Relatório - Laboratório 1
Lempel-Ziv

Lucas da Silva Moutinho - 15/0015747

[1]Universidade de Brasília - Instituto de Ciências Exatas
Departamento de Ciência da Computação - CIC 116394
Organização e Arquitetura de Computadores
2018.1 - Turma D - Professor Flávio Vidal
Prédio CIC/EST - Campus Universitário Darcy Ribeiro
Asa Norte 70919-970 Brasília, DF

lucas.silvamoutinho@gmail.com

Resumo. *Neste relatório tem-se por objetivo discutir e analisar criticamente um programa de codificação e decodificação construído a partir da linguagem Assembly MIPS, com o intuito de formar um espírito crítico de avaliação a respeito do desempenho real provido pelo sistema computacional. Para tal, fora desenvolvido o programa utilizando o algoritmo de compressão de dados Lempel-Ziv LZ78 por meio do simulador MARS. Benchmarks para testes do algoritmo e análises estatísticas foram utilizadas neste programa para um melhor embasamento na discussão pretendida.* ***Palavras-chave:*** *OAC. Arquitetura de Computadores. Lempel-Ziv. Compactação. UnB.*

1. Objetivos

Obter conhecimentos relacionados à linguagem Assembly MIPS e sua utilização por meio da implementação do algoritmo de Lempel-Ziv para compressão de dados em arquivos texto e analisá-lo criticamente em relação a performance e otimização, permitindo, assim, uma melhor compreensão de um sistema computacional.

2. Introdução

O algoritmo LZ78 consiste em um algoritmo de compressão sem perda de dados, isto é, uma compressão que permite que os dados originais sejam perfeitamente reconstruídos a partir a descompressão dos dados comprimidos.

Conhecido como uma das versões do algoritmo de Lempel-Ziv, fora publicado em artigos por Abraham Lempel e Jacob Ziv em 1978; e por isto recebeu o nome LZ78.

O Algoritmo LZ78 utiliza de um dicionário para armazenar sequência de caracteres encontradas no arquivo original. Por meio deste dicionário, permite a compressão substituindo ocorrência repetida de dados no arquivo por códigos que referenciam a posição correta da sequência de caracteres no dicionário.

Em termos matemáticos, pode-se dizer que o Dicionário pode ser representado por *D*, tal que:

$$D = [I, SC] \tag{1}$$

onde, *I* é o índice da linha do dicionário, e *SC* é uma sequência de caracteres.

Por outro lado, cada código no arquivo comprimido é um par *(I,c)*, onde *I* faz referência ao índice da linha do dicionário onde se encontra a sequência de caracteres que estaria naquela posição no arquivo original; enquanto que *c* é um caractere adicional após a sequência de caracteres.

Este algoritmo apresenta-se como uma ótima oportunidade, e desafio, para implementação de um código em baixo nível, permitindo a análise crítica de desempenho e otimização das instruções para a compressão de dados, principalmente no quesito de grandes arquivos.

Fora construído o algoritmo utilizando o simulador MARS em sua versão 4.5, permitindo assim, a utilização de estatísticas para analisar a quantidade de instruções no código e a influência destes para o processamento.

Ressalta-se, também, que foram disponibilizados benchmarks para análise de desempenho do algoritmo, estes que consistem em arquivos com uma quantidade grande de dados e que demandam de mais tempo para a compressão total.

Será, portanto, detalhado a implementação do algoritmo e debatida a sua performance em uma série de testes de desempenho, além, claro, da própria demonstração da compressão e descompressão.

3. Materiais e Métodos

- Mars 4.5
- Benchmarks data.txt disponibilizados pelo professor via plataforma moodle
- Livro de Organização e Arquitetura de Computadores Patterson
- Editor de texto Sublime

Será detalhado, primeiramente, a estratégia utilizada para a construção do algoritmo em Assembly MIPS.

O programa desenvolvido apresenta um menu simples de escolha como entrada para o usuário, permitindo, por meio de um input numérico, que este selecione entre utilizar o compressor ou fechar o programa.

