

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

Trabalho 2

Organização de Computadores

Cecília Peret Paulino e Nicolle Bertolino
Professor: Pedro Silva

Ouro Preto
20 de julho de 2023

Sumário

0.1	Instruções de compilação e execução	1
1	Implementação	2
1.1	Criação da cache L3	2
1.2	Escolhendo o mapeamento	2
1.2.1	LRU:	2
1.2.2	LFU:	3
1.2.3	MMU SearchMