UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Sobre a correlação entre a alocação de memória e o consumo de energia da memória RAM em Haskell

Leonardo Chaves Galdino de Moraes

RECIFE

2022

Leonardo Chaves Galdino de Moraes

Sobre a correlação entre a alocação de memória e o consumo de energia da memória RAM em Haskell

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação

Orientador: Marcio Lopes Cornelio

RECIFE

2022

Sobre a correlação entre a alocação de memória e o consumo de energia da memória RAM em Haskell

Leonardo Chaves Galdino de Moraes

BANCA EXAMINADORA

Prof. Dr. Juliano Manabu Iyoda

Centro de Informática / UFPE

Prof. Dr. Marcio Lopes Cornelio

Centro de Informática / UFPE

(**Orientador**)

Aos meus pais e irmão

Agradecimentos especiais

Gratidão profunda à minha mãe e ao meu pai por todo o incentivo e os mais diversos tipos de suporte durante meus estudos. Também sou especialmente grato à minha namorada pelo apoio durante o desenvolvimento deste trabalho e ao meu irmão por me ajudar a manter viva a minha curiosidade pelo conhecimento em nossas inúmeras conversas sobre ciência e pequenos projetos. Por último, agradeço ao corpo docente do Centro de Informática por sua excelência e profunda contribuição em minha formação.

Resumo

À medida em que a computação se torna cada vez mais ubíqua, o entendimento do consumo energético da execução de software ganha relevância na busca de uma sociedade mais sustentável. Pesquisas recentes avançaram o conhecimento do padrão de consumo de energia de softwares escritos na linguagem de interesse deste trabalho, Haskell, e também desenvolveram o ferramental necessário para novas análises. Nestas pesquisas, ficou claro que há uma forte correlação entre o tempo de execução de um programa e a energia consumida pelo processador e também pela unidade de memória, porém, não há nenhuma informação a respeito da alocação total de memória. Dessa forma, neste trabalho, foi estendido o *profiler* do GHC, de maneira a também coletar dados do consumo de energia pela unidade de memória. Junto com o dado de alocação total de memória, já provido pelo *profiler*, foi possível calcular a correlação entre essas duas variáveis, e, inesperadamente, não foi encontrada uma forte correlação entre elas.

Palavras-chave: Eficiência Energética. Consumo de Energia. Haskell. Programação Funcional. Análise de Desempenho.

ABSTRACT

As computers become ubiquitous, our understanding about energy consumption of software execution is increasingly relevant in order to achieve a sustainable society. Recent research pushed the understanding of Haskell programs' energy consumption footprint and developed the tooling necessary for further analysis. These research demonstrated a strong correlation between execution time and processor energy consumption, as well as execution time and memory unit energy consumption, but no information regarding memory allocation was provided. Therefore, in this project, GHC's profiler was extended so that it collects memory unit energy consumption. Along with total memory allocation data, already provided by the profiler, it was possible to compute the correlation between these two variables, and, unexpectedly, no strong correlation was found.

Keywords: Energy-Efficiency. Energy Consumption. Haskell. Concurrent Programming. Functional Programming. Performance Analysis.

Sumário

1. Introdução

1.1. Contexto e motivação

Enquanto a sustentabilidade é um problema importante para a sociedade, e, outras áreas da engenharia têm prestado atenção e avançado no tema, a Engenharia de Software, de maneira geral, não anda no mesmo ritmo [1]. A necessidade de avanços na computação sustentável é, ainda, exacerbada à medida que a computação se torna cada vez mais ubíqua.

A publicação do manifesto de Karlskrona [1] foi um passo importante para tornar a computação mais sustentável e, desde então, várias pesquisas na área têm sido publicadas [9]. Porém, ainda se faz necessário estudar o tema mais profundamente, especialmente porque, na Engenharia de Software, os desenvolvedores comumente não têm conhecimento pleno da eficiência energética dos softwares que desenvolvem, nem de boas práticas de codificação para a redução do consumo energético do software [3]. Além de entendimento, falta, também, o ferramental necessário para explorar quais seções do código precisam ser otimizadas em termos de consumo de energia [9, 16]. Por outro lado, não faltam ferramentas para analisar tempo de execução e uso memória, estando elas muitas vezes embutidas no próprio compilador, como no caso do GHC, um dos principais compiladores da linguagem Haskell [15].

O problema da eficiência energética de software vai além da sustentabilidade. No caso de tablets, smartphones, relógios e óculos inteligentes, em contraste com computadores pessoais e servidores, bateria é um recurso escasso [9]. Portanto, é do interesse dos usuários que o consumo de energia dos aplicativos utilizados seja o menor possível. Eficiência energética de software também é de importância para data centers. Por exemplo, em 2010, nos Estados Unidos, os data centers consumiram cerca de 2% de toda a energia usada no país [8].