Caso o usuário escolha utilizar o compressor, será requisitado um input com o nome do arquivo a ser lido. Fora construída uma lógica tal que por meio da sequência de caracteres escritas, o programa consiga ler a extensão e determinar se será realizada uma compressão ou descompressão. Caso o input tenha a extensão *.txt*, o programa criará um arquivo com o mesmo nome e a extensão *.lzw* que receberá os dados comprimidos. De maneira similar, caso o arquivo tenha a extensão *.lzw*, o programa criará um arquivo *.txt* para armazenar os dados descomprimidos. Tal lógica fora implementada por meio das chamadas de sistema *syscalls* do MIPS, tanto para receber o inputs quanto para a abertura, leitura e escrita dos arquivos de texto.

Explicando de maneira sucinta a lógica da compressão, esta fora implementada utilizando um buffer para armazenar uma sequência de caracteres lida do arquivo original. Caso a sequência de caracteres no buffer consista em uma ainda não presente no dicionário, o buffer é descarregado e salvo em uma nova linha do dicionário, ao mesmo tempo em que é escrito no arquivo de saída, o par comprimido, composto pelo último caractere lido juntamente do índice da sequência $SC = SCatual - c$, ou seja, o índice de uma sequência de caracteres igual a sequência atual do buffer sem o último caractere. Para separar os pares escritos na saída, fora escolhido, arbitrariamente, o caractere 96 da tabela ASCII, que corresponde ao símbolo *'*. Desta forma, é necessário considerar, principalmente para arquivos maiores, que a complexidade O do algoritmo cresce exponencialmente de acordo com o tamanho das sequências de caracteres e do número de linhas do dicionário. Fora escolhido para este programa que cada linha do dicionário teria capacidade de 32 words, 128 bytes, o que equivale a uma sequência de 128 caracteres em cada linha; além de 20000 linhas no total, o que implica na utilização de 640000 bytes de memória para o armazenamento do dicionário, o que resulta, assim, na utilização da memória do *.data* e do *.heap* do MARS. Optou-se pela utilização de mais linhas com menos words, para diminuir espaços inutilizados de caracteres nas sequências de linhas do dicionário na memória, evitando a fragmentação desta.

A descompressão, por sua vez, utiliza de maneira similar, um buffer de entrada para ler os pares do arquivo de entrada. Ressalta-se que não é necessário que o dicionário fique armazenado na memória, pois seguindo a lógica da descompressão, este dicionário é recriado e o arquivo original restaurado. Assim, cada leitura de um novo par implica na construção de uma nova linha no dicionário, contendo o caractere c e a sequência de caracteres do índice I no par (caso o índice seja 0, não existe uma sequência anterior no

dicionário). A cada leitura de par, é também recuperado os dados e escritos no arquivo de saída descomprimido, recuperando, assim, os dados originais.

Tendo atestado o funcionamento do programa, seguiu-se para a elaboração de uma lógica para os testes de desempenho, funcionamento e otimização a serem aplicados, este que será explicado a seguir.

Para averiguar o funcionamento, criou-se um data0.txt com diversos palíndromos, sendo este, o primeiro benchmark, com 26 Kbytes, tamanho consideravelmente menor do que os outros benchmarks. Por meio deste, é averiguado a correta compressão e descompressão dos dados, além de alguns testes iniciais de desempenho. Fora, também, comparado o tamanho do arquivo comprimido com o descomprimido, juntamente com outro benchmark, o data.txt.

Com o foco maior na análise de estatísticas das instruções, utilizando o *instruction statistics* do MARS, foram utilizados os outros arquivos textos, benchamarks, fornecidos pelo professor, com uma quantidade de dados bastante superior. Nestes o foco da discussão é mais a otimização e o tempo de execução do algoritmo. Para cada benchmark averiguado, fora colado as imagens do *statistics*.

