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

Trabalho Prático 2

BCC 266 - Organização de computadores

Lucas Chagas, Pedro Morais, Nicolas Mendes
Professor: Pedro Henrique Lopes Silva

Ouro Preto
16 de fevereiro de 2023

Implementação

No desenvolvimento desse projeto, para atender o que está sendo solicitado nesse TP, implementamos, no código-fonte fornecido, as seguintes funcionalidades: **Criação da cache L3 e mapeamento associativo das memórias - juntamente com a criação das políticas LFU (*last frequent used*) , LRU (*last recent used*), FIFO (*first-in first-out*) e RANDOM para substituição de linhas de cache.**

Implementação da cache L3

Para implementar a cache L3 no código fonte fornecido, adicionamos uma variável, do tipo **Cache** (uma struct do código que contém um vetor do tipo **Line** [linhas da cache] e um int **size** para armazenar o tamanho da cache), chamada **l3,** na struct **Machine,** no arquivo **cpu.h.**

```
typedef struct {
    Instruction* instructions;
    RAM ram;
    Cache l1; // cache L1
    Cache l2; // cache L2
    Cache l3; //cache L3
    int hitL1, hitL2, hitL3,hitRAM;
    int missL1, missL2, missL3;
    int totalCost;
} Machine;
```

Imagem 1 - struct Machine

Posteriormente, após analisarmos o funcionamento do código, repetimos os mesmos processos que estavam estabelecidos para a **cache l1** e **cache l2.** Como a chamada da função **startCache()** e da **stopCache().** Além disso, o vetor com o tamanho das memórias (**memoriesSize**) ganhou uma posição a mais, o size da cache l3, e o usuário para executar o programa, via terminal, terá que passar um argumento a mais, que, também, corresponde ao tamanho da cache l3.

Além disso, foram criados int missL3 e o int hitL3 na struct Machine.

Mapeamento associativo das memórias

Para implementar o mapeamento associativo das memórias, foi adicionado um **for** dentro da função **memoryCacheMapping(),** no mmu.c, que percorre a cache em busca de uma linha que contém, no item **tag**, o endereço do bloco da RAM requisitado. Caso seja encontrado, ele retorna **a posição da linha no vetor**, senão, ele retorna a **posição 0**.

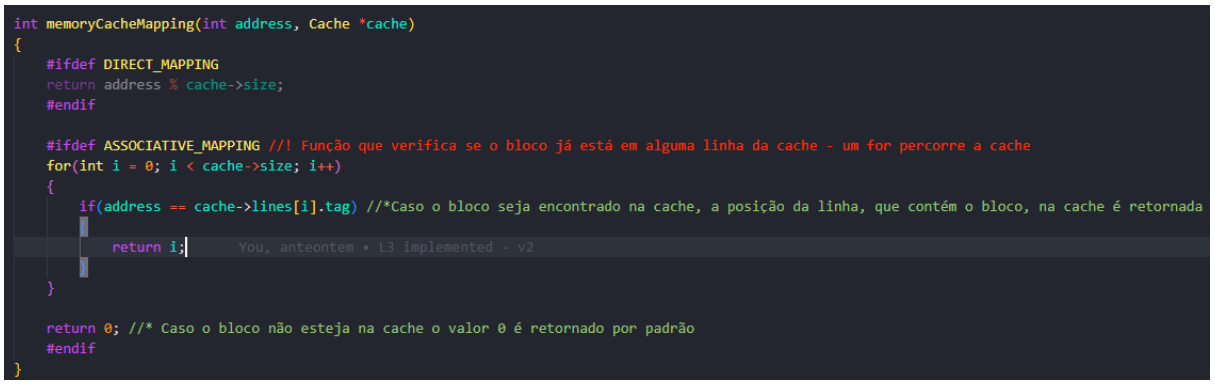

Imagem 2 - A função memoryCacheMapping

Políticas de substituição da linha

Para implementar os protocolos a seguir, foi adicionado um int **qtdVezesUsado** na struct **Line.** Conforme a política a ser utilizada, a interação com esse contador mudará.

LFU

Para implementar o LFU, implementamos a função **startCont(),** que contém um for que percorre toda a cache, zerando todos os contadores das Linhas. Essa função é chamada dentro da função **start(),** no cpu.c.

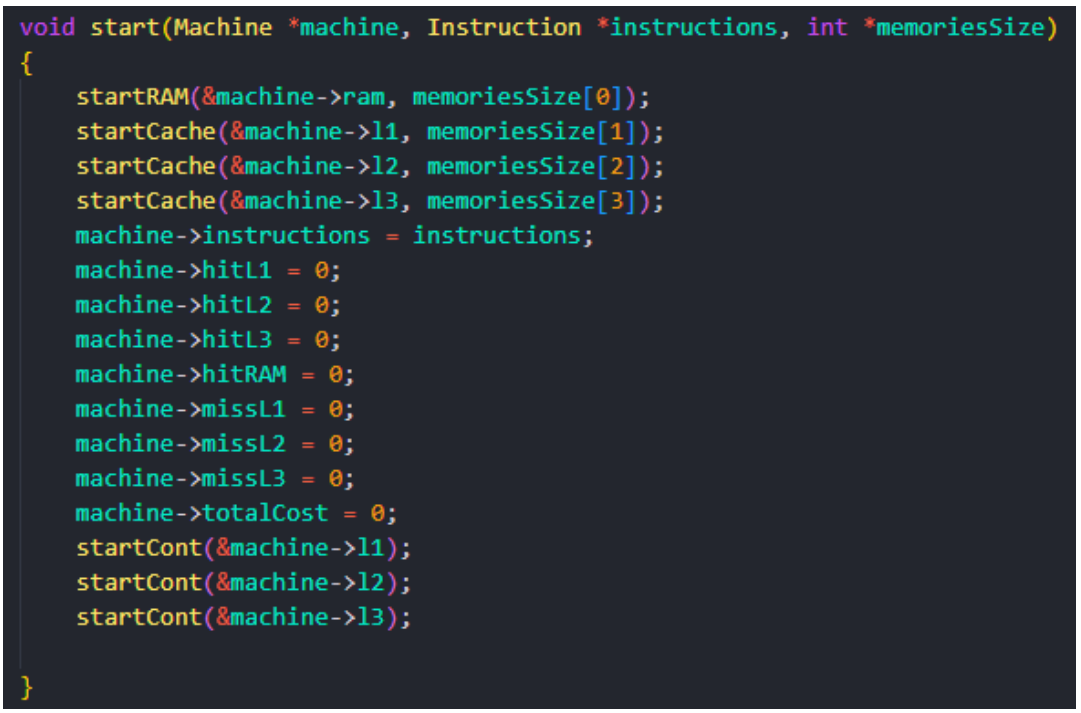

Imagem 3 - A função start

Para manter o controle sobre a **quantidade de vezes que uma linha é requisitada** o contador **qtdVezesUsado** é incrementando em uma unidade, toda vez que há um cacheHit, na função **MMUSearchOnMemorys()**, com tal linha ou toda vez que essa linha é solicitada em algum outro processo. Quando essa linha é movida para outra memória, o contador é zerado.

```
else if (cache2[l2pos].tag == add.block)
{
    /* Block is in memory cache L2 */
    // cache2[l2pos].updated = false; //Não foi atualizado
    cache2[l2pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2;
    cache2[l2pos].cacheHit = 2;
    #ifdef LFU
    cache2[l2pos].qtdVezesUsado += 1;
    #endif
```

Imagem 4 - Quando há um cacheHit de uma linha na cache l2, o contador é incrementado

Seguindo os critérios da política LFU, a linha que será substituída será a linha que tiver o menor valor no contador **qtdVezesUsado**. Para obter essa linha, há um for na função **blockWillBeRemoved()**, que irá comparar os contadores das linhas e retornar a posição do que estiver com o menor valor (linha a ser removida).

```
#ifdef LFU
//*Caso não tenha posições vazias, a posição o menos utilizado (método LFU) será retornado
for(int i = 1; i < cache->size; i++)
{
    if(cache->lines[i].qtdVezesUsado < lessUsed.qtdVezesUsado) //*O critério para decidir qual bloco será escolhido é o que foi menos utilizado (menor valor no contador)
    {
        lessUsed = cache->lines[i];
        pos = i;
    }
}

return pos;
#endif
```

Imagem 5 - trecho da função blockWillBeRemoved() para a política LFU.

LRU e FIFO

Assim como no LFU, a função **startCont()** é chamada no início da execução quando a política LRU ou a política FIFO está sendo usada, zerando todos os **qtdVezesUsado** das caches

Para manter o controle de qual linha da cache está sendo usada com frequência, foi implementada a função **atualizaLRUorFIFO(),** que incrementa o contador **qtdVezesUsado** de todas as linhas da cache que não possuem uma **tag = INVALID_ADD (-1) [uma posição vazia].**

Essa função é utilizada para ambos protocolos, entretanto, quando o protocolo do LRU está sendo utilizado, um trecho de código a mais é executado. Nesse trecho, o contador **qtdVezesUsado** da linha requisitada, cuja a posição foi passada por parâmetro, é zerado, indicando que essa linha foi usada recentemente.

```
#if defined LRU || defined FIFO
    void atualizaLRUorFIFO(Cache *cache, int pos){

        for (int i = 0; i < cache->size; i++)
        {
            if(cache->lines[i].tag != INVALID_ADD) //*Soma mais um em posições não-vazias, ou seja, block != -1
            {
                cache->lines[i].qtdVezesUsado += 1; //*Incrementa as posições não-vazias
            }
        }

        #ifdef LRU
        cache->lines[pos].qtdVezesUsado = 0; //*Zera o elemento que acabou de ser acessado. No LRU, ele vai remover a linha que tem o maior valor no contador, ou seja, que não foi ze
        #endif

    }
#endif
```

Imagem 6 - Função atualizaLRUorFIFO()

Além disso, a função **atualizaLRUorFIFO()** é executada toda vez que há uma busca nas memórias, no caso do uso da política FIFO, ou seja, quando a função **MMUSearchOnMemorys()** é chamada durante a execução. A mesma é executada toda vez que há um cachehit, no caso da política LRU.

```
Line *MMUSearchOnMemorys(Address add, Machine *machine)
{
    Line *cache1 = machine->l1.lines;
    Line *cache2 = machine->l2.lines;
    Line *cache3 = machine->l3.lines;

    //*Verifica se o bloco da RAM está em uma das linhas da cache (mapeamento associativo). Caso est
   int l1pos = memoryCacheMapping(add.block, &machine->l1);
   int l2pos = memoryCacheMapping(add.block, &machine->l2);
   int l3pos = memoryCacheMapping(add.block, &machine->l3);

//*Soma mais um em todos os elementos das caches, para preservar a diferença no tempo de entrada de
#ifdef FIFO
    atualizaLRUorFIFO(&machine->l1, l1pos);
    atualizaLRUorFIFO(&machine->l2, l2pos);
    atualizaLRUorFIFO(&machine->l3, l3pos);
#endif

    MemoryBlock *RAM = machine->ram.blocks;
```

Imagem 7 - chamada atualizaLRUorFIFO

Seguindo os critérios da política **LRU**, a linha que deverá ser substituída é a linha que foi utilizada por último recentemente. Seguindo a lógica descrita anteriormente da função **atualizaLRUorFIFO(),** que zera o contador **qtdVezesUsado** da linha que foi usada/requisitada recentemente e incrementa nas demais linhas com a tag diferente de INVALID_ADD, a linha que será substituída é a linha que tiver com o contador com o maior valor, ou seja, que faz muito tempo que não é utilizado. Para obter a posição dessa linha, é feito um for comparando os contadores das linhas da cache e retorna a posição dessa linha. Essa lógica é executada na função **blockWillBeRemoved().**

```
#if defined LRU || defined FIFO
//*O Bloco que será removido será o que tiver o maior número. No caso do LRU, corresponderia a linha que foi menos utilizada recentemente. No caso do FIFO, seria a primeira linha
for (int i = 1; i < cache->size; i++)
{
    if(cache->lines[i].qtdVezesUsado > lessUsed.qtdVezesUsado)
    {
        lessUsed = cache->lines[i];
        pos = i;
    }
}

return pos;
#endif
```

Imagem 8 - trecho da blockWillBeRemoved() para a política LRU

Seguindo os critérios da política **FIFO,** a linha que deverá ser substituída é a linha que está a mais tempo na cache (a primeira a entrar - *first in*). Seguindo a lógica descrita anteriormente da função **atualizaLRUorFIFO(),** o contador **qtdVezesUsado** é incrementado de todas as linhas que possuem a tag diferente de INVALID_ADD, as primeiras linhas a receberem um endereço da RAM terão os maiores valores nesse contador, logo, essas serão as linhas a serem substituídas. Para obter essa linha, há um for que irá comparar o valor deste contador de todas as linhas e retornará a posição da linha com o maior valor no contador. Essa lógica é executada na função **blockWillBeRemoved().**

```
#if defined LRU || defined FIFO
//*O Bloco que será removido será o que tiver o maior número. No caso do LRU, corresponderia a linha que foi menos utilizada recentemente. No caso do FIFO, seria a primeira linha
for (int i = 1; i < cache->size; i++)
{
    if(cache->lines[i].qtdVezesUsado > lessUsed.qtdVezesUsado)
    {
        lessUsed = cache->lines[i];
        pos = i;
    }
}

return pos;
#endif
```

Imagem 9 - trecho da blockWillBeRemoved() para a política FIFO (mesma da imagem 8)

RANDOM

Para a implementação da função RANDOM, é declarada uma variável que irá receber uma posição aleatória da cache para que ela seja substituída, adotando o critério da aleatoriedade. Esse trecho de código está dentro da **blockWillBeRemoved().**

```
#ifdef RANDOM
    int posRandom = rand() % cache->size;
    return posRandom;
#endif
```

Imagem 10 - trecho da função RANDOM.

OBS: Quando uma linha é substituída, o contador qtdVezesUsado é zerado, pela função zeraQtdVezesUsado().

Para definir a política a ser utilizada, utilizamos um sistema de define, que controla trechos de código a serem executados, conforme a política escolhida.

Na blockWillBeRemoved(), a prioridade são as linhas vazias, por isso, independentemente da política utilizada no mapeamento associativo, se tiver linha vazia, a posição dela será retornada.

MMUSearchOnMemorys()

Essa função é responsável pela busca de uma linha na memória. Ela recebe o endereço da RAM e a struct Machine, que contém todas as caches e a RAM. Inicialmente ela chama a função **memoryCacheMapping()** para todas as caches. Posteriormente, há uma cadeia de IFs que verificam se a linha, da posição retornada pelo mapeamento, está com a informação do endereço da RAM. Se isso acontecer, os cacheshits e cachesmiss são atualizados, juntamente com o custo de acesso da memória, pela **updateMachineInfos(),** e a linha é retornada por referência. Caso ela não esteja na L1, L2 e L3, a informação terá que ser transportada da RAM para L1, para isso ele chama a função **blockWillBeRemoved(),** que, de acordo com a política utilizada, estabelecerá os critérios para indicar qual posição será substituída. Feito isso, ele verifica se os níveis mais próximos são linhas vazias, o que facilitaria as movimentações, pois, como não há nenhuma informação, poderia-se somente substituir. No pior caso, onde deve-se fazer todas as movimentações nas memórias, ele vai substituindo todas as linhas e, ao final, traz da RAM para a L1, retornando essa última linha por referência.

Impressões Gerais

De modo geral, a implementação da cache L3, LFU (l*ast frequent used*),LRU (*last recent used*) e do FIFO(*first in,first out*) foi feito em trabalho em equipe,com cada membro dissertando idéias para encontrar a melhor resolução do problema em questão na execução das funções.

Em relação às coisas que mais agradaram os membros da equipe, destaca-se a grande quantidade de conceitos e noções aprofundados e revistos durante o processo de implementação, que agregaram e agregam fortemente para o decorrer da disciplina e do curso como manipulação de caches e da RAM.

