Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Trabalho Prático 2

BCC266 - Organização de Computadores I

Felipe Braz Marques
Matheus Peixoto Ribeiro Vieira
Pedro Henrique Rabelo Leão de Oliveira

Professor: Pedro Henrique Lopes Silva

Ouro Preto
15 de fevereiro de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

Para este trabalho é necessário entregar um código em C para simular o funcionamento da memória cache de um computador, acessando os níveis L1, L2, L3 e RAM.

1.1 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

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

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *GCC*: versão 11.3.0.
- *Valgrind*: versão 3.18.1.

1.3 Especificações da máquina

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

- Processador: Intel i5-9300H.
- Memória RAM: 16Gb.
- Sistema Operacional: Ubuntu 22.04.1 LTS.

1.4 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc -c cpu.c -Wall;
gcc -c generator.c -Wall;
gcc -c instruction.c -Wall;
gcc -c main.c -Wall;
gcc -c memory.c -Wall;
gcc -c mmu.c;
gcc cpu.o generator.o instruction.o main.o memory.o mmu.o -o exe -lm
```

Usou-se para a compilação as seguintes opções:

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

Para a execução do programa basta digitar:

```
./exe instrucao-desejada tam-RAM tam-L1 tam-L2 tam-L3
Exemplo: ./exe random 32 4 8 10
```

[1] Disponível em https://www.overleaf.com/

2 Desenvolvimento

2.1 constants.h

Este aquivo .h, contém tem todos os defines necessários. Como por exemplo o custo de acesso das caches, e da RAM. Por meio dele também é possível escolher o tipo de mapeamento que vai executar. Se for selecionado o mapeamento associativo ou associativo por conjunto, deve selecionar também o método que a máquina irá usar para substituir as informações nas caches.

2.2 Cache L3

Para a primeira parte do projeto, foi solicitado a implementação de uma simulação da cache L3. Dessa forma, muito do que já estava implementado para as caches L1 e L2 foram reaproveitados, como, por exemplo, nas funções de print de memória, onde só foi adicionado uma instrução para exibir a L3.

Na struct Machine, localizada no arquivo cpu.h, foi adicionado os item Cache l1, int hitL3 e int missL3, sendo, respectivamente, o tipo e o nome da variável. Assim, na função start() do cpu.c, os valores destas variaveis são inicializados, e a cache L3 tem o seu free na função stop().

Já no arquivo mmu.c, na função MMUSearchOnMemorys, foi adicionada a chamada do mapeamento para a possível posição de um bloco na L3 pela chamada da memoryCacheMapping. E para os if else que verificam os valores encontrados, foi duplicado o que estava para a cache L3, sendo, agora, alterado para adequar-se à cache L3.

Em relação ao else de quando não encontra-se os dados em nenhuma cache, nele há a instrução de levar o item que estava na L3 para a RAM.

2.3 Mapeamento associativo

O mapeamento associativo foi implementado dentro da função memoryCacheMapping, que percorre o vetor de memória de forma linear, procurando nas tags da cache um valor igual ao address passado. Caso encontre, retorna qual o indíce da iteração. Todavia, caso não encontre, retorna um valor padrão, no caso, o zero.

2.4 Mapeamento associativo por conjunto

O mapeamento associativo por conjunto também foi implementado na função memoryCacheMapping, mas para evitar interferencias entre codigos, ele só é ativo quando a constante "MAPEAMENTO_ASSOCIATIVO_POR_CONJUNTO"está definida no arquivo constants.h.

Assim, é chamada uma outra função, a mapeamentoAssociativoPorConjunto, que recebe os mesmos parâmetros da função anterior.

Dessa forma, primeiro verifica-se o tamanho da cache, e, caso seja menor que 12, é retornado o valor -1 e é feito o mapeamento associativo normal. Todavia, caso seja maior, é feito uma busca dividindo a cache em uma quantia definida por uma constante chamada DIVISOR e, a partir de um for aninhado, onde o exterior vai de zero até a quantidade de divisões possíveis, e o interior de 0 até a constante do divisor. É feita uma procura em todas essas partes pelo valor de tag desejado.

Caso a divisão não seja exata e a linha ainda não foi encontrada, é feito mais uma busca, mas dessa vez indo somente na parte em que ficou de fora, ou seja o resto da operação matemática.

```
1 int mapeamentoAssociativoPorConjunto(int address, Cache* cache){
2     if(cache->size < 12)
3         return -1;
4     const int DIVISOR = 4;
5     int divisoes = cache->size / DIVISOR;
6     int resto = cache->size % DIVISOR;
7     int posicaoProcura = 0;
8     for(int i = 0; i < divisoes; i++){
9         for(int j = 0; j < DIVISOR; j++){
```

