# Caracterizando o Consumo de Energia de APIs de E/S da Linguagem Java

**Gilson Rocha**[1], **Gustavo Pinto**[1], **Fernando Castor**[2]

[1]Universidade Federal do Pará (UFPA)
Belém – PA – Brasil

[2]Unversidade Federal de Pernambuco (UFPE)
Recife – PE - Brasil

gilsonrocha@gmail.com,gpinto@ufpa.br,castor@cin.ufpe.br

***Resumo.*** *APIs que implementam operações de entrada e saída (E/S) são a base para a construção de grande parte dos sistemas de software desenvolvidos que precisam, por exemplo, acessar um banco de dados, a internet ou periféricos. No entanto, há poucos trabalhos conduzidos com o objetivo de melhor entender e caracterizar o consumo de energia de APIs de que implementam operações de E/S. Neste trabalho, foram instrumentadas 23 classes Java com essas características (micro-benchmarks), além de dois benchmarks que fazem uso de algumas das classes instrumentadas, com o fim de entender o comportamento de consumo de energia das mesmas.*

## Introdução

O consumo de energia em software tem atraído recente interesse não só de grande empresas de tecnologia, mas também de pesquisadores interessados em entender as causas e, eventualmente, mitigar suas raízes [Pinto et al. 2014, Sahin et al. 2016, McIntosh et al. 2018]. Apesar de importantes avanços terem sido introduzidos no passado recente [Bartenstein and Liu 2013, Liu et al. 2015, Oliveira et al. 2017], pouco ainda se sabe sobre o consumo de energia de APIs que implementam operações de entrada e saída (E/S). Essas APIs são amplamente utilizadas por desenvolvedores de software, em particular, pela necessidade que sistemas de software não-triviais têm em realizar tais operações, seja para acessar um banco de dados, periféricos, ou serviços externos.

O presente trabalho tem como objetivo minimizar essa lacuna através de uma caracterização do comportamento do consumo de energia de APIs que realizam operações de E/S na linguagem Java. Através da instrumentação de 23 classes do pacote `java.io`, foi possível responder questões de pesquisa sobre qual classe Java apresenta um melhor consumo de energia (QP1), se as classes com maior consumo de energia são também aquelas mais frequentemente empregadas (QP2) e se classes intercambiáveis podem ser alternadas de forma a melhorar o consumo de energia de benchmarks não-triviais (QP3).

Os resultados preliminares desse trabalho apontam para uma significante variação do consumo de energia entre as classes instrumentadas. Por exemplo, enquanto a classe `Files` apresenta o menor consumo de energia, a classe `FileInputStream` apresenta consumo de energia 3 vezes pior. Foi também observado que as classes frequentemente empregadas em projetos de código fonte aberto não são necessariamente as que tem o melhor consumo de energia (`Scanner`, apesar de ser frequentemente empregada, apresenta

o quarto pior consumo de energia). Por fim, quando classes intercambiáveis são empregadas em benchmarks não-triviais, o consumo de energia pode aumentar mais de 100%. Isso significa que há uma oportunidade para reduzir o consumo de energia de aplicações existentes que demandaria pouquíssimo esforço dos desenvolvedores.

## Método

### Questões de Pesquisa

Este trabalho é guiado pelas seguintes questões de pesquisa (QP):

**QP1:** Qual o consumo de energia das APIs Java que implementam operações de E/S?
**QP2:** As APIs que implementam operações de E/S mais frequentemente utilizadas são as que tem menor consumo de energia?
**QP3:** APIs intercambiáveis podem ser alternadas de forma a melhorar o consumo de energia de benchmarks não-triviais?

Para responder a primeira questão de pesquisa (**QP1**), foram estudados 23 APIs que implementações operações de entrada e saída na linguagem Java. Para responder a segunda questão de pesquisa (**QP2**), os dados da primeira questão de pesquisa foram correlacionados com a frequência da utilização das classes em projetos de código aberto. Esta ocorrência foi medida através da infraestrutura de código `BOA` [Dyer et al. 2013]. Por fim, por intercambiáveis, na **QP3**, entende-se implementações que possam ser alteradas como mínimo ou nenhum esforço (por exemplo, troca entre classes que implementam a mesma interface). Essas modificações foram então empregadas em benchmarks consolidados e utilizados por estudos anteriores [Lima et al. 2016, Oliveira et al. 2017].

