Universidade Federal de Ouro Preto - UFOP

Instituto de Ciências Exatas e Biológicas - ICEB

Departamento de Computação - DECOM

Ciência da Computação

Trabalho Prático I

BCC266 - Organização de Computadores

Caio Lucas Pereira da Silva, Vinicius Nunes dos Anjos, Thalles Felipe Rodrigues de Almeida Santos

Professor: Pedro Henrique Lopes Silva

Ouro Preto
16 de fevereiro de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Para este trabalho é necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é um simulador do mapeamento associativo para a troca de linhas entre as caches e a memória principal (RAM), o qual respeite o comportamento da hierarquia de memória presente em sistemas de computação modernos, permitindo a avaliação de diferentes políticas de gerenciamento de memória e a análise de seu impacto sobre o desempenho do sistema.

1.1 Especificações do problema

Necessário a criação de um modelo computacional que simule o comportamento da hierarquia de memória presente em sistemas de computação modernos. Esse modelo deve permitir a simulação do acesso à memória principal e à memória cache. Para isso, é necessário implementar funções em C que permitam a leitura e escrita de dados na memória, bem como funções que implementem as políticas de gerenciamento de memória na cache.

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

1.3 Ferramentas adicionais

Algumas ferramentas foram utilizadas para auxiliar no desenvolvimento, como:

- *Live Share*: ferramenta usada para *pair programming* à distância.
- *Valgrind*: ferramentas de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: AMD Ryzen 5-5500U.
- Memória RAM: 8GB.
- Sistema Operacional: Linux Pop_OS.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar o seguinte comando para rodar o arquivo *Makefile* disponível:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

- *-o*: para definir o arquivo de saída.
- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.

[1] VScode está disponível em https://code.visualstudio.com/
[2] Disponível em https://www.overleaf.com/

- -*Wall*: para mostrar todos os possível *warnings* do código.

- -*c*: para compilação do código e geração dos arquivos objetos.

Para a execução do programa basta digitar :

```
./exe < arquivo.in
```

Onde o arquivo de entrada contém o tamanho da matriz e seus valores.

2 Desenvolvimento

O desenvolvimento foi realizado utilizando da técnica de pair programming, onde todos os integrantes do grupo programaram e participaram ativamente do código ao mesmo tempo. O uso das ferramentas Live Share para o compartilhamento de código e do Discord para a comunicação em equipe foi o que viabilizou o uso da técnica citada.

2.1 Codificação

Para a codificação foram utilizados treze arquivos (7 ".c"e 6 ".h"), os quais já haviam sido disponibilizado pelo professor e foram preechidos corretamente para o funcionamento.

2.2 Políticas de substituição de cache

As políticas de substituição de cache são algoritmos que determinam qual bloco de dados deve ser removido do cache quando é necessário espaço para armazenar novos dados. Existem várias políticas de substituição de cache que podem ser usadas, cada uma com suas próprias vantagens e desvantagens em termos de desempenho e eficiência.

É importante escolher uma política de substituição de cache que seja adequada para a aplicação específica e as características do sistema. Uma política de substituição de cache mal escolhida pode levar a um desempenho ruim e uma baixa eficiência do cache.

```
1  int memoryCacheMapping(int address, Cache *cache){
2      int pos = 0;
3      printf("%d", cache->size);
4      switch (MAP_METHOD)
5      {
6      case 1:
7          // Least recently used: increment the times used for all lines, and
               return the line with the largest times used
8          for (int i = 0; i < cache->size; i++)
9          {
10             if (cache->lines[i].timesUsed > cache->lines[pos].timesUsed)
11             {
12                 pos = i;
13             }
14         }
15         cache->lines[pos].timesUsed++;
16         return pos;
17     case 2:
18         // Least frequently used: increment the times used for all lines, and
               return the line with the largest times used
19         for (int i = 0; i < cache->size; i++)
20         {
21             if (cache->lines[i].timesUsed < cache->lines[pos].timesUsed)
22             {
23                 pos = i;
24             }
25         }
26         cache->lines[pos].timesUsed++;
27         return pos;
28     default:
29         return address % cache->size;
30     }
31 }
```

Código 1: memoryCacheMapping

A função primeiro verifica a estratégia de mapeamento definida na constante "MAP_Method". Se o valor for 1, a estratégia usada é "least recently used"(LRU), ou seja, a linha com o menor

número de usos é escolhida para substituição. Se o valor for 2, a estratégia usada é "least frequently used"(LFU), ou seja, a linha com o menor número de acessos é escolhida para substituição. Caso contrário, a estratégia usada é a de mapeamento direto, onde o endereço é simplesmente mapeado para uma linha na cache usando o operador módulo (%).

