	105.91	110.63	105.33	97.66	98.79	95.30	110.52	109.67	89.19	91.15	101.41	7.80%	102.06
		tCPU (s)	89.61	96.44	89.56	81.90	83.39	79.15	95.80	94.53	86.43	90.31	88.71	6.70%	89.59
		energia (J)	89.28	67.68	64.51	71.57	91.44	74.30	69.26	73.15	63.07	101.81	76.61	15.96%	72.36
	JS	tExec (s)	37.60	44.99	43.94	42.50	36.27	45.00	44.45	44.95	45.54	44.82	43.00	6.35%	44.63
		tCPU (s)	32.78	41.75	41.52	38.79	33.45	42.73	41.36	42.10	44.43	40.34	39.93	7.43%	41.44
		energia (J)	36.00	32.11	34.27	42.19	32.26	38.30	31.10	41.04	39.60	28.08	35.50	13.21%	35.14
CountingSort	Java	tExec (s)	43.54	37.17	38.14	38.52	38.14	37.13	38.05	37.76	31.63	36.62	37.67	5.31%	37.91
		tCPU (s)	32.31	22.74	30.12	30.08	29.89	22.76	26.14	30.35	31.19	36.24	29.18	13.92%	30.10
		energia (J)	33.12	22.90	29.52	26.35	24.62	25.20	22.46	29.38	31.82	24.34	26.97	11.39%	25.78
	JS	tExec (s)	19.43	19.33	21.95	19.45	19.03	20.93	18.63	22.93	18.25	18.29	19.82	8.01%	19.38
		tCPU (s)	12.44	12.09	15.04	15.40	14.82	14.83	14.72	14.57	16.18	13.61	14.37	7.61%	14.77
		energia (J)	14.10	10.57	15.84	14.11	14.40	14.83	15.26	15.12	12.48	10.30	13.70	14.15%	14.26
GnomeSort	Java	tExec (s)	162.15	146.36	154.37	153.23	153.13	153.18	154.17	153.23	137.24	144.22	151.13	3.74%	153.21
		tCPU (s)	143.68	133.10	142.55	142.45	138.58	142.28	142.94	141.41	132.81	141.45	140.13	2.73%	141.87
		energia (J)	123.26	105.98	120.53	102.10	140.40	134.35	140.11	102.38	94.32	116.93	118.04	14.02%	118.73
	JS	tExec (s)	64.50	64.00	64.15	55.05	61.35	61.14	61.43	61.50	61.34	61.40	61.59	4.00%	61.42
		tCPU (s)	57.12	57.76	58.99	50.20	56.17	53.13	53.36	61.38	56.44	56.66	56.12	5.65%	56.55
		energia (J)	46.22	44.06	44.35	36.29	40.46	37.30	38.16	45.22	41.62	40.18	41.39	7.37%	41.04
HeapSort	Java	tExec (s)	63.23	63.07	63.01	63.45	62.61	63.18	62.10	63.02	69.08	68.99	64.17	4.00%	63.12
		tCPU (s)	56.09	55.89	52.70	56.38	55.44	60.60	59.72	55.60	60.64	68.42	58.15	7.51%	56.24
		energia (J)	40.18	40.90	45.65	46.08	39.89	45.50	44.35	55.30	42.77	48.53	44.91	9.60%	44.93
	JS	tExec (s)	52.04	54.94	49.54	50.87	49.04	52.53	54.92	57.33	52.18	52.46	52.58	4.85%	52.32
		tCPU (s)	47.46	51.04	44.74	40.22	39.08	42.26	51.53	52.92	50.66	51.88	47.18	11.06%	49.06
		energia (J)	37.15	38.45	35.71	42.05	30.10	45.65	39.60	38.30	37.58	49.39	39.40	13.40%	38.38
InsertSort	Java	tExec (s)	84.78	94.22	94.29	84.25	94.55	94.91	94.10	94.82	92.33	92.19	92.04	3.47%	94.16
		tCPU (s)	73.99	86.93	90.84	81.43	90.81	90.10	90.61	84.00	83.92	91.76	86.44	4.22%	88.52
		energia (J)	72.86	62.78	71.42	60.48	79.49	80.21	65.52	77.33	60.05	65.38	69.55	11.00%	68.47
	JS	tExec (s)	85.33	85.38	85.62	85.98	85.29	87.45	87.02	85.25	84.14	82.05	85.35	1.75%	85.35
		tCPU (s)	82.82	81.19	78.64	81.98	74.67	79.66	82.89	81.39	81.33	74.91	79.95	3.55%	81.26
		energia (J)	59.04	58.03	67.39	79.49	54.86	58.46	64.37	71.57	62.06	54.86	63.01	12.33%	60.55