1.2. Objetivos

Este trabalho tem como objetivo contribuir com o entendimento dos padrões de consumo de energia pela execução de software, especificamente daqueles escritos na linguagem funcional Haskell para o compilador GHC. Em mais detalhes, este trabalho visa estender o estudo feito por Lima et al. [11], no qual foi encontrado uma forte correlação entre o consumo de energia do processador e o tempo de execução, porém, neste trabalho, buscamos a perspectiva do consumo de energia pela memória. Para isso, foi necessário estender a seção de *profiling* do runtime do GHC, assim como feito por Lima et al. [11] para

coletar dados do consumo energético do processador e também pela unidade de memória. Isso possibilitará estudar como a memória se comporta em termos de consumo de energia.

2. Conceitos Básicos

Antes de começar a discutir trabalhos relacionados e os experimentos realizados, é importante estabelecer alguns fundamentos sobre os quais este trabalho se baseia. Esta seção, primeiramente, discute alguns aspectos da linguagem de programação de interesse, Haskell. Em seguida, discute sobre a origem dos dados de consumo de energia. E, por último, analisa o funcionamento do *profiler* do compilador utilizado e estendido. Esses tópicos devem se encaixar de maneira coerente com o resto do trabalho.

2.1. Haskell

Haskell é uma linguagem de programação funcional, lançada em 1990 [15]. Como uma linguagem funcional, seu foco são funções e aplicação de funções. Executar um programa escrito em Haskell significa avaliar expressões definidas no programa.

Uma das características mais importantes de Haskell é a avaliação preguiçosa, em contraste com avaliação estrita, que é mais popular entre outras linguagens de programação. A avaliação preguiçosa é o mecanismo usado para avaliar expressões e tornar a semântica da linguagem não estrita [15]. Essencialmente, esse mecanismo posterga a avaliação das sub-expressões que encontra, ao avaliar a expressão que as envolvem, até que seja absolutamente necessário, descartando a computação inteiramente caso não seja. Escrever o resultado de uma expressão no terminal, no sistema de arquivos, ou enviar à internet são todos exemplos de operações que forçam a avaliação das expressões envolvidas. Em suma, operações com efeitos colaterais.

O conceito de avaliação preguiçosa é de especial importância para este trabalho, pois, sem a devida intervenção durante o design dos experimentos, eles não seriam executados, já que num experimento de medição, como neste trabalho, a função a ser analisada é artificialmente invocada para incorrer os custos de sua avaliação, descartando, então, seus resultados. A solução desse problema será descrita na seção de metodologia.

2.2. RAPL

Os experimentos conduzidos neste trabalho fazem uso da interface RAPL [13] (Running Average Power Limit), presente em processadores recentes da Intel. Essa interface é capaz de controlar alguns aspectos, relacionados à energia, do processador. Porém, neste estudo, o foco é em sua capacidade de expor dados sobre o consumo de energia.

A interface relata dados em 4 domínios:

- PKG: energia consumida por um processador inteiro

- PP0: energia consumida consumida por todos os cores do processador e por suas memórias cache
- PP1: energia consumida pela placa de vídeo integrada com o processador
- DRAM: energia consumida por todas as unidades de memória RAM

Para atingir o objetivo deste trabalho, são relevantes, apenas, os domínios PKG e DRAM, logo, os domínios PP0 e PP1 serão desprezados.

Dados obtidos através dessa interface são estimativas modeladas em software e não medições analógicas da energia consumida. Apesar disso, vários estudos já comprovaram a acurácia das estimativas feitas por RAPL [18, 13]. A única ressalva é uma diferença constante entre a estimativa e a real energia consumida, mas que, por não variar entre experimentos em um mesmo ambiente, não compromete a comparabilidade dos resultados [13]. Dessa forma, a interface RAPL é adequada e conveniente para medir, em software, a energia consumida pelo processador e unidade de memória, sem a necessidade de instrumentar os equipamentos.

2.3. GHC *profiler*

GHC, o principal compilador de Haskell, inclui um *profiler* que pode ser usado para coletar estatísticas de tempo de execução e uso de memória [14]. O manual de usuário do GHC contém, desde versões mais recentes e amplamente disponíveis, um capítulo específico sobre essa ferramenta. Portanto, detalhes sobre seu uso e configuração serão, em maior parte, omitidos deste trabalho. Porém, dois tipos de configuração de *profiling* se destacam:

1. Relatório padrão: gera um relatório com o tempo de execução e alocação de memória de toda a execução e por centro de custo (ver Figura 1). Pode ser usado para apontar as expressões mais custosas e otimizar o trecho de código mais impactante.