```
            posicaoProcura = i + j * divisoes;
            if(address == cache->lines[posicaoProcura].tag)
                return posicaoProcura;
        }
    }
    for(int i = cache->size - resto; i < cache->size; i++){
        if(address == cache->lines[posicaoProcura].tag)
            return i;
    }
    return 0;
}
```

Código 1: Código do mapeamento associativo por conjunto

2.5 LFU

Para o método LFU (Least Frequently Used), foi criado um int contador na struct Line que receberá 0 para todas as linhas assim que a máquina for iniciada, a partir da chamada de inicializaContador() na start(). Em seguida, terá o seu valor incrementado em um toda vez que houver um cache hit naquela linha, contando, assim, quantas vezes este valor foi utilizado pelo sistema.

Dessa forma, seguindo o princípio do LFU de remover sempre o bloco que foi menos utilizado, assim que for solicitado um local para remover um item, com a chamada da função procurarBlocoASair(), no MMUSearchOnMemorys(), ocorrerá um laço de repetição que irá em todo o vetor de linhas à procura do menor valor do contador. Pois, ele incidirá no item que foi menos utilizado, podendo, assim, ser removido da cache.

Vale ressaltar que, caso haja alguma posição na ram com a tag -1, ela será a escolhida para ter o valor alterado.

2.6 LRU

O LRU (Last Recently Used), também utiliza o contador do inserido para o LFU, porém ele é incrementado sempre quando ocorre um cache hit, a partir da chamada da função adicionarMaisUmNoContador(), que irá adicionar mais um em todos os contadores da cache e irá colocar como zero o contador da posição em que foi encontrado o item, a fim de simbolizar que ele é o mais recente a ser utilizado.

Por conseguinte, quando é necessário remover um valor da cache a partir da chamada da função procurarBlocoASair(), é procurado o contador com o maior valor, pois ele indicará qual o item que está há mais tempo naquela cache sem ser utilizado.

Vale ressaltar que, caso haja alguma posição na ram com a tag -1, ela será a escolhida para ter o valor alterado.

2.7 FIFO

Já no método FIFO (First in First Out), também é utilizada a mesma variável, chamada de "contador", entretanto, partindo do conceito de que o primeiro a entrar na cache, será o primeiro a sair, todos os contadores são incrementados com mais 1 sempre que houver uma pesquisa de memória naquela cache. Dessa forma, quando é necessário remover um valor da cache para substituí-lo por outro utilizando a chamada da função procurarBlocoASair(), o item mais antigo que está ali, terá o maior valor em seu contador, representando ser o primeiro a ter entrado naquela cache, e , assim, ele será o primeiro da "fila"para ser removido.

2.8 RANDOM

Para o método random, o valor escolhido para ser removido é aleatório, assim, é chamada a função procurarBlocoASair(), que irá retorna um valor inteiro aleatório gerado a partir da semente definida na main e o tamanho da cache.

2.9 Disposição dos métodos no código

Todos os métodos foram implementados em um mesmo arquivo, e, agluns deles, compartilham as mesmas funções de código, como a de encontrar o maior contador para o LRU e o FIFO no arquivo mmu.c.

Todavia há alguns métodos que, caso sejam chamados, irão interferir no funcionamento do código em si. Dessa forma, faz-se útil a utilização do define no constants.h, pois é por lá que será escolhido o método a ser utilizado.

Tomando por exemplo a função memoryCacheMapping(), ela possui o mapeamento direto, disponibilizado pelo professor, mapeamento associativo e o mapeamento associativo por conjunto, mas o uso de cada um irá depender do que foi definido no arquivo constants.h, como pode ser visto no código da função abaixo:

```
1  int memoryCacheMapping(int address, Cache* cache) {
2      #ifdef MAPEAMENTO_DIRETO
3          return address % cache->size;
4      #endif
5      #if defined MAPEAMENTO_ASSOCIATIVO || defined
           MAPEAMENTO_ASSOCIATIVO_POR_CONJUNTO
6          #ifdef MAPEAMENTO_ASSOCIATIVO_POR_CONJUNTO
7              //Faz a procura por conjunto, dividindo o numero por algum valor.
                   Caso o tamanho da cache seja primo, faz o associativo normal
8              int posicao = mapeamentoAssociativoPorConjunto(address, cache);
9              if(posicao != -1)
10                 return posicao;
11         #endif
12         for(int i=0; i<cache->size; i++){
13             //Varre a cache, procurando a tag que contem o endereco desejado
14             if(address == cache->lines[i].tag){
15                 return i;
16             }
17         }
18         return 0;
19     #endif
20 }
```

Código 2: memoryCacheMapping

3 Análise dos resultados

Utilizando a geração aleatória de 100 instruções, L1 com tamanho 4, L2 com tamanho 8, L3 com tamanho 16 e RAM com tamanho 32, pode-se obter os seguintes resultados, utilizando tanto o mapeamento associativo quanto o associativo por conjunto:

Método	L1 Hit	L1 Miss	L2 Hit	L2 Miss	L3 Hit	L3 Miss	RAM Hit	Custo
LFU	39	223	60	163	50	113	113	131792
LRU	32	230	66	164	62	102	102	120962
FIFO	40	222	57	165	59	106	106	124982
Random	36	226	59	167	57	110	110	129222

Modificando a parte do código do "!can be improved?"para trocar os valores entre a L1 e a L2, para as mesmas entradas e quantidade de instruções, foi possível obter os seguintes resultados:

Método	L1 Hit	L1 Miss	L2 Hit	L2 Miss	L3 Hit	L3 Miss	RAM Hit	Custo
LFU	38	224	54	170	72	88	88	117502
LRU	32	230	63	167	63	104	104	123262
FIFO	33	229	63	166	63	103	103	122152
Random	39	223	59	164	48	116	116	134892

4 Impressões Gerais e Considerações Finais

Realizando o trabalho, foi possível entender mais sobre o conceito de memmory cache mapping, sobre o funcionamento das memórias caches ao lado da RAM, e sobre os métodos de troca chamados de LRU (Last Recently Used), LFU (Least Frequently Used) e FIFO (First In - First Out).

Em um primeiro momento, o grupo se deparou com uma certa dificuldade em entender como seria a implementação do que foi proposto, posto que a quantidade de arquivos disponibilizados era muito maior e possuia mais especificidades que no TP1. Todavia, após um tempo de discussão, análise do código, conversas com outros grupos e com o professor, foi possível compreender mais claramente a proposta do projeto e como ele seria feito.

Em geral, pode-se dizer que a parte que mais demandou tempo foi a compreensão do código, mas a implementação em si não foi muito complicada, principalemnte após o desenvolvimento de uma lógica para o LFU, o primeiro método que implementamos, pois os próximos foram muito similares.