Tabela E.5: Tabela 4 de 4 contendo os dados da execução do Rosetta Code.

Benchmark	Lang	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Média	σ	Mediana
MergeSort	Java	tExec (s)	51.59	51.80	52.02	52.29	53.02	52.22	52.17	52.03	60.58	60.95	53.87	6.62%	52.20
		tCPU (s)	54.20	49.73	54.16	57.98	53.79	58.57	59.16	56.53	61.78	67.54	57.34	8.41%	57.26
		energia (J)	57.89	56.59	61.92	60.91	60.19	66.38	68.98	66.38	59.04	67.82	62.61	6.57%	61.42
	JS	tExec (s)	25.33	25.67	26.62	26.58	25.09	25.68	25.81	29.42	29.70	29.72	26.96	6.70%	26.19
		tCPU (s)	18.31	18.37	20.15	20.11	19.52	15.67	21.02	21.37	26.44	21.70	20.27	13.45%	20.13
		energia (J)	14.98	16.99	17.14	20.16	16.42	15.41	18.86	21.60	19.44	17.71	17.87	10.43%	17.42
PancakeSort	Java	tExec (s)	43.21	41.31	42.81	44.18	42.68	42.85	42.93	43.69	47.60	47.53	43.88	4.71%	43.07
		tCPU (s)	33.46	28.31	35.16	36.62	33.21	36.48	33.90	29.92	42.39	47.18	35.66	15.51%	34.53
		energia (J)	24.77	23.90	26.93	38.59	28.37	31.54	25.34	29.52	30.96	33.26	29.32	14.33%	28.94
	JS	tExec (s)	24.52	25.05	24.48	24.54	24.80	24.59	24.86	24.64	24.70	24.94	24.71	0.73%	24.67
		tCPU (s)	21.64	22.32	23.65	24.43	24.61	19.76	23.59	20.42	20.12	25.12	22.57	8.72%	22.96
		energia (J)	16.27	20.02	17.42	22.61	23.18	17.57	22.32	20.02	14.83	19.01	19.32	13.54%	19.51
QuickSort	Java	tExec (s)	53.98	61.08	71.65	53.02	57.12	56.76	56.30	58.74	49.51	70.50	58.87	11.84%	56.94
		tCPU (s)	62.89	78.64	102.10	77.04	79.44	79.34	80.17	85.50	81.02	115.82	84.20	14.98%	79.81
		energia (J)	86.98	85.10	109.30	83.52	85.54	85.97	88.85	92.16	84.10	101.52	90.30	9.37%	86.47
	JS	tExec (s)	68.82	69.79	79.52	62.75	72.00	75.07	77.20	64.37	71.73	71.38	71.26	7.25%	71.55
		tCPU (s)	64.28	64.19	69.82	52.40	66.53	69.05	70.74	57.72	63.75	66.78	64.53	8.80%	65.41
		energia (J)	51.84	52.13	54.29	62.35	53.57	69.84	53.57	66.38	44.93	48.96	55.79	13.90%	53.57
ShellSort	Java	tExec (s)	55.66	55.35	55.34	55.68	55.79	55.51	55.44	55.28	53.80	53.88	55.17	1.27%	55.39
		tCPU (s)	52.29	45.39	51.37	50.62	49.79	50.32	50.32	50.04	51.37	53.58	50.51	4.05%	50.47
		energia (J)	51.41	36.72	50.11	42.48	52.27	41.90	45.79	37.44	37.30	37.44	43.29	12.81%	42.19
	JS	tExec (s)	75.39	78.06	81.82	75.98	77.12	78.42	81.14	75.59	76.13	78.44	77.81	2.68%	77.59
		tCPU (s)	70.40	74.07	78.02	70.58	72.29	73.38	77.41	68.87	69.68	65.08	71.98	5.41%	71.44
		energia (J)	68.83	68.54	56.16	70.85	56.02	52.56	72.86	56.59	49.25	70.13	62.18	13.71%	62.57
HofstadterQ	Java	tExec (s)	73.91	78.70	84.05	83.45	83.56	83.14	83.07	83.15	83.02	83.37	81.94	1.86%	83.14
		tCPU (s)	67.19	72.04	77.82	77.23	79.92	78.59	77.95	73.26	76.42	79.33	75.98	3.34%	77.53
		energia (J)	62.78	74.16	96.19	61.20	91.30	86.11	76.75	73.15	70.99	93.74	78.64	14.33%	75.46
	JS	tExec (s)	16.15	16.45	17.56	16.27	16.38	16.34	16.07	17.50	16.72	16.11	16.55	3.20%	16.36
		tCPU (s)	12.61	6.46	10.34	13.55	6.42	9.63	11.17	12.72	11.37	10.94	10.52	22.18%	11.06
		energia (J)	10.02	5.16	8.21	11.03	5.98	7.66	8.78	9.62	10.17	11.45	8.81	23.12%	9.20