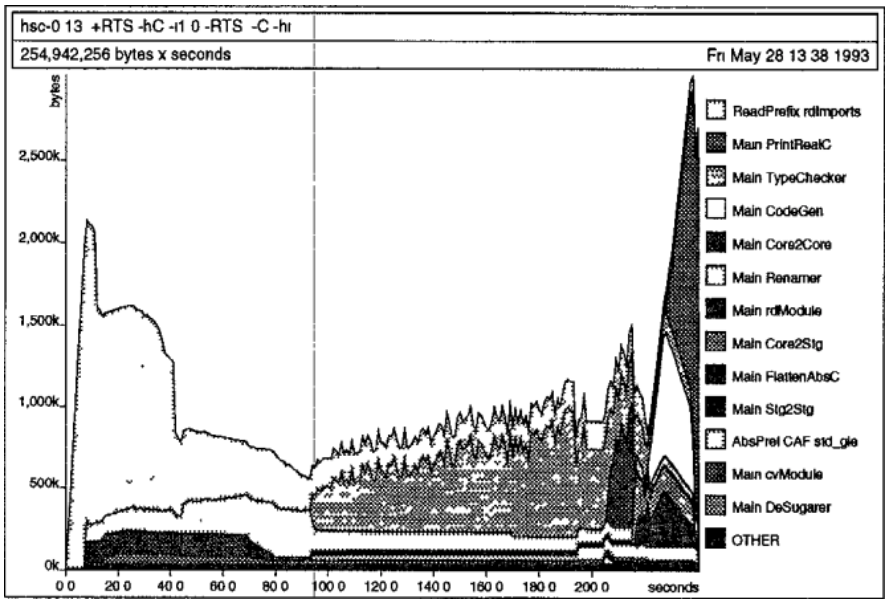
Figura 1: relatório padrão do *profiler* do GHC

```
Tue May 18 17:03 1993 Time and Allocation Profile
              (Lexical Scoping)

  hsc-0.13  +RTS -H25M -p -RTS  -C -hi ...

total time  =      240.48 secs (12024 ticks@20ms)
total alloc = 619,779,088 bytes
              51,846,277 closures

COST CENTRE      scc subcc  %time %alloc
TypeChecker        1     0   45.4   44.6
Renamer            1     0   25.0   27.0
builtinNames       1     0    7.6   14.4
PrintRealC         1     0    4.5    4.3
Core2Core          1     0    3.9    2.5
MAIN               1     1    2.7    2.7
CodeGen            1     0    1.5    1.1
rdModule           1     0    1.8    1.2
Stg2Stg            1     0    1.1    0.5
FlattenAbsC        1     0    0.7    0.5
cvModule           1     1    0.6    0.5
Core2Stg           1     0    0.6    0.3
  ...
```

Fonte: P. M. Sansom and S. L. Peyton Jones [14]

2. Relatório de heap: gera um relatório de alocação de memória em função da passagem do tempo por centro de custo (ver Figura 2). É útil para entender como o consumo de memória evolui durante a execução do programa.

Figura 2: Relatório de heap do *profiler* do GHC



Fonte: P. M. Sansom and S. L. Peyton Jones [14]

O relatório padrão supre todas as necessidades deste trabalho, portanto, ao falar de *profiling*, este será o tipo referenciado.

Como o *profiler* faz parte do runtime do programa, e, portanto, do binário gerado, quando compilado com a configuração de *profiling* ativada, uma rotina é executada regularmente para coletar estatísticas. A frequência padrão é de 20ms e, apesar de configurável, foi a utilizada neste trabalho.

Junto com o *profiler*, o GHC traz a noção de centro de custo, que é a unidade de atribuição de custos e está sempre associada a uma expressão. O usuário tem total controle sob quais expressões serão observadas. Isso pode ser feito manualmente ao diretamente marcar, com a diretiva apropriada, uma expressão no código, ou, pode ser feito automaticamente quando especificado durante a compilação, fazendo com que todas as funções nomeadas tenham um centro de custo associado. Centros de custos também são armazenados em *thunks*, expressões ainda não avaliadas, para que, quando finalmente avaliadas, seus custos sejam devidamente computados [14].

3. Trabalhos Relacionados

Essa seção discutirá trabalhos recentes e relacionados a este. Primeiramente, serão apresentados três trabalhos que focaram especificamente em programas escritos em Haskell. Em seguida, será discutido um quarto trabalho no qual Haskell é apenas um dos objetos de estudo.

Os trabalhos analisados nesta seção, assim como este, seguem um framework de experimentação em comum. Primeiramente, todos coletam estimativas de consumo de energia através da interface RAPL, descrita na seção anterior. Além disso, seus experimentos se baseiam em usar as estruturas de dados, providas pela biblioteca Edison [2], para executar as operações definidas na Tabela 1, inicialmente proposta para benchmarking de programas em Java [12]. Dessa forma, é possível fazer comparações consistentes, inclusive, entre diferentes linguagens de programação.

Tabela 1: Tabela de experimentação

iters	*operation*	*base*	*elems*
1	add	100000	100000
1000	addAll	100000	1000
1	clear	100000	n.a.
1000	contains	100000	1
5000	containsAll	100000	1000
1	iterator	100000	n.a.
10000	remove	100000	1
10	removeAll	100000	1000
10	retainAll	100000	1000
5000	toArray	100000	n.a.

Fonte: Leo Lewis. [12]

Cada trabalho estuda um determinado conjunto de estruturas de dados. Para cada estrutura, um experimento é executado para cada uma das operações listadas na Tabela 1. Um experimento consiste em aplicar uma das operações na estrutura de dados sob análise, com o número de elementos indicado na coluna 'base' e com uma estrutura auxiliar de tamanho indicado pela coluna 'elems', caso se aplique. A operação é, então, repetida o número de vezes indicado pela coluna 'iters'.

Como um primeiro passo, Lima et al. [11] estenderam duas ferramentas para análise de consumo de energia: GHC *profiler*, usado neste trabalho, e a biblioteca de benchmarking Criterion, usada nos demais. Então, com a versão estendida do Criterion, eles realizaram os experimentos descritos anteriormente, para várias estruturas de dados, e encontraram uma

forte correlação entre tempo de execução e consumo de energia no domínio PKG, i.e., pelo processador.

Melfe et al. [7] deram continuidade ao estudo, mas, desta vez, também coletando e analisando o consumo de energia da unidade de memória (domínio DRAM). Nesse trabalho, observaram uma forte correlação entre tempo de execução e consumo de energia da memória, além de que, para estruturas de dados de sequência, esse consumo contabiliza entre 18.3% e 29.4% do total da energia usada.

Em um subsequente estudo, Melfe et al. [6] compararam o consumo de energia, do processador e memória, entre implementações preguiçosas e rigorosas de estruturas de mapa. Além de confirmar as correlações já observadas nos trabalhos anteriores, notaram também uma forte correlação entre a energia consumida pelo processador e a energia consumida pela memória.

Além dos trabalhos discutidos anteriormente, que focam especificamente na linguagem Haskell, Pereira et al. [17] ranquearam diversas linguagens de programação, incluindo Haskell, em termos de consumo energético. Para tal, também usaram a interface RAPL, mas não utilizaram a biblioteca de benchmarking Criterion e nem a versão do GHC com *profiler* estendido. Também não usaram a tabela de experimentos mencionada anteriormente. No lugar, executaram diferentes tipos de algoritmos clássicos, como travessia de árvores, algoritmos genéticos, entre outros. Apesar das diferenças de abordagem, usando o método de correlação Spearman, encontraram, para Haskell, uma forte correlação, de 0.85, entre tempo de execução e energia total consumida (PKG+DRAM). Também encontraram uma moderada correlação, de 0.21, entre alocação de memória e energia total.

4. Metodologia

Para obter os resultados descritos na seção de experimentos, este trabalho seguiu a metodologia descrita a seguir. Além disso, também serão apresentadas as semelhanças e divergências entre a abordagem aqui tomada e a dos demais trabalhos.

Assim como nas pesquisas anteriores, a Tabela 1 será utilizada para definir os parâmetros dos experimentos. Também serão utilizadas as estruturas de dados presentes na biblioteca Edison. Dessa forma, cada experimento é definido pela execução de uma das operações listadas na tabela, com os parâmetros definidos na mesma, usando uma das estruturas de dados a ser estudada. Por fim, as estruturas são populadas com números inteiros, por simplicidade. Nesses aspectos, os experimentos aqui apresentados estão de acordo com os demais, anteriormente realizados.

A primeira divergência entre este trabalho e os demais é o conjunto de estruturas de dados analisado. Os trabalhos relacionados estudaram várias estruturas de dados entre três categorias: coleções, coleções associativas e sequências. Neste trabalho, por uma questão de tempo, apenas uma estrutura de dados foi usada para executar os experimentos, uma fila simples, SimpleQueue, da categoria de sequências.

4.1. Extensão do GHC

A interface RAPL também será utilizada como fonte de dados de consumo de energia, mas, apesar de a fonte dos dados de energia ser a mesma, o mecanismo utilizado para coletá-los é diferente. Esta é a segunda diferença entre este trabalho e os relacionados. Lima et al. [11] estenderam o *profiler* GHC de forma a coletar e relatar dados de consumo de energia por parte do processador. Essa extensão é útil para analisar o comportamento energético relativo ao processador, mas não da memória. Portanto, neste trabalho, foi necessário estender, pela segunda vez, o *profiler* do GHC para que também colete e relate os dados de energia relativos à memória. Portanto, o *profiler* do GHC fornecerá os dados de tempo, alocação de memória, energia do processador e energia da unidade memória, para cada centro de custo, como pode ser visto nas figuras 3, 4, 5, a seguir.

Figura 3: Dados de performance de cada centro de custo (herdados) no relatório padrão do GHC estendido

```
    inherited
%time %alloc %energy_pkg %energy_dram

100.0  100.0       100.0        100.0
100.0  100.0       100.0        100.0
  0.0    0.0         0.0          0.0
  0.0    0.0         0.0          0.0
  0.0    0.1         0.0          0.0
  0.0    0.1         0.0          0.0
  0.0    0.1         0.0          0.0
```

Fonte: este trabalho

Figura 4: Dados de performance de cada centro de custo (individual) no relatório padrão do GHC estendido

```
    individual
%time %alloc %energy_pkg %energy_dram

  0.0    0.0         0.0          0.0
  0.0    0.0         0.0          0.0
  0.0    0.0         0.0          0.0
  0.0    0.0         0.0          0.0
  0.0    0.0         0.0          0.0
  0.0    0.0         0.0          0.0
  0.0    0.0         0.0          0.0
```