### Micro-Benchmarks e Benchmarks

Nesta Seção descrevemos os Micro-Benchmarks e Benchmarks utilizados neste estudo.

**Micro-benchmarks**. A linguagem de programação Java é particularmente rica quando se trata de APIs que manipulam operações de entrada e saída. Em particular, a implementação das operações de E/S está concentrada nas subclasses de quatro classes abstratas: `OutputStream`, `InputStream`, `Reader` e `Writer`. As classes `InputStream` e `OutputStream` implementam E/S de um vetor de bytes, enquanto que as classes `Reader` e `Writer` para E/S de caracteres. A Tabela 1 sumariza as classes e os métodos estudados.

Cada classe estudada é responsável por implementar um método de leitura ou um método de escrita. Foram estudadas a maior parte das classes que implementam E/S que residem no pacote `java.io`. Por questões de simplicidade, nossos benchmarks incluem todas as subclasses das quatro classes citadas anteriormente, com exceção de:

- `DataOutputStream`: por ter o método `readLines()` depreciado;
- `LineNumberInputStream` e `StringBufferInputStream`: por terem sido depreciadas;
- `ObjectOutputStream`: por focar em E/S de objetos java;
- `PipedOutputStream` e `PipedInputStream`: por funcionarem apenas em conjunto. As duas precisam ser implementadas juntas uma vez que uma gera dados para a outra consumir;

**Tabela 1. Classes e a assinatura das APIs que realizam operações de E/S. A coluna versão indica a versão do Java em que a classe foi introduzida.**

| Nome da classe | Método Instrumentado | Herda de | Versão |
|---|---|---|---|
| BufferedWriter | `void write(String str)` | java.io.Writer | 1.1 |
| FileWriter | `void write(String str)` | java.io.Writer | 1.1 |
| StringWriter | `void write(String str)` | java.io.Writer | 1.1 |
| PrintWriter | `void write(String str)` | java.io.Writer | 1.1 |
| CharArrayWriter | `void write(String str)` | java.io.Writer | 1.1 |
| BufferedReader | `int read()` | java.io.Reader | 1.1 |
| LineNumberReader | `int read()` | java.io.Reader | 1.1 |
| CharArrayReader | `int read()` | java.io.Reader | 1.1 |
| PushbackReader | `int read()` | java.io.Reader | 1.1 |
| FileReader | `int read()` | java.io.Reader | 1.1 |
| StringReader | `int read()` | java.io.Reader | 1.1 |
| FileOutputStream | `void write(byte[] b)` | java.io.OutputStream | 1.0 |
| ByteArrayOutputStream | `void write(byte[] b)` | java.io.OutputStream | 1.0 |
| BufferedOutputStream | `void write(byte[] b)` | java.io.OutputStream | 1.0 |
| PrintStream | `void print(String str)` | java.io.OutputStream | 1.0 |
| FileInputStream | `int read()` | java.io.InputStream | 1.0 |
| BufferedInputStream | `int read()` | java.io.InputStream | 1.0 |
| PushbackInputStream | `int read()` | java.io.InputStream | 1.0 |
| ByteArrayInputStream | `int read()` | java.io.InputStream | 1.0 |

- `PipedReader` e `PipedWriter`: pelo mesmo motivo anterior;
- `SequenceInputStream`: por ele ler em sequências dois inputs, como aponta o construtor `SequenceInputStream(InputStream s1, InputStream s2)`. Ele inicia e termina o s1 e depois pula automaticamente para o s2;