Tabela E.6: Tabela 1 de 2 contendo os dados da execução do The Computer Language Benchmark Game no aparelho original.

Benchmark	Lg	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Média	σ	Mediana
BinaryTree	Java	65.14	64.66	65.25	64.79	65.20	65.30	64.75	65.01	64.76	65.29	65.01	0.38%	65.07	52.20
		79.85	76.98	76.82	76.82	77.58	76.90	82.14	73.53	76.38	79.34	77.63	2.83%	76.94	57.26
		78.62	76.61	74.74	76.18	78.62	75.89	81.22	73.87	73.73	77.62	76.71	2.95%	76.39	61.42
	JS	43.34	41.36	41.66	41.83	42.17	42.37	42.02	41.85	40.49	43.52	42.06	1.82%	41.94	26.19
		37.66	34.70	35.49	34.74	40.54	35.72	37.54	37.47	36.05	41.79	37.17	6.41%	36.76	20.13
		35.71	34.99	30.82	27.36	38.16	30.96	32.98	28.22	28.37	33.12	32.07	10.38%	31.97	17.42
Fannkuch	Java	108.83	103.53	105.69	104.11	104.49	103.99	102.84	103.38	103.89	105.18	104.59	0.81%	104.05	43.07
		346.34	359.47	373.09	390.75	365.05	376.18	359.22	372.60	381.58	387.12	371.14	2.89%	372.85	34.53
		269.28	270.72	277.92	292.32	275.04	283.68	270.72	282.24	286.56	289.44	279.79	2.65%	280.08	28.94
	JS	162.90	174.69	157.52	144.77	133.62	141.50	137.85	135.10	139.71	142.19	146.98	8.39%	141.85	24.67
		151.81	169.29	150.48	140.16	125.93	139.41	130.37	128.13	137.46	136.49	140.95	8.91%	138.44	22.96
		113.62	120.53	112.46	123.70	86.69	96.19	100.37	93.17	95.62	106.99	104.93	11.47%	103.68	19.51
Fasta SC	Java	37.39	37.11	37.40	32.61	37.04	37.23	37.17	37.11	37.23	37.17	36.75	3.92%	37.17	56.94
		33.13	34.82	32.40	28.37	32.94	27.07	27.06	31.75	27.15	32.19	30.69	9.22%	31.97	79.81
		40.46	35.86	37.01	35.28	24.77	27.94	30.24	34.99	20.88	36.86	32.43	17.13%	35.14	86.47
	JS	60.45	62.02	64.30	63.93	66.03	66.52	64.37	63.35	64.84	65.02	64.08	2.00%	64.34	71.55
		54.40	55.11	56.86	56.93	58.61	59.97	57.75	56.90	57.31	54.18	56.80	2.85%	56.92	65.41
		68.83	52.70	54.43	66.82	55.01	71.14	43.78	41.76	65.38	61.78	58.16	16.48%	58.39	53.57
Fasta MC	Java	32.46	29.00	27.70	30.13	19.72	30.87	31.52	30.33	27.73	28.45	28.79	11.54%	29.56	55.39
		43.07	38.89	40.36	34.87	32.54	40.64	34.39	39.08	31.65	41.00	37.65	9.23%	38.99	50.47
		36.86	38.74	40.03	32.83	31.39	36.43	27.65	38.59	30.24	37.58	35.04	11.85%	36.65	42.19
	JS	60.45	62.02	64.30	63.93	66.03	66.52	64.37	63.35	64.84	65.02	64.08	2.00%	64.34	77.59
		54.40	55.11	56.86	56.93	58.61	59.97	57.75	56.90	57.31	54.18	56.80	2.85%	56.92	71.44
		68.83	52.70	54.43	66.82	55.01	71.14	43.78	41.76	65.38	61.78	58.16	16.48%	58.39	62.57
Nbody	Java	83.41	88.61	83.23	80.27	80.08	79.97	80.48	80.25	80.13	80.21	81.66	3.31%	80.26	83.14
		73.78	74.53	73.82	75.21	73.13	77.80	73.77	73.35	73.28	73.74	74.24	1.87%	73.78	77.53
		55.01	65.66	55.15	69.41	51.55	57.17	75.02	63.79	58.46	56.16	60.74	11.82%	57.82	75.46
	JS	57.69	58.06	59.04	62.50	57.39	56.96	59.68	59.17	58.73	53.03	58.23	4.11%	58.40	16.36
		52.61	47.79	57.00	55.52	50.71	46.89	52.42	54.06	57.03	44.15	51.82	8.48%	52.52	11.06
		46.08	48.67	50.98	56.45	37.87	34.85	38.74	40.90	42.05	35.42	43.20	16.37%	41.47	9.20