- **LRU** - (Least Recently Used) é uma política de mapeamento de cache que tenta otimizar o uso da memória cache, removendo as entradas mais antigas que não foram usadas recentemente. Essa política de substituição de cache é baseada na ideia de que, se uma entrada não foi acessada por um longo tempo, é improvável que seja acessada novamente no futuro próximo.

  Na implementação do LRU, cada linha do cache possui um contador de tempo que é atualizado a cada acesso. Sempre que uma nova entrada é adicionada ao cache, o contador de tempo de todas as outras entradas é incrementado. Quando é necessário remover uma entrada para abrir espaço para uma nova, a entrada com o menor valor no contador de tempo é escolhida.

  Uma possível desvantagem do LRU é que ele requer um acompanhamento constante do tempo de uso de cada entrada no cache. Isso pode ser um desafio em sistemas de cache de alta velocidade com grande número de entradas. Além disso, a implementação do LRU pode ser mais complexa do que outras políticas de mapeamento de cache, o que pode impactar negativamente o desempenho do sistema em geral.

- **LFU** - (Least Frequently Used) é uma política de substituição de cache que seleciona a linha que foi usada menos vezes para ser substituída quando a cache está cheia e uma nova linha precisa ser inserida.

  Essa política de substituição de cache baseada em frequência pode ser útil em certos cenários, especialmente aqueles em que existem padrões de acesso variáveis. Em um sistema que apresenta esses padrões, é possível que um conjunto de dados seja muito utilizado por um período de tempo, e então não seja mais necessário por um longo período. O LFU pode ser útil nessas situações, pois pode ajudar a manter a cache atualizada com os dados mais relevantes.

  No entanto, o LFU tem algumas desvantagens. Por exemplo, se um determinado conjunto de dados for necessário apenas uma vez, ele pode permanecer na cache indefinidamente, mesmo que haja outras informações que possam ser mais úteis. Além disso, a implementação do LFU pode ser mais complexa que a do LRU, pois é necessário manter um registro do número de vezes que cada linha foi usada.

2.3 Algoritmo de busca

```
1  Line *MMUSearchOnMemories(Address add, Machine *machine)
2  {
3      int l1pos, l2pos, l3pos;
4      Line *cache1, *cache2, *cache3;
5      MemoryBlock *RAM = machine->ram.blocks;
6      // Strategy => write back
7      l1pos = memoryCacheMapping(add.block, &machine->l1);
8      l2pos = memoryCacheMapping(add.block, &machine->l2);
9      l3pos = memoryCacheMapping(add.block, &machine->l3);
10
11     cache1 = machine->l1.lines;
12     cache2 = machine->l2.lines;
13     cache3 = machine->l3.lines;
14     RAM = machine->ram.blocks;
15
16     // Direct memory map
17
18     if (cache1[l1pos].tag == add.block)
```

```
    {
        /* Block is in memory cache L1 */
        cache1[l1pos].cost = COST_ACCESS_L1;
        cache1[l1pos].cacheHit = 1;
    }
    else if (cache2[l2pos].tag == add.block)
    {
        /* Block is in memory cache L2 */
        cache2[l2pos].tag = add.block;
        cache2[l2pos].updated = false;
        cache2[l2pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2;
        cache2[l2pos].cacheHit = 2;

        updateMachineInfos(machine, &(cache2[l2pos]));
        return &(cache2[l2pos]);
    }
    else if (cache3[l3pos].tag == add.block)
    {
        /* Block is in memory cache L3 */
        cache3[l3pos].tag = add.block;
        cache3[l3pos].updated = false;
        cache3[l3pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2 + COST_ACCESS_L3;
        cache3[l3pos].cacheHit = 3;

        updateMachineInfos(machine, &(cache3[l3pos]));
        return &(cache3[l3pos]);
    }
    else
    {
        /* Block only in memory RAM, need to bring it to cache and manipulate
           the blocks */
        l2pos = memoryCacheMapping(cache1[l1pos].tag, &machine->l2); /* Need
           to check the position of the block that will leave the L1 */
        l3pos = memoryCacheMapping(cache2[l2pos].tag, &machine->l3); /* Need
           to check the position of the block that will leave the L1 */
        if (!canOnlyReplaceBlock(cache1[l1pos]))
        {
            /* The block on cache L1 cannot only be replaced, the memories
               must be updated */
            if (!canOnlyReplaceBlock(cache2[l2pos]))
            {
                /* The block on cache L2 cannot only be replaced, the memories
                    must be updated */

                if (!canOnlyReplaceBlock(cache3[l3pos]))
                    /* The block on cache L3 cannot only be replaced, the
                       memories must be updated */
                    RAM[cache3[l3pos].tag] = cache3[l3pos].block;
                cache3[l3pos] = cache2[l2pos];
            }
            cache2[l2pos] = cache1[l1pos];
        }
        cache1[l1pos].block = RAM[add.block];
        cache1[l1pos].tag = add.block;
        cache1[l1pos].updated = false;
        cache1[l1pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2 + COST_ACCESS_L3
           + COST_ACCESS_RAM;
        cache1[l1pos].cacheHit = 4;
    }
    updateMachineInfos(machine, &(cache1[l1pos]));
    return &(cache1[l1pos]);
```

```
}
```

Código 2: MMUSearchOnMemories

Esse código é uma implementação de um algoritmo de busca na memória em múltiplos níveis de cache. A função "MMUSearchOnMemories"recebe um endereço "add"e uma estrutura "Machine"que representa a máquina na qual a busca será realizada.

A função começa declarando algumas variáveis que serão usadas posteriormente. "l1pos", "l2pos"e "l3pos"são as posições do bloco de memória nos respectivos níveis de cache da máquina, enquanto "cache1", "cache2"e "cache3"são os ponteiros para os arrays de linhas desses níveis de cache. "RAM"é um ponteiro para o primeiro bloco da memória RAM da máquina.

A função usa a estratégia "write back", que consiste em escrever os dados na memória principal (RAM) apenas quando os blocos de cache forem substituídos.

O algoritmo segue a seguinte lógica:

1. Verificar se o bloco de memória está no cache L1. Se estiver, atualizar o custo da linha e o status de cacheHit para 1 e retornar a linha correspondente.
2. Se o bloco não estiver no L1, verificar se está no cache L2. Se estiver, atualizar o custo da linha e o status de cacheHit para 2, mover o bloco para o L1 e retornar a linha correspondente.
3. Se o bloco não estiver no L1 nem no L2, verificar se está no cache L3. Se estiver, atualizar o custo da linha e o status de cacheHit para 3, mover o bloco para o L2 e o L1 e retornar a linha correspondente.
4. Se o bloco não estiver em nenhum dos caches, o bloco precisa ser trazido da memória RAM. Um bloco de cache no L1 precisa ser substituído para que o bloco da RAM possa ser colocado nele. Se o bloco que será substituído também estiver no L2, ele precisará ser movido para o L3. Se o bloco que será substituído também estiver no L3, ele precisará ser movido para a RAM . Por fim, o bloco é carregado para o cache L1 e a linha correspondente é retornada.

A função chama a função updateMachineInfos para atualizar as informações da máquina (como o número de leituras e escritas na RAM) após cada operação de leitura ou escrita na memória.

2.4 Código main

```c
#include "cpu.h"
#include "generator.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main(int argc, char **argv)
{

    srand(1507); // Inicializacao da semente para os numeros aleatorios.

    int memoriesSize[4];
    Machine machine;
    Instruction *instructions;

    // mudar para fazer funcionar a leitura de arquivos
    if (strcmp(argv[1], "random") == 0)
    {
        if (argc != 6)
        {
```

```
            printf("Numero de argumentos invalidos! Sao 5.\n");
            printf("Linha de execucao: ./exe TIPO_INSTRUCAO [TAMANHO_RAM|
                ARQUIVO_DE_INSTRUCOES] TAMANHO_L1 TAMANHO_L2 TAMANHO_L3\n");
            printf("\tExemplo 1 de execucao: ./exe random 10 2 4 6\n");
            printf("\tExemplo 2 de execucao: ./exe file arquivo_de_instrucoes.
                txt\n");
            return 0;
        }

        memoriesSize[0] = atoi(argv[2]);
        instructions = generateRandomInstructions(memoriesSize[0]);
        memoriesSize[1] = atoi(argv[3]);
        memoriesSize[2] = atoi(argv[4]);
        memoriesSize[3] = atoi(argv[5]);
    }
    else if (strcmp(argv[1], "file") == 0)
    {
        instructions = readInstructions(argv[2], memoriesSize);
    }
    else
    {
        printf("Invalid option.\n");
        return 0;
    }

    printf("Starting machine...\n");
    start(&machine, instructions, memoriesSize);
    if (memoriesSize[0] < 20)
        printMemories(&machine);
    run(&machine);
    if (memoriesSize[0] < 20)
        printMemories(&machine);
    stop(&machine);
    printf("Stopping machine...\n");
    return 0;
}
```

Código 3: Main

O programa começa inicializando a semente do gerador de números aleatórios para um valor fixo (1507). Em seguida, ele declara um vetor de inteiros "memoriesSize"com tamanho 4, um objeto "machine"do tipo Machine e um ponteiro para Instruction chamado "instructions".

A partir daí, o programa verifica se o primeiro argumento passado na linha de comando é "random"ou "file". Se for "random", o programa verifica se há exatamente 6 argumentos na linha de comando (o primeiro é o nome do programa). Em seguida, o programa converte o segundo, terceiro, quarto e quinto argumentos em inteiros e armazena em "memoriesSize[0]", "memoriesSize[1]", "memoriesSize[2]"e "memoriesSize[3]", respectivamente. Então, o programa gera aleatoriamente as instruções de máquina a serem executadas e armazena o ponteiro para essas instruções em "instructions".

Se o primeiro argumento for "file", o programa lê as instruções de máquina a partir do arquivo cujo nome é passado como segundo argumento e armazena o ponteiro para essas instruções em "instructions".

Se o primeiro argumento não for nem "random"nem "file", o programa exibe uma mensagem de opção inválida.

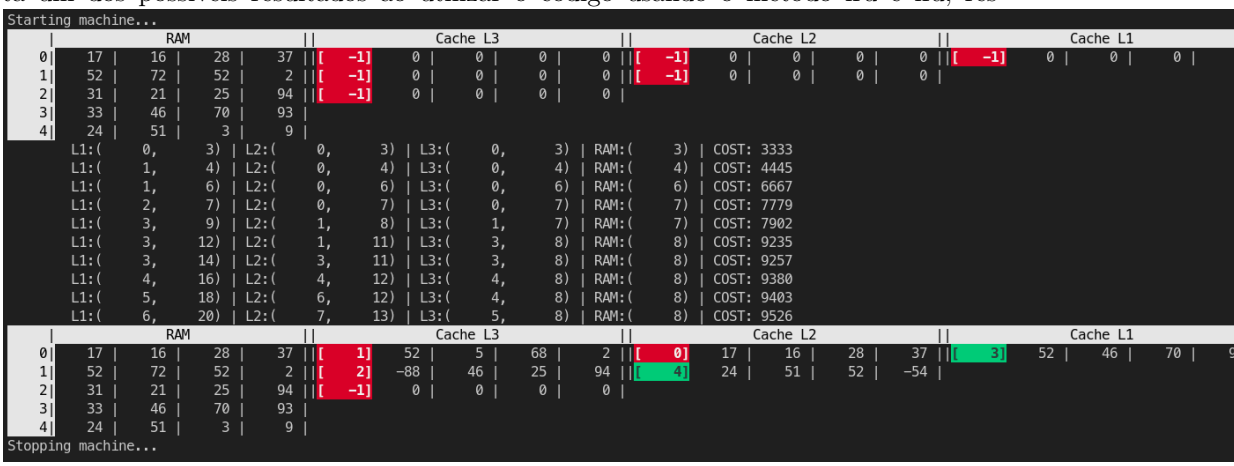
Em seguida, o programa exibe uma mensagem indicando que está iniciando a máquina, inicia a máquina chamando a função "start", passando como argumentos a máquina, as instruções e o vetor "memoriesSize". O programa verifica se o tamanho da memória principal é menor que 20 e, se for o caso, exibe o conteúdo das memórias. Em seguida, o programa executa as instruções de máquina chamando a função "run". Se o tamanho da memória principal for menor que 20, o conteúdo das memórias é exibido novamente. Finalmente, o programa interrompe a execução

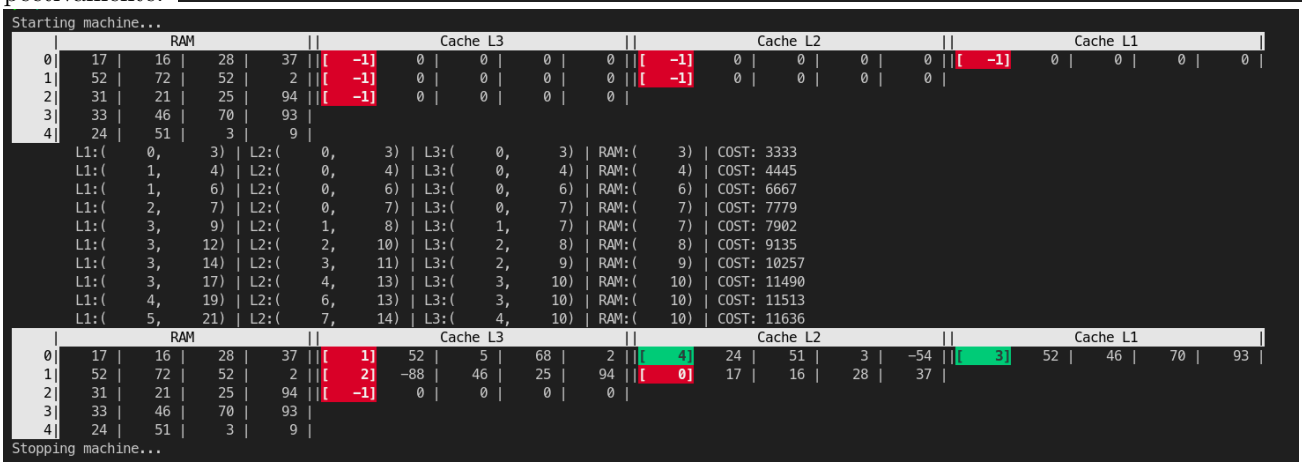
da máquina chamando a função "stop"e exibe uma mensagem indicando que está encerrando a máquina. O programa então retorna 0, encerrando a execução.

3 Conclusão

3.1 Resultado

A seguir está um dos possíveis resultados ao utilizar o código usando o método lru e lfu, respectivamente:





Também possui uma tabela mostrando um exemplo da taxa das caches e RAM usando cada método em dois exemplos para cada:

LFU	Cache 1	Cache 2	Cache 3	Taxa C1	Taxa C2	Taxa C3	Taxa RAM	Tempo
M1	1	2	3	20%	50%	40%	22%	11636
m2	2	4	6	18%	29%	13%	7%	16946

LRU	Cache 1	Cache 2	Cache 3	Taxa C1	Taxa C2	Taxa C3	Taxa RAM	Tempo
M1	1	2	3	30%	53%	62.5%	19	9526
M2	2	4	6	18%	37.5	6%	28%	16846

Podesse concluir, que a implementação das memórias e a hierarquização delas está no funcionamento correto, com a exibição do custo e dos valores de cada setor de RAM e das 3 Caches.