As demais classes do pacote `java.io` são de exceção ou classes de configuração. Adicionalmente às classes apresentadas na Tabela 1, foram também incluídas a classes `java.util.Scanner` (introduzida no Java 1.5) e `java.nio.file.Files` (introduzida no Java 1.7). Estas classes não herdam de nenhuma das quatro classes abstratas supra-citadas, logo implementam operações de E/S de forma ligeiramente diferente das classes apresentadas anteriormente. Por exemplo, a classe `Files` implementa três métodos diferentes para leitura de dados (`List<String> readAllLines(Path path)`, `Stream<String> lines(Path path)` e `BufferedReader newBufferedReader(Path path)`). Ainda, a classe `Scanner` conta com dois métodos (`String nextLine()` & `boolean hasNext()`) que devem ser utilizados em conjunto para leitura de dados.

Para cada uma das classes, foi criado um teste que lê um arquivo de formato `HTML` de 20 MBs (ou escrevendo, nas operações de escrita). As 23 classes apresentadas nessa seção compreendem o nosso conjunto de **micro-benchmarks**.

**Benchmarks**. Além dos micro-benchmarks, foram utilizados também dois benchmarks do Computer Language Benchmark Game[1], que tem por objetivo comparar os desempenhos de diversas linguagens de programação. São eles:

**FASTA:** Este benchmark organiza as estruturas de DNA, RNA ou proteínas. FASTA escreve os resultados (que são textos com sequências do tipo "GGGATACCG-TACA", etc) através do método `write(byte[] b)` de um `OutputStream`. Este benchmark tem 329 linhas de código.

[1] https://benchmarksgame-team.pages.debian.net

**K-NUCLEOTIDE:** Este benchmark trabalha em conjunto com o FASTA. Através do método `String readLine()` de um `BufferedReader`, este benchmark lê todas as linhas do arquivo gerado pelo FASTA contabilizando as diversas sequências de DNA contidas neste arquivo. Este benchmark tem 205 linhas de código.

Estes benchmarks foram escolhidos pois fazem uso de pelo menos um dos micro-benchmarks descritos nessa seção. Ao contrário dos micro-benchmarks, os benchmarks são programas que, embora pequenos (em torno de 200 linhas de código), foram projetados para ter bom desempenho. Estes benchmarks, inclusive, são frequentemente utilizados em estudos que visam entender ou otimizar o desempenho de linguagens de programação [Lima et al. 2016] e máquinas virtuais [Barrett et al. 2017]. Para cada um dos benchmarks escolhidos, seu código fonte foi modificado de forma a utilizar diferentes implementações de micro-benchmarks intercambiáveis.

### Execução de Experimentos

Para executar os experimentos, foi utilizado um notebook com processador Intel® Core i7-2670QM CPU @ 2.20GHz com 4 núcleos físicos e memória de 16GB DDR3 1600MHz. O sistema operacional foi o Ubuntu 16.04 LTS (kernel 4.4.0-112-generic) e a linguagem de programação Java(TM) SE Runtime Environment, na versão 1.8.0-151.

Para medições de consumo de energia, foi utilizada a biblioteca `jRAPL` [Liu et al. 2015]. Esta biblioteca é uma interface para o módulo MSR (Machine-Specific Register), o qual está disponível para arquiteturas Intel que dão suporte a RAPL (Running Average Power Limit). Este módulo é responsável por armazenar informações relacionadas ao consumo de energia, as quais posteriormente podem ser acessadas pelo sistema operacional. Desta maneira, com o `jRAPL` é possível coletar de um trecho de código-fonte Java a dissipação de potência ($P$) ao longo do tempo ($t$). Através da relação entre $P$ (medido em watts) e $t$ (medido em segundos) temos o consumo de energia $E$ (medido em joules), ou seja, $E = P \times t$ [Pinto and Castor 2017].