Tabela E.7: Tabela 2 de 2 contendo os dados da execução do The Computer Language Benchmark Game no aparelho original.

Benchmark	Lang	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Média	σ	Mediana
Spectral	Java	39.77	47.72	41.28	47.80	38.25	47.52	47.56	47.48	47.51	47.64	45.25	7.39%	47.51	52.20
		131.35	179.37	137.10	167.94	120.05	177.51	164.81	143.42	160.72	162.75	154.50	12.01%	161.74	57.26
		121.10	134.50	118.80	125.14	104.26	132.91	123.98	119.38	119.66	120.53	122.03	6.85%	120.82	61.42
	JS	86.13	88.98	89.05	89.25	88.88	89.00	97.89	100.80	88.77	88.74	90.75	4.83%	88.99	26.19
		79.52	84.44	83.75	84.53	82.08	86.42	92.91	98.56	86.62	78.83	85.77	6.52%	84.49	20.13
		82.94	62.64	72.00	92.88	58.90	86.54	67.25	76.32	65.38	57.89	72.27	15.88%	69.62	17.42
RegenDna SC	Java	149.04	146.81	136.54	148.12	146.00	147.99	145.57	145.95	145.49	146.52	145.80	2.25%	146.26	43.07
		139.40	137.13	127.24	143.10	140.54	141.60	139.95	140.49	141.37	141.85	139.27	3.25%	140.52	34.53
		143.71	133.06	114.34	125.28	106.70	129.60	116.21	105.84	117.65	110.59	120.30	7.67%	116.93	28.94
	JS	11.19	10.28	10.33	10.17	10.30	10.46	10.27	9.35	10.21	10.25	10.28	2.97%	10.28	24.67
		5.56	6.54	7.33	5.74	5.28	5.46	3.29	2.68	4.53	5.41	5.18	26.65%	5.44	22.96
		5.40	7.00	7.62	6.90	5.26	6.16	3.51	2.89	4.56	5.14	5.44	27.99%	5.33	19.51
RegenDna MC	Java	65.26	63.55	62.70	62.21	62.40	62.39	62.14	62.37	62.49	62.45	62.79	0.64%	62.42	56.94
		68.76	59.26	63.85	62.79	62.94	63.74	62.98	63.94	59.32	63.90	63.15	2.84%	63.36	79.81
		65.23	57.17	61.06	56.59	61.92	60.34	58.75	58.75	57.02	60.34	59.72	3.04%	59.54	86.47
	JS	11.19	10.28	10.33	10.17	10.30	10.46	10.27	9.35	10.21	10.25	10.28	2.97%	10.28	71.55
		5.56	6.54	7.33	5.74	5.28	5.46	3.29	2.68	4.53	5.41	5.18	26.65%	5.44	65.41
		5.40	7.00	7.62	6.90	5.26	6.16	3.51	2.89	4.56	5.14	5.44	27.99%	5.33	53.57
RevComp	Java	65.26	63.55	62.70	62.21	62.40	62.39	62.14	62.37	62.49	62.45	62.79	0.64%	62.42	55.39
		68.76	59.26	63.85	62.79	62.94	63.74	62.98	63.94	59.32	63.90	63.15	2.84%	63.36	50.47
		65.23	57.17	61.06	56.59	61.92	60.34	58.75	58.75	57.02	60.34	59.72	3.04%	59.54	42.19
	JS	172.46	173.17	170.90	170.40	171.24	169.22	172.16	173.04	170.64	171.98	171.52	0.71%	171.61	77.59
		170.79	151.11	149.13	163.50	163.98	164.45	165.80	166.75	160.39	167.04	162.29	3.89%	164.22	71.44
		128.88	142.85	138.53	139.25	149.76	133.63	124.27	141.12	124.27	138.38	136.09	5.79%	138.46	62.57
Knucleotide	Java	61.30	59.40	52.22	58.24	59.23	58.98	58.37	58.38	58.62	58.16	58.29	3.56%	58.50	83.14
		190.31	198.57	156.91	195.84	197.69	194.10	193.27	194.67	195.20	192.95	190.95	6.38%	194.39	77.53
		143.42	146.88	136.80	146.88	148.32	145.44	143.14	144.00	145.44	145.44	144.58	2.18%	145.44	75.46
	JS	97.12	93.69	88.57	93.84	92.27	91.61	90.90	93.69	91.78	93.35	92.68	1.76%	92.81	16.36
		93.01	89.22	83.83	86.27	85.73	84.84	84.38	87.24	86.42	85.44	86.64	1.79%	86.00	11.06
		80.93	68.11	62.06	71.86	80.64	88.70	60.34	76.90	61.78	85.82	73.71	13.69%	74.38	9.20

Tabela E.8: Tabela 1 de 2 contendo os dados da execução do The Computer Language Benchmark Game no segundo aparelho.

Benchmark	Lang	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Média	σ	Mediana
BinaryTree	Java	53.24	55.52	55.82	55.41	54.36	54.79	54.18	54.25	52.94	56.52	54.70	1.85%	54.57	52.20
		15.22	17.65	71.22	31.71	60.43	54.70	78.42	31.75	71.84	65.10	47.42	43.18%	54.70	57.26
		11.71	13.12	55.58	24.91	47.23	41.18	61.34	25.20	55.58	51.41	38.73	41.06%	44.21	61.42
	JS	57.58	57.37	57.50	57.42	56.34	56.44	57.49	57.91	56.73	57.27	57.20	0.88%	57.39	26.19
		24.77	31.02	10.95	42.06	11.48	47.11	25.85	50.05	22.27	54.64	32.02	48.38%	28.44	20.13
		18.29	24.77	7.78	32.98	7.99	36.86	19.58	39.46	17.14	41.47	24.63	49.93%	22.18	17.42
Fannkuch	Java	113.41	108.70	110.28	110.75	110.48	109.66	110.68	109.67	108.82	108.84	110.13	0.72%	109.97	43.07
		420.10	384.30	274.08	285.85	400.78	311.41	288.87	373.96	406.98	402.63	354.90	14.98%	379.13	34.53
		316.80	295.20	216.14	230.40	318.24	246.24	226.08	300.96	325.44	318.24	279.37	15.12%	298.08	28.94
	JS	201.80	215.55	203.36	202.64	202.97	202.58	201.24	202.69	204.28	201.88	203.90	2.02%	202.66	24.67
		157.80	196.18	141.69	191.10	159.14	168.62	167.78	192.59	157.12	192.55	172.46	10.75%	168.20	22.96
		125.57	156.96	113.62	152.64	127.58	134.35	134.78	154.08	125.57	154.08	137.92	10.70%	134.57	19.51
Fasta SC	Java	61.82	61.25	62.72	61.60	61.45	61.51	61.23	62.28	64.52	61.76	62.01	1.60%	61.68	56.94
		13.63	42.52	49.34	53.53	37.58	54.67	24.44	58.15	30.20	56.21	42.03	27.42%	45.93	79.81
		8.80	33.84	38.02	42.48	29.95	42.62	19.15	45.50	22.75	43.20	32.63	27.68%	35.93	86.47
	JS	105.83	105.49	108.35	104.63	105.25	107.15	104.94	105.93	105.66	105.99	105.92	1.03%	105.75	71.55
		81.66	101.47	102.27	55.26	101.44	87.39	100.86	53.47	97.15	85.30	86.63	21.36%	92.27	65.41
		64.66	79.49	78.91	43.78	79.49	64.37	79.06	41.62	75.17	66.96	67.35	21.30%	71.06	53.57
Fasta MC	Java	31.88	31.40	30.95	31.34	33.41	30.98	31.58	31.60	31.53	31.94	31.66	2.18%	31.55	55.39
		46.31	38.22	24.84	51.55	37.72	38.79	32.56	52.24	11.50	50.36	38.41	32.97%	38.51	50.47
		33.84	30.24	18.58	37.44	29.09	28.66	25.78	38.74	8.06	37.87	28.83	32.69%	29.66	42.19
	JS	105.83	105.49	108.35	104.63	105.25	107.15	104.94	105.93	105.66	105.99	105.92	1.03%	105.75	77.59
		81.66	101.47	102.27	55.26	101.44	87.39	100.86	53.47	97.15	85.30	86.63	21.36%	92.27	71.44
		64.66	79.49	78.91	43.78	79.49	64.37	79.06	41.62	75.17	66.96	67.35	21.30%	71.06	62.57
Nbody	Java	152.60	152.13	151.98	153.25	151.88	152.18	152.56	152.22	152.63	151.96	152.34	0.27%	152.20	83.14
		125.32	134.49	135.19	138.89	147.96	128.94	100.11	128.31	120.39	112.90	127.25	10.71%	128.63	77.53
		100.66	107.86	108.14	111.17	116.93	103.68	78.77	102.96	95.33	89.28	101.48	11.03%	103.32	75.46
	JS	119.17	120.49	119.91	120.36	120.40	118.35	118.98	118.76	119.62	119.98	119.60	0.62%	119.77	16.36
		104.93	108.74	81.11	105.96	83.44	110.75	103.93	63.37	83.98	88.50	93.47	16.18%	96.22	11.06
		83.52	85.68	64.08	85.10	66.10	88.27	82.80	49.68	66.24	69.70	74.12	16.62%	76.25	9.20

Tabela E.9: Tabela 2 de 2 contendo os dados da execução do The Computer Language Benchmark Game no segundo aparelho. O benchmark RegenDna SC e RegenDna MC não executaram no segundo aparelho devido a falta de memória.