4. Resultados

Apresentando os resultados do primeiro benchmark, este que fora utilizado para averiguar o desempenho e o correto funcionamento do algoritmo. Este arquivo contém uma série de palíndromos à serem comprimidos, seguem imagens de exemplo:

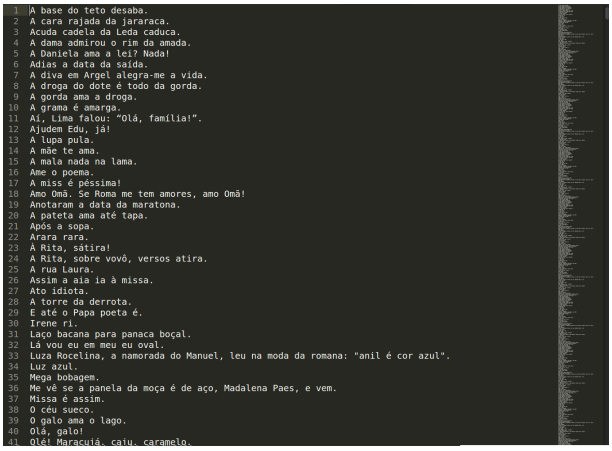

Figura 1. Arquivo com primeiras linhas do gerado data0.txt

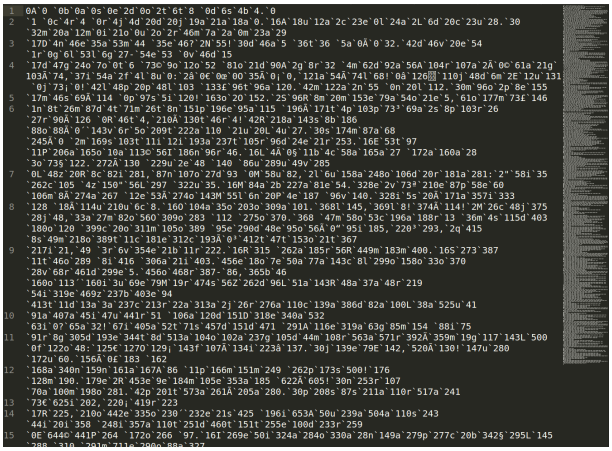

Figura 2. Arquivo com primeiras linhas do gerado data0.lzw

Figura 3. Arquivo com primeiras linhas do gerado Dicionario.txt

Fora comprimido o data0.txt utilizando o programa, este resultou em um data0.lzw codificado. Para averiguar o correto funcionamento, utilizou-se a descompressão no data0.lzw gerado. Verificou-se que o novo data0.txt gerado pela descompressão, de fato, continha os dados do arquivo original, e que estes foram recuperados corretamente.

A fim de se comparar o poder de compressão. Tabelou-se o tamanho em bytes do arquivo comprimido e descomprimido do data0. Para comparativos, fez-se o mesmo com um novo arquivo, o data.txt, que continha originalmente 40,1 Kbytes. Resultados encontram-se abaixo, juntamente com o tempo de execução para o término da compressão.

Quadro 1. Tabela de Arquivos e Tamanhos

Título	Tam. original do .txt	Tam. comprimido do .lzw	T. de Execução
data0.txt	26 kB	27,8 kB	9,37min
data.txt	40,1 kB	42,2 kB	15,12min

Observa-se que, mesmo a compressão e descompressão tendo ocorrido corretamente. O arquivo comprimido apresenta tamanho maior do que o original. Isto deve-se porque a quantidade de caracteres adicionais para o índice no arquivo comprimido e

também para o caractere de espaçamento entre os pares de saída não compensam, em média, a quantidade da sequência de caracteres que fora suprimida do arquivo original. Para arquivos maiores, o arquivo original aproxima-se do comprimido, o que pode implicar que para arquivos muito extensos, o número de caracteres substituídos compensa a compressão. Para melhorar esta compressão, poderia-se, também, pensar em outra forma de separar os pares de saída no arquivo comprimido, evitando assim a utilização de um byte para separar os pares.

Segue-se também estatísticas iniciais do *instruction statistics* do MARS para ambos arquivos. Tais estatísticas serão mais discutidas com os próximos benchamarks.