Os experimentos foram realizados sem nenhuma outra carga de trabalho em execução simultânea no computador que realizou os experimentos (exceto processos do próprio sistema operacional). Uma vez que se necessita de tempo para que o Just-In-Time (JIT) compilador da Máquina Virtual Java (JVM) identifique as partes de código que podem ser otimizadas, a JVM, em particular, e VMs que utilizam JIT compiladores, em geral, tratam as primeiras execuções de um programa como *warmup*. Recentes estudos apontaram que a execução de um programa é mais lenta durante o período de *warmup* e com melhor desempenho nas execuções subsequentes ao *warmup* [Georges et al. 2007, Pinto et al. 2014]. Dessa forma, experimentos que utilizam linguagens de programação que se beneficiam de compiladores JIT devem descartar as execuções realizadas durante o período de *warmup* e analisar somente as execuções subsequentes. Neste trabalho, cada (micro-) benchmark foi executado 10 vezes ao longo de um mesmo processo de uma JVM; as três primeiras execuções são descartadas, enquanto a média das sete subsequentes é reportada.

## Resultados

Os resultados estão organizados em termos das perguntas de pesquisa.

**QP1: Qual o consumo de energia das APIs que implementam operações de E/S?**

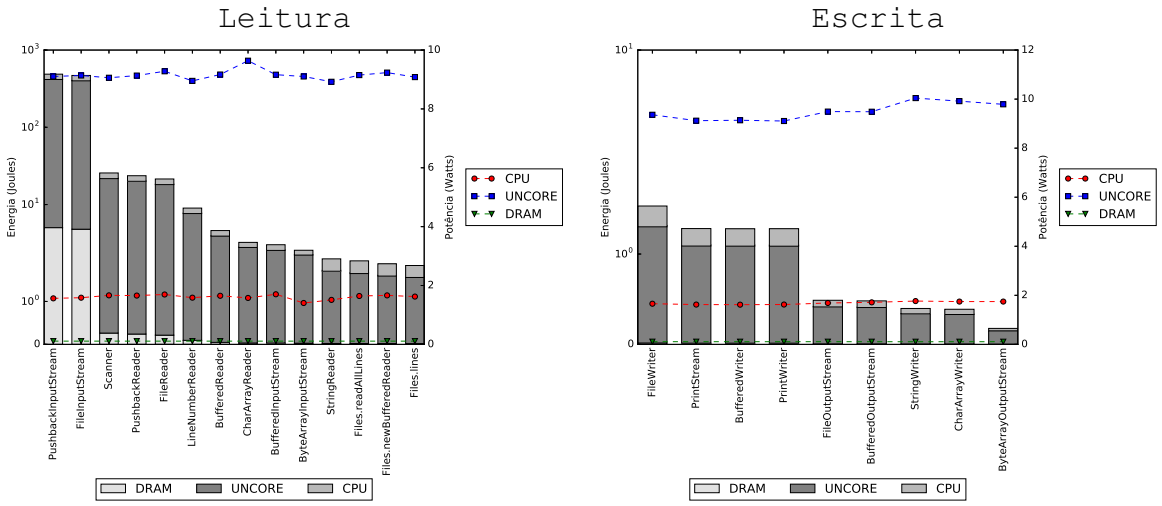


**Figura 1. Consumo de energia das APIs de E/S Java. Consumo de energia é apresentado em escala logarítmica.**

Na Figura 1, as barras representam consumo de energia, enquanto as linhas representam potência. O consumo de energia é a soma dos consumos de CPU, UNCORE (parte da CPU que não inclui os núcleos) e DRAM. À esquerda estão os micro-benchmarks que implementam operações de leitura, enquanto que os que implementam operações de escrita estão à direita. O micro-benchmark `PushbackInputStream` é o que apresenta maior consumo de energia dentre aqueles que realizam operações de leitura (492 joules consumidos), seguido de `FileInputStream` (474 joules). Por outro lado, `Files` é o micro-benchmark com melhor consumo energético usando os métodos: `lines` (1,86 joules), `newBufferedReader` (1,90 joules) e `readAllLines` (1,97 joules). O micro-benchmark `FileWriter` é o mais ineficiente do ponto de vista energético entre os que realizam operações de escrita (1,55 joules consumidos), seguido de `PrintStream` (1,30 joules) e `PrintWriter` (1,29 joules). `ByteArrayOutputStream`, por outro lado, é o que apresentou o menor consumo de energia (0,18 joules). Estes experimentos foram replicados com outras cargas de trabalho (1mb e 10mb) e os resultados se mostraram estáveis.