Benchmark	Lang	Medida	Exec 1	Exec 2	Exec 3	Exec 4	Exec 5	Exec 6	Exec 7	Exec 8	Exec 9	Exec 10	Média	σ	Mediana
Spectral	Java	53.24	55.52	55.82	55.41	54.36	54.79	54.18	54.25	52.94	56.52	54.70	1.85%	54.57	52.20
		15.22	17.65	71,22	31.71	60.43	54.70	78.42	31.75	71.84	65.10	47.42	43.18%	54.70	57.26
		11.71	13.12	55.58	24.91	47.23	41.18	61.34	25.20	55.58	51.41	38.73	41.06%	44.21	61.42
	JS	57.58	57.37	57.50	57.42	56.34	56.44	57.49	57.91	56.73	57.27	57.20	0.88%	57.39	26.19
		24.77	31.02	10.95	42.06	11.48	47.11	25.85	50.05	22.27	54.64	32.02	48.38%	28.44	20.13
		18.29	24.77	7.78	32.98	7.99	36.86	19.58	39.46	17.14	41.47	24.63	49.93%	22.18	17.42
RevComp	Java	113.41	108.70	110.28	110.75	110.48	109.66	110.68	109.67	108.82	108.84	110.13	0.72%	109.97	43.07
		420.10	384.30	274.08	285.85	400.78	311.41	288.87	373.96	406.98	402.63	354.90	14.98%	379.13	34.53
		316.80	295.20	216.14	230.40	318.24	246.24	226.08	300.96	325.44	318.24	279.37	15.12%	298.08	28.94
	JS	201.80	215.55	203.36	202.64	202.97	202.58	201.24	202.69	204.28	201.88	203.90	2.02%	202.66	24.67
		157.80	196.18	141.69	191.10	159.14	168.62	167.78	192.59	157.12	192.55	172.46	10.75%	168.20	22.96
		125.57	156.96	113.62	152.64	127.58	134.35	134.78	154.08	125.57	154.08	137.92	10.70%	134.57	19.51
Knucleotide	Java	61.82	61.25	62.72	61.60	61.45	61.51	61.23	62.28	64.52	61.76	62.01	1.60%	61.68	56.94