No que tange aos assuntos que mais desagradaram os membros da equipe, foi a dificuldade inicial para entender o funcionamento das funções para a criação de uma lógica para implementar as propostas do trabalho. Entretanto, ao procurar o professor da disciplina e conversar com colegas de outros grupos para discutirmos eventuais soluções em termos de código, e assim conseguir fazer as funcionalidades.

Análise dos resultados

Após a implementação do método de mapeamento associativo e suas políticas: LFU (*last frequent used*), LRU (*last recent used*) e do FIFO (*first in,first out*), pudemos observar o custo de todos os métodos. Estes foram os resultados obtidos:

Número de instruções geradas para todos os testes: 200

Linha de execução para cada rodada de testes:
./exe file instrucao.in tamanho_l1 tamanho_l2 tamanho_l3

OBS:
Tamanho da RAM = 1000 (fornecido pelo arquivo instrucao.in).
TEXEC = Tempo de execução (unidade de tempo = custo de acesso as memórias / 100000)
CH = Cache Hit
CM = Cache Miss

Política adotada: LFU

TRAM	TC1	TC2	TC3	CH-C1 %	CH-C2 %	CH-C3 %	CM-C1 %	CM-C2%	CM-C3%	CH-RAM	TEXEC
M1	8	16	32	50,7	39,3	24	49,3	60,7	76	5554	64,3
M1	32	64	128	82,4	6,6	13,4	17,6	93,4	86,6	3471	39,4
M1	16	64	256	82,1	6,3	3,2	17,9	93,7	96,8	3076	35,5
M1	8	32	128	50,7	39,9	30,2	49,3	60,1	69,8	5047	59,1
M1	16	32	64	82,1	3,3	6,5	17,9	96,7	93,5	3927	44,1

Política adotada: LRU

TRAM	TC1	TC2	TC3	CH-C1 %	CH-C2 %	CH-C3 %	CM-C1 %	CM-C2%	CM-C3%	CH-RAM	TEXEC
M1	8	16	32	19,7	64,7	4,8	80,3	35,3	95,2	6582	74,9
M1	32	64	128	82,4	6,2	13,7	17,6	93,8	86,3	3484	39,5
M1	16	64	256	33	73,8	23,9	77	26,2	76,1	3255	38,7
M1	8	32	128	18,6	65,2	7,5	81,4	34,8	92,5	6400	73,1
M1	16	32	64	32	73,4	6,2	68	26,6	93,8	4136	47,67

Política adotada: FIFO

TRAM	TC1	TC2	TC3	CH-C1 %	CH-C2 %	CH-C3 %	CM-C1 %	CM-C2%	CM-C3%	CH-RAM	TEXEC
M1	8	16	32	24,6	15,2	37,8	73,4	84,8	63,2	9709	114,7
M1	32	64	128	39	25	62,6	61	75	27,4	4176	54,7
M1	16	64	256	32	14,5	69,6	68	85,5	30,4	4302	59,1
M1	8	32	128	24,8	10,8	51,8	75,2	89,2	48,2	7863	97,1
M1	16	32	64	37,7	20,4	51	62,3	79,6	49	5926	73,1

Política adotada: Random

TRAM	TC1	TC2	TC3	CH-C1 %	CH-C2 %	CH-C3 %	CM-C1 %	CM-C2%	CM-C3%	CH-RAM	TEXEC
M1	8	16	32	27,4	16,4	38,9	72,6	83,6	61,1	9043	107,3
M1	32	64	128	39,1	29,6	60	61,9	70,4	40	4172	53,9
M1	16	64	256	30,9	14,1	70,7	69,1	85,9	29,3	4232	58,7
M1	8	32	128	26,2	12	51,4	73,8	88	48,6	7595	93,6
M1	16	32	64	39,1	26,2	48,4	60,9	73,8	51,6	5809	71,1

Conclusão

Durante e após a realização do projetos, percebe-se diversos conhecimentos e conceitos que são adquiridos e reforçados da disciplina de BCC266, tendo em vista que no projeto utiliza-se muitos tópicos do gerenciamento de memória que são de extrema importância.

Com relação às dificuldades encontradas, foi identificado uma dificuldade em entender como o código disponibilizado funciona, entender como as funções, variáveis e ponteiros foi o que mais demandou tempo no trabalho.