Fonte: este trabalho

Figura 5: Cabeçalho do relatório padrão do GHC estendido

```
Thu Oct  6 18:09 2022 Time and Allocation Profiling Report  (Final)

   tg +RTS -p -RTS SimpleQueue ToList 5000 100000

total time   =       16.73 secs   (16730 ticks @ 1000 us, 1 processor)
total alloc  = 20,012,465,688 bytes  (excludes profiling overheads)
total energy pkg =      420.38 joules
total energy dram =       35.56 joules

COST CENTRE      MODULE                       %time %alloc %energy_pkg %energy_dram

strictWith       Data.Edison.Seq.ListSeq       58.5    0.0        58.1         59.5
toList           Data.Edison.Seq.SimpleQueue   28.5   99.9        28.8         27.9
toListExperiment DataStructures.SimpleQueue    12.9    0.0        13.0         12.5
```

Fonte: este trabalho

Note as colunas '%energy_dram', adicionadas neste trabalho. Esses dados serão, então, utilizados para computar as correlações encontradas nos trabalhos relacionados e compará-las. Além disso, com os dados coletados, será calculada a correlação Pearson entre a alocação de memória e energia consumida pela mesma, que até então não foi computada.

Como apenas a biblioteca Criterion, também estendida por Lima et al. [11], foi usada para coletar dados de tempo e energia nos trabalhos relacionados, a comparação entre os

resultados será de valor para entender a equivalência entre os dois métodos. Como a abordagem usada é diferente, não se pode esperar exatamente os mesmos resultados, mas é justo esperar que as correlações encontradas sejam, ao menos, próximas.

4.2. Compilação do GHC

Uma parte significativa do tempo investido neste trabalho foi devido à compilação do GHC. Recompilar o GHC foi necessário porque, como descrito na seção anterior, o código fonte do seu *profiler* foi alterado.

A versão do GHC estendida por Lima et al. [11] é a 7.11, de 2015, portanto, está bem desatualizada. A consequência disso é que vários problemas surgiram ao usar um sistema operacional recente. Foi necessário:

1. Compilar o compilador GCC, de C, com uma configuração diferente da padrão (*--disable-default-pie*), para que o binário gerado pelo código C (como o do *runtime system*) não seja independente de posição (PIC, *position independent code*). Isso é necessário porque versões antigas do GHC, como a que está sendo compilada, geram código dependente de posição, portanto, assim também deve estar configurado o GCC do sistema. Porém, versões recentes de sistemas operacionais, como o Ubuntu 22.04, trazem uma versão do GCC cuja configuração padrão é para gerar código independente de posição (PIC);
2. Adicionar a flag de compilação *-optc-Wno-error=expansion-to-defined* para que o warning *expansion-to-defined* não seja tratado como erro de compilação;
3. Reescrever as URLs dos submódulos do repositório do GHC usando URLs absolutas. No repositório padrão do GHC, as URLs dos submódulos estão definidas relativas a ele. Dessa forma, ao tentar compilar o GHC a partir de um repositório fork, os submódulos não são encontrados.

Também como resultado desse projeto, foi produzido um documento detalhando os passos necessários para realizar a compilação desta versão do GHC em sistemas operacionais modernos.

4.3. Mudanças no GHC

As mudanças no GHC, feitas neste trabalho, foram baseadas nas mudanças feitas por Lima et al. [11] e estão disponíveis publicamente no Github [5]. No total, foram adicionadas 57 linhas de código que são, em suma, responsáveis por:

1. Incluir o campo de *energy_dram* (energia da memória) às estruturas de centro de custo e pilha de centro de custo no arquivo *includes/rts/prof/CCS.h*.
2. Incluir o campo *inherited_energy_dram* (energia da memória) à estrutura pilha de centro de custo (guarda dados de uma expressão e suas subexpressões) no arquivo *includes/rts/prof/CCS.h*.
3. Adicionar uma variável global *total_energy_dram* ao arquivo *rts/Profiling.c* para guardar toda a energia consumida durante a execução do programa.
4. Ainda no mesmo arquivo, uma linha para somar a energia de cada centro de custo ao total de energia da pilha do centro de custo.
5. Várias linhas para escrever o novo dado coletado, de consumo de energia da memória, no relatório final. Também no arquivo *rts/Profiling.c*.
6. Por fim, no arquivo *rts/Proftimer.c*, adicionar uma variável global *last_energy_dram* para guardar a última leitura RAPL da quantidade de energia consumida pela memória. Útil para subtrair da nova leitura (mais uma linha de código para acessar a leitura do RAPL) e obter a quantidade de energia gasta no intervalo. Esse dado é, então, somado ao atual centro de custo com mais uma linha.

Resumidamente, é feita a leitura do RAPL do domínio da memória, calcula-se a diferença em relação à última leitura e, então, associa-se o resultado ao devido centro de custo. Por fim, ao terminar a execução do programa, os centros de custos e suas pilhas são lidas e usadas para gerar o relatório padrão final.