**QP2: As APIs que implementam operações de E/S mais frequentemente utilizadas são as que tem menor consumo de energia?**

A infraestrutura de código `BOA` [Dyer et al. 2013] foi utilizada como base para avaliar as ocorrências dos métodos que implementam E/S em projetos de código aberto. `BOA` conta com o código fonte de 7.830.023 projetos de código fonte aberto. Através de uma DSL, é possível navegar pela AST dos projetos, de forma a quantificar dados e meta-dados dos projetos armazenados. A Figura 2 contabiliza a utilização das classes estudadas.

Na figura abaixo, as linhas representam a quantidade de projetos que fazem ao menos um uso da classe, enquanto as barras contabilizam o total das ocorrências das classes Java. Ocorrências são medidas como o número de declarações de variáveis (tanto em escopo de classe quanto de método), além das assinaturas dos métodos; `imports` não

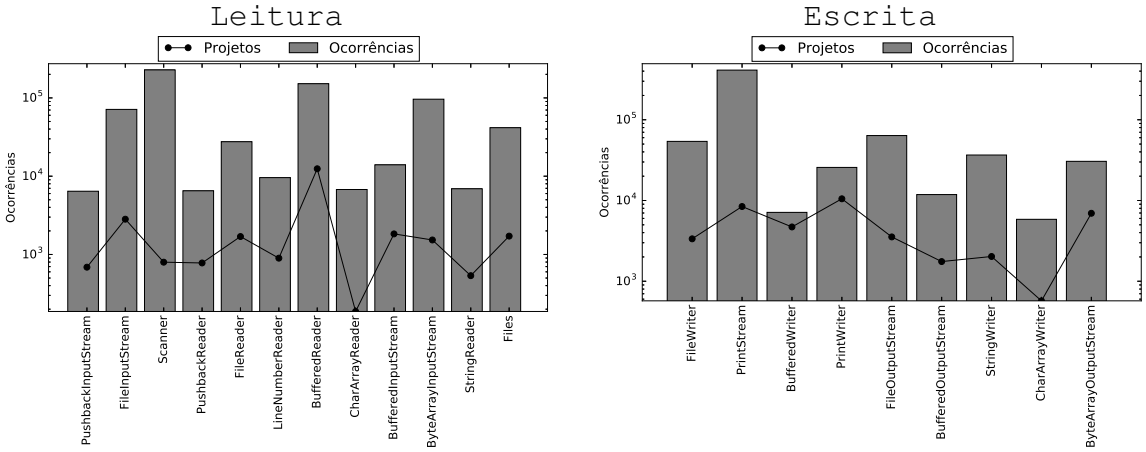


**Figura 2. Utilização dos Micro-benchmarks Java em projetos de código aberto.**

são levados em consideração. A classe `PrintStream` é a classe com mais ocorrências nos projetos estudados (412.163 ocorrências encontradas em 8.424 projetos diferentes), enquanto a classe `BufferedReader` é a mais utilizada em número absoluto de projetos de código aberto (12,441 projetos fazem 151.911 usos dessa classe).

A classe `Scanner` é a segunda mais utilizada (227.949 instancias) mas tem o quarto pior consumo de energia (26 joules). Dentre as classes mais energeticamente ineficientes que implementam operações de leitura, a `FileInputStream` é a mais utilizada, empregada em 71.339 instancias de 2.823 projetos de código aberto. Em contrapartida, a classe `FileWriter` é a terceira mais utilizada e apresentou o maior consumo de energia das que realizam operações de escrita.

### QP3: APIs intercambiáveis podem ser alternadas de forma a melhorar o consumo de energia de benchmarks não-triviais?

Com esta QP, queremos saber se é possível reduzir o consumo de energia de aplicações existentes simplesmente refatorando-as para que utilizem classes de E/S alternativas. A Figura 3 apresenta o consumo de energia dos benchmarks após a aplicação dos micro-benchmarks intercambiáveis. Por intercambiável, entende-se APIs que possam ser alternadas com esforço mínimo, por exemplo, subclasses de uma mesma classe abstrata.