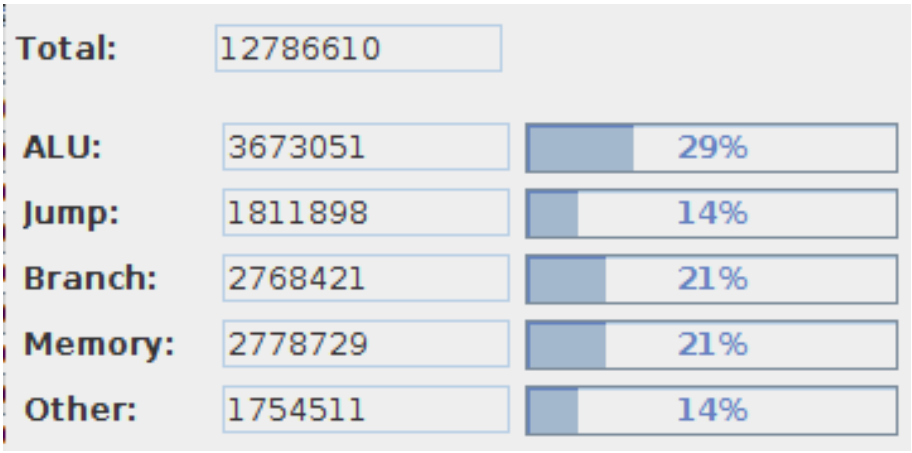

Figura 4. Tabela de Instruções para data0.txt

Por sua vez, agora foram utilizados os arquivos benchmarks providos pelo professor. Como estes possuem uma quantidade bem maior de bytes, o tempo de execução aumentou exponencialmente, impossibilitando a visualização do término destes. Segue-se as imagens do statistics enquanto o algoritmo estava sendo executado e havia estabilizado.