5. Experimentos e Análise

Uma vez estendido e recompilado o *profiler* do GHC, como descrito na seção anterior, é, então, possível realizar os experimentos e coletar os dados necessários. Nesta seção, será descrito como a avaliação dos experimentos foi forçada, para evitar que a avaliação preguiçosa ignorasse a execução dos experimentos. Em seguida, detalhes do ambiente no qual os experimentos foram feitos. Por último, serão apresentados os resultados. O repositório com código dos experimentos estão publicamente disponíveis no Github [4].

5.1. Avaliação forçada

Um aspecto importante da implementação dos experimentos é fazer com o que a avaliação preguiçosa não descarte a computação dos experimentos. Se não tomado o devido cuidado, isso pode acontecer porque o resultado da operação não é utilizado para nenhum fim prático, especialmente os resultados das operações intermediárias, já que, segundo a Tabela 1, algumas operações serão repetidas mais de uma vez.

Uma solução simples para este problema seria escrever, na saída do processo do experimento, a cada iteração, o resultado da operação. Essa abordagem não é interessante pelo simples fato de que os custos das operações de escrita poderiam ser incluídos na análise. Ainda que esse custo seja devidamente removido da análise, os experimentos consumiriam uma quantidade consideravelmente maior de recursos, como tempo de execução, especialmente porque as estruturas de dados são relativamente grandes (centenas de milhares de elementos).

A solução empregada consiste em utilizar a biblioteca DeepSeq, já incluso nos experimentos por parte da instalação da biblioteca Edison. Essa biblioteca introduz uma classe de tipos que é capaz de forçar a avaliação do tipo sendo manipulado, NFData (ver Código 1). Portanto, é necessário declarar a estrutura SimpleSeq como parte dessa classe de tipos, para que, quando aplicada à função deepseq, todos os elementos da estrutura, assim como as listas mantidas internamente, sejam avaliados.

Código 1: Forçando avaliação do experimento através de NFData

```
13  -- We need to force SimpleQueue evaluation to execute experiments N times
14  instance (NFData a) => NFData (SQ.Seq a) where
15      rnf s = SQ.strictWith rnf s `seq` ()
```

Fonte: este trabalho

Finalmente, basta aplicar a função deepseq ao resultado de cada operação para garantir que essa seja efetivamente executada.

5.2. Ambiente de experimentação

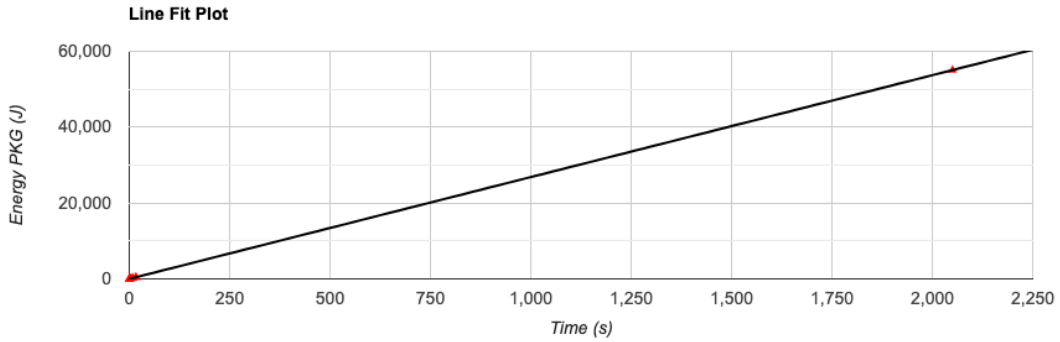
Um aspecto importante deste trabalho, e dos que a este se relacionam, é a difícil reprodutibilidade dos resultados dos experimentos, em razão das limitações da interface RAPL. Como descrito na seção de conceitos básicos, a interface provê dados em 4 domínios. Mas, do interesse deste trabalho, apenas dois são utilizados: de todos os processadores e da unidade de memória. Como esses são recursos compartilhados entre vários processos, e, durante a realização dos experimentos, muitos estão em execução, os resultados obtidos são naturalmente influenciados por eles, de maneira que duas execuções seguidas, de um mesmo experimento, podem levar a resultados levemente diferentes. Dito isso, assim como em outros trabalhos, e com o intuito de mitigar a interferência mencionada, os experimentos foram executados sem nenhuma carga extra no sistema.

A máquina utilizada roda o sistema operacional Ubuntu server 22.04 LTS (kernel 5.15.0-47-generic). Seu hardware consiste em um processador Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz com 8192 KB de cache em cada um de seus 8 núcleos e uma memória RAM de 32GB ECC DDR4 2666 MT/s.

5.3. Resultados

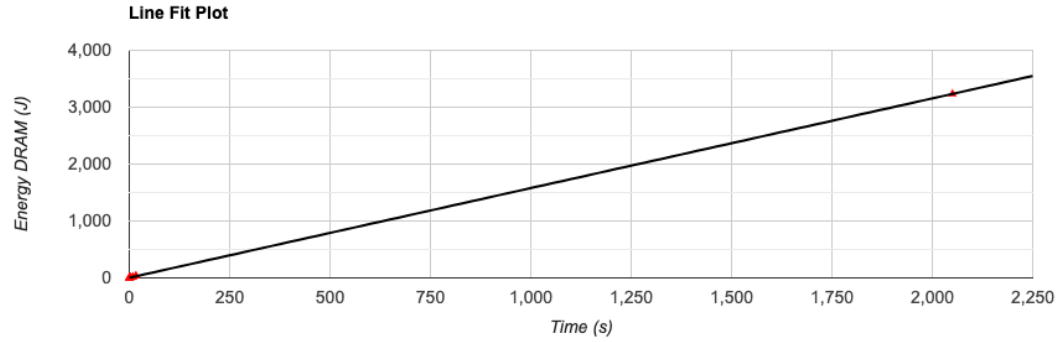
Após executar os experimentos, seguindo a metodologia descrita na seção anterior, é possível apresentar os gráficos que demonstram as correlações obtidas. Esta seção apresenta as correlações encontradas entre as variáveis analisadas nos trabalhos anteriores: tempo de execução e energia do processador (Figura 6), tempo de execução e energia da memória RAM (Figura 7), energia do processador e energia RAM (Figura 8). Finalmente, também apresenta o gráfico demonstrando a correlação de interesse maior deste trabalho: entre a alocação de memória e energia consumida pela memória (Figura 9). Todos os gráficos foram gerados a partir do mesmo dataset contendo os dados de tempo de execução, alocação de memória, consumo de energia do processador e consumo de energia da memória RAM coletados ao executar cada um dos experimentos definido pela Tabela 1.

Figura 6: Energia processador por tempo de execução, correlação de 0.9999998



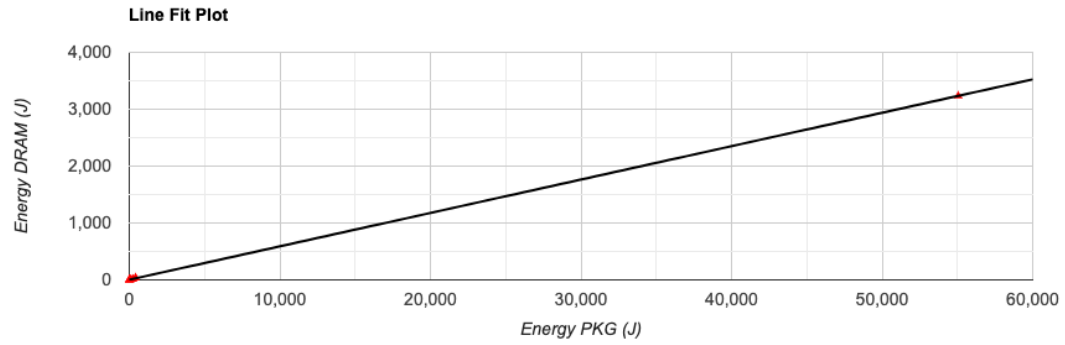
Fonte: este trabalho

Figura 7: Energia memória por tempo de execução, correlação de 0.999994



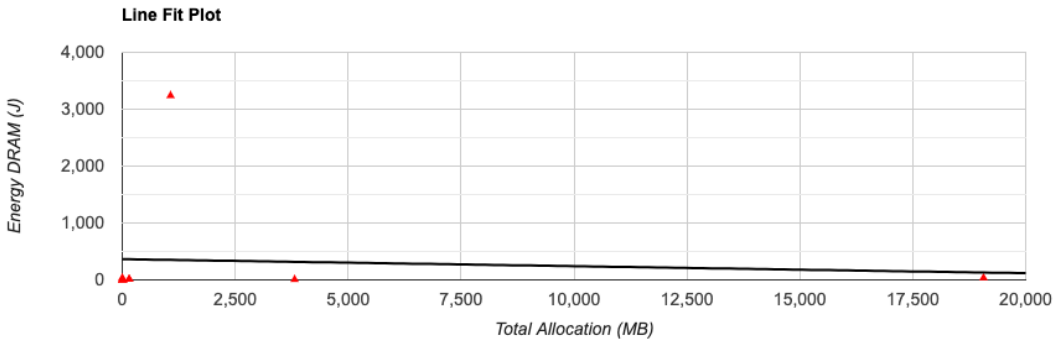
Fonte: este trabalho

Figura 8: Energia memória por energia processador, correlação de 0.999991



Fonte: este trabalho

Figura 9: Energia memória por alocação de memória, correlação de -0.07212



Fonte: este trabalho

6. Conclusões e trabalhos futuros

Com este trabalho, foi possível validar tanto o funcionamento do mecanismo de coleta de dados de consumo de energia pelo processador, introduzido por Lima et al. [11], quanto estendê-lo para também coletar o consumo de energia da unidade de memória. Além disso, também foram encontradas as fortes correlações entre tempo de execução e consumo de energia pelo processador, tempo de execução e consumo de energia pela memória e entre o consumo de energia pelo processador e o consumo de energia pela memória, constatadas previamente pelos trabalhos relacionados. Como os mesmos graus de correlação foram obtidos usando o *profiler* do GHC, ao invés da biblioteca Criterion, há uma confirmação de que as duas ferramentas são capazes de coletar os dados de maneira adequada.

Uma vez estabelecida a validade do mecanismo utilizado, também foi possível constatar uma nova informação. Como demonstrado na Figura 9, não há uma correlação relevante entre a alocação de memória e consumo de energia pela memória, o que está, relativamente, de acordo com os resultados encontrados por Pereira et al. [17], onde a correlação encontrada foi de 0.21, ou seja, fraca ou moderada. Por outro lado, existe uma correlação muito forte entre esse consumo de energia e tempo de execução. Uma possível explicação é que o consumo de energia pela unidade de memória acontece, em maior parte, devido às suas operações de leitura e escrita que ocorrem enquanto o processador executa o programa, o que é independente da quantidade de alocação.

6.1. Limitações

Apesar da falta de correlação observada nos resultados, é importante ressaltar que este trabalho, diferentemente dos relacionados, focou em apenas uma estrutura de dados: SimpleQueue, uma fila simples. Diferentes estruturas de dados, e, consequentemente, diferentes algoritmos utilizados para implementar as operações dos experimentos, podem acarretar em diferentes comportamentos de alocação de memória e consumo de energia. Portanto, os experimentos realizados não são suficientemente exaustivos para uma conclusão definitiva.

6.2. Trabalhos Futuros

Como mencionado na seção de limitações, apenas uma estrutura de dados foi utilizada para conduzir este trabalho. Dessa forma, futuras iterações, nas quais mais estruturas de dados são estudadas para calcular e confirmar a correlação de interesse, são importantes.

Bibliografia

[1] C. Becker et al., "Sustainability Design and Software: The Karlskrona Manifesto," 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 467-476, doi: 10.1109/ICSE.2015.179.

[2] C. Okasaki, “An overview of edison,” Electronic Notes in Theoretical Computer Science, vol. 41, no. 1, pp. 60–73, 2001.

[3] C. Pang, A. Hindle, B. Adams and A. E. Hassan, "What Do Programmers Know about Software Energy Consumption?," in IEEE Software, vol. 33, no. 3, pp. 83-89, May-June 2016, doi: 10.1109/MS.2015.83.

[4] Galdino, L. (2022). *BSc Thesis*. [online] GitHub. Available at: https://github.com/LeonardoGaldino/ThesisExperiments [Accessed 3 Nov. 2022].

[5] Galdino, L. (2022). *The Glasgow Haskell Compiler*. [online] GitHub. Available at: https://github.com/LeonardoGaldino/ghc [Accessed 3 Nov. 2022].

[6] Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. 2018. Evaluation of the impact on energy consumption of lazy versus strict evaluation of Haskell data-structures. In Proceedings of the XXII Brazilian Symposium on Programming Languages (SBLP '18). Association for Computing Machinery, New York, NY, USA, 83–89. https://doi.org/10.1145/3264637.3264648

[7] Gilberto Melfe, Alcides Fonseca, and João Paulo Fernandes. 2018. Helping Developers Write Energy Efficient Haskell Through a Data-structure Evaluation. In Proceedings of the 6th International Workshop on Green and Sustainable Software (GREENS ’18). ACM, 9–15

[8] Glanz J. Power, pollution and the internet. New York Times; Sept 23, 2012, Sect. A:1.

[9] Gustavo Pinto and Fernando Castor. 2017. Energy efficiency: a new concern for application software developers. Communications of the ACM 60, 12 (2017), 68–75.

[10] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna and C. Le, "RAPL: Memory power estimation and capping," 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED), 2010, pp. 189-194, doi: 10.1145/1840845.1840883.

[11] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe and J. P. Fernandes, "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language," 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2016, pp. 517-528, doi: 10.1109/SANER.2016.85.

[12] Leo Lewis. 2011. Java Collection Performance. (2011). https://dzone.com/articles/java-collection-performance

[13] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. SIGMETRICS Perform. Eval. Rev. 40, 3 (Jan. 2012), 13–17.

[14] P. M. Sansom and S. L. Peyton Jones, “Time and space profiling for non-strict, higher-order functional languages,” in Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1995, pp. 355–366

[15] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III). Association for Computing Machinery, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[16] PINTO, G.; CASTOR, F.; LIU, Y. D. Mining questions about software energy consumption. In: IEEE WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR ’14). Proceedings... New York, NY, USA: ACM, 2014. p. 22–31. ISBN 978-1-4503-2863-0.

[17] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, João Saraiva, Ranking programming languages by energy efficiency, Science

of Computer Programming, Volume 205, 2021, 102609, ISSN 0167-6423, https://doi.org/10.1016/j.scico.2021.102609.
[18] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 2016. A Validation of DRAM RAPL Power Measurements. In Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16). ACM, 455–470