O benchmark FASTA faz uso de um `OutputStream`, logo as opções de APIs intercambiáveis se restrigem a: `FileOutputStream` (F1), `ByteArrayOutputStream` (F2), `BufferedOutputStream` (F3) e `PrintStream` (F4). Na implementação original, o FASTA utiliza `System.out` para escrever os resultados na saída do terminal. Como pode-se observar na Figura 3, o melhor consumo de energia se manteve na versão padrão (F) do benchmark, enquanto a implementação utilizando `FileOutputStream` foi 26% menos eficiente em termos de consumo de energia. Já o K-NUCLEOTIDE faz o uso de um `BufferedReader` (K), que é uma das implementações da classe abstrata `Reader`. Para este benchmark, as classes `Scanner` (K1) e `LineNumberReader` (K2) ficam sendo as opções de APIs intercambiáveis, uma vez que estas são as únicas que apresentam o método `String readLines()` (as demais classes apresentam o `int read()`, que tem

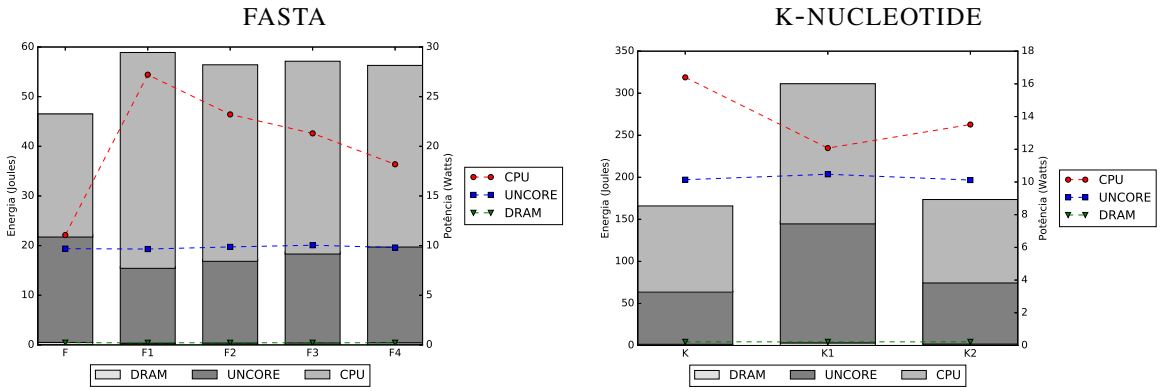


**Figura 3. Consumo de energia dos benchmarks após refatoração para utilizar outras classes de E/S.**

comportamento ligeiramente diferente). Como se pode observar na Figura 3, a versão padrão (K) apresentou o melhor consumo de energia, enquanto a versão (K1) apresentou um consumo de energia 87% maior, quando comparado ao baseline.

Esse resultado, onde as implementações padrão foram as mais eficientes, já era esperado, tendo em vista que os benchmarks do Computer Language Benchmarks Game são otimizados agressivamente para ter bom desempenho. Por outro lado, mostra que há oportunidades para se economizar energia com pouco esforço, já que aplicações que usem as classes menos eficientes podem ser refatoradas para usar alternativas menos custosas.

## Limitações

Este trabalho se limita a estudar as classes Java que residem no pacote `java.io`. Esse pacote contém classes que implementam métodos de E/S por padrão na linguagem Java. No entanto, não foram investigadas outras classes de comportamento similar que residem em outros pacotes, nem classes desenvolvidas por terceiros disponibilizadas em bibliotecas de software. Além disto, apesar de algumas variações de configuração terem sido objeto de estudo (como a variação do tamanho de entrada do arquivo), algumas das classes apresentadas contam com diversas opções de ajustes de configuração (como na presença de vários construtores). Outras variações dessas configurações podem ser exploradas em trabalho futuros. Uma outra ameaça está relacionado aos projetos armazenados na na infraestrutura BOA, que tiveram sua ultima atualização em 2015. Ademais, como o objetivo do trabalho é trazer uma caracterização inicial dos métodos que implementam operações de E/S, os benchmarks utilizados nesse trabalho são simples repetições dos métodos estudados, o que não necessariamente caracteriza o perfil de uma aplicação real em utilização. O estudo de como os achados se comportam em aplicações reais não foi contemplado neste trabalho, bem como o estudo aprofundado dos fatores que impactam um maior/menor consumo de energia.

## Trabalhos Relacionados

Existem estudos que concentram-se em ferramentas para estimar o consumo de energia em software [Liu et al. 2015], os quais apresentam ferramentas que permitem aos desenvolvedores de software estimar o consumo de energia de seu software sem a ne-

cessidade de um profundo conhecimento dos detalhes de mais baixo nível. Outros trabalhos analisam o comportamento energético em projetos de software através de estudos empíricos [Pinto et al. 2014, Lima et al. 2016, McIntosh et al. 2018]. Estes trabalhos apontam que pequenas mudanças podem introduzir grandes diferença em termos de consumo de energia e que refatorações simples, que fazem mudanças em tipos de dados podem ter bons resultados na eficiência energética de um software. Embora alguns estudos abordem eficiência energética em softwares que fazem uso intenso de dados [Bartenstein and Liu 2013, Liu et al. 2015], este trabalho se diferencia dos demais no seu foco nas APIs da liguagem Java que implementam operações de E/S.

## Conclusões

Este trabalho apresentou uma caracaterização de 23 classes que implementam operações de E/S na linguagem Java. Dentre os achados, destacam-se: (1) a significativa variação do consumo de energia entre as classes instrumentadas (enquanto a classe `Files` apresenta o menor consumo de energia, a classe `FileInputStream` apresenta consumo 3 vezes pior), (2) classes que são frequentemente utilizadas, como a classe `Scanner`, não necessariamente têm um bom comportamento em termos de consumo de energia e (3) quando classes intercambiáveis são empregadas em benchmarks não-triviais, o consumo de energia pode aumentar mais de 80%.

## Referências


[Barrett et al. 2017] Barrett, E., Bolz-Tereick, C. F., Killick, R., Mount, S., and Tratt, L. (2017). Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA):52:1–52:27.

[Bartenstein and Liu 2013] Bartenstein, T. W. and Liu, Y. D. (2013). Green streams for data-intensive software. In *ICSE*, pages 532–541.

[Dyer et al. 2013] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431.

[Georges et al. 2007] Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. In *OOPSLA*, pages 57–76.

[Lima et al. 2016] Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., and Fernandes, J. P. (2016). Haskell in green land: Analyzing the energy behavior of a purely functional language. In *SANER*, pages 517–528.

[Liu et al. 2015] Liu, K., Pinto, G., and Liu, Y. D. (2015). Data-oriented characterization of application-level energy optimization. In *FASE*.

[McIntosh et al. 2018] McIntosh, S., Hassan, A., and Hindle, A. (2018). What can android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering*.

[Oliveira et al. 2017] Oliveira, W., Oliveira, R., and Castor, F. (2017). A study on the energy consumption of android app development approaches. In *MSR*, pages 42–52.

[Pinto and Castor 2017] Pinto, G. and Castor, F. (2017). Energy efficiency: a new concern for application software developers. *Commun. ACM*, 60(12):68–75.

[Pinto et al. 2014] Pinto, G., Castor, F., and Liu, Y. D. (2014). Understanding energy behaviors of thread management constructs. In *OOPSLA*, pages 345–360.

[Sahin et al. 2016] Sahin, C., Wan, M., Tornquist, P., McKenna, R., Pearson, Z., Halfond, W. G. J., and Clause, J. (2016). How does code obfuscation impact energy usage? *Journal of Software: Evolution and Process*, 28(7):565–588.