		13.63	42.52	49.34	53.53	37.58	54.67	24.44	58.15	30.20	56.21	42.03	27.42%	45.93	79.81
		8.80	33.84	38.02	42.48	29.95	42.62	19.15	45.50	22.75	43.20	32.63	27.68%	35.93	86.47
	JS	105.83	105.49	108.35	104.63	105.25	107.15	104.94	105.93	105.66	105.99	105.92	1.03%	105.75	71.55
		81.66	101.47	102.27	55.26	101.44	87.39	100.86	53.47	97.15	85.30	86.63	21.36%	92.27	65.41
		64.66	79.49	78.91	43.78	79.49	64.37	79.06	41.62	75.17	66.96	67.35	21.30%	71.06	53.57

Apêndice F - Gráficos

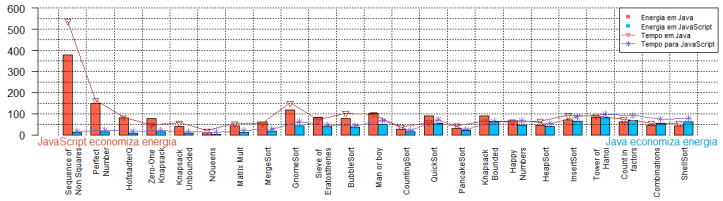

Figura F.1: Resultados dos *benchmarks* do Rosetta Code com todos os benchmarks. As barras são organizadas de acordo com o ganho relativo em consumo de energia segundo a fórmula (*Consumo de energia em Java ÷ Consumo de energia em JavaScript*) para cada um dos benchmarks.

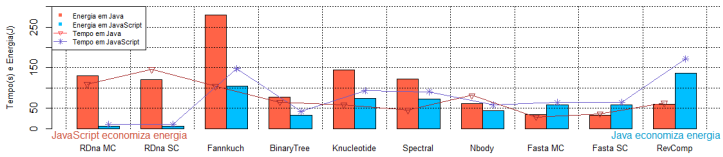

Figura F.2: Resultados dos *benchmarks* do The Computer Language Benchmark Game com todos os benchmarks. As barras são organizadas de acordo com o ganho relativo em consumo de energia segundo a fórmula (*Consumo de energia em Java ÷ Consumo de energia em JavaScript*) para cada um dos benchmarks.

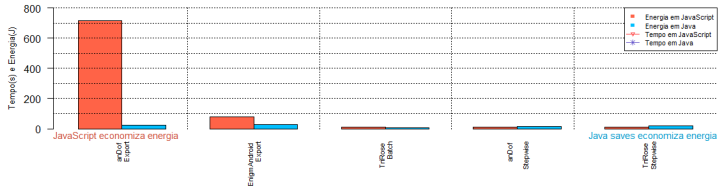

Figura F.3: Resultados de todos os aplicativos modificados. As barras são organizadas de acordo com o ganho relativo em consumo de energia segundo a fórmula (*Consumo de energia em Java ÷ Consumo de energia em JavaScript*) para cada um dos benchmarks.