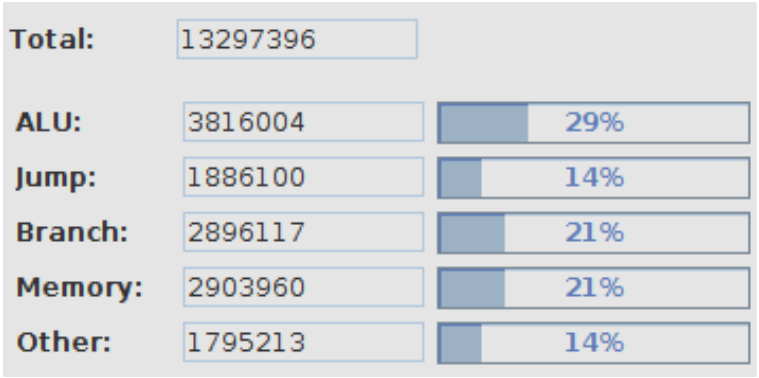

Figura 5. Tabela de Instruções para data1.txt

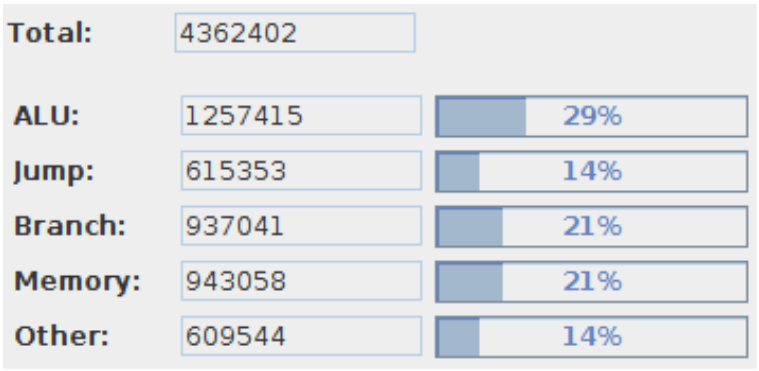

Figura 6. Tabela de Instruções para data2.txt

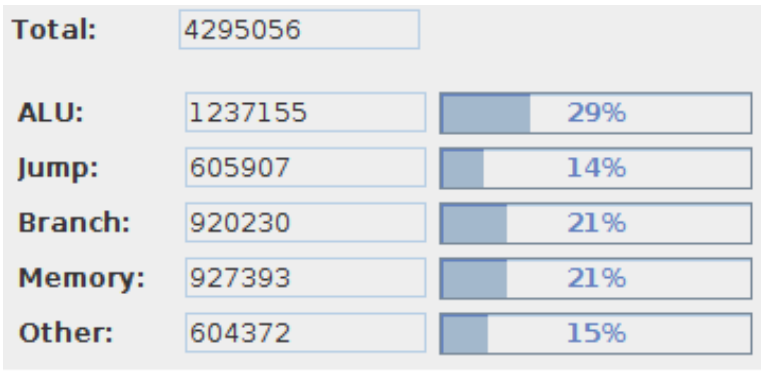

Figura 7. Tabela de Instruções para data3.txt

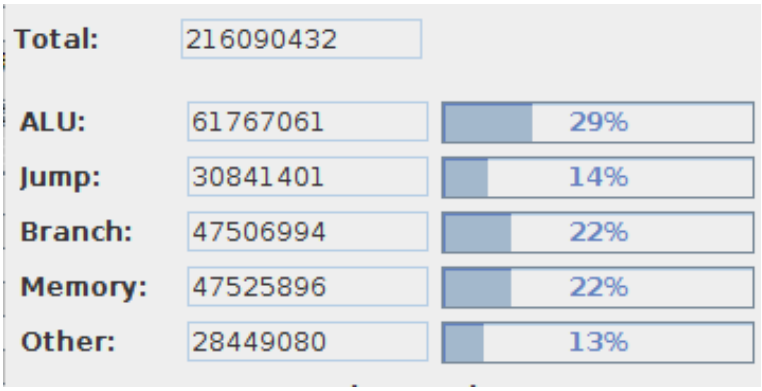

Figura 8. Tabela de Instruções para data4.txt

Pode-se observar que em média, o percentual de uso de cada uma das instruções aparentemente está equilibrado. Observa-se que operações aritméticas na ULA são as que tem um maior percentual de uso; provavelmente devido a constante necessidade dos offsets para navegar entre os dicionários e os buffers, além de outras operações aritméticas exigidas também pelas outras instruções. Este pode ser considerado um resultado esperado dado que todas as instruções, exceto alguns casos do jump, exigem algum tipo de

acesso ao processador para a realização de um cálculo. Observa-se, também, o alto acesso à memória para resgatar dados do dicionário, o que é um dos maiores motivos para o alto tempo de execução e um desempenho não tanto otimizado. Outro motivo para queda de desempenho seriam as instruções Other, que incluem os *syscalls*. Como acessos ao disco rígido são ainda mais impactantes para o tempo de execução, este provavelmente tem grande influência no desempenho.

Observa-se pela análise destas estatísticas que um provável código mais otimizado exigiria uma lógica para diminuir o número de acessos ao dicionário, memória, e coletas de bytes do arquivo texto por meio do acesso ao disco rígido. Uma estratégia poderia considerar ler mais bytes por meio de uma única leitura ao disco rígido, salvando-os em um buffer intermediário. Fica claro a necessidade de se pensar nestas otimizações, principalmente para arquivos mais extensos, e também, o poder de se trabalhar em baixo nível para a utilização de estratégias para aumentar o desempenho, principalmente diminuindo o acesso à memória e ao disco.

5. Discussão e Conclusões

Fora corretamente desenvolvido o programa de compressão e descompressão Lempel-Ziv na linguagem MIPS. Por meio do desenvolvimento deste programa, pode-se treinar e compreender diversos dos aspectos de um programa desenvolvido em baixo nível, permitindo a familiarização, principalmente, com a linguagem MIPS e com o simulador MARS.

A lógica do algoritmo fora corretamente implementada e, de fato, permite a compressão e descompressão de dados. Entretanto, por meio de testes comparativos dos tamanhos de arquivos comprimidos e descomprimidos, testes de tempo de execução, benchmarks para avaliação de desempenho e estatísticas do MARS, observou-se diversos motivos para concluir que o código não estava otimizado e poderia ter um melhor desempenho. Fica claro a necessidade de se avaliar estratégias para a construção de algoritmos e a possibilidade de melhoras de desempenho disponibilizadas por uma linguagem de baixo nível.

Referências

David A Patterson and John L Hennessy. Computer organization and design. *zadnje izdanje*, 1994.

Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Wikipedia. Lz78, a. URL `https://pt.wikipedia.org/wiki/LZ78`.

Wikipedia. Lossless compression, b. URL `https://en.wikipedia.org/wiki/Lossless_compression`.

Wikipedia. Lz77 and lz78, c. URL `https://en.wikipedia.org/wiki/LZ77_and_LZ78`.

University Academy. Lempel-ziv coding. URL `https://www.youtube.com/watch?v=EgreLYa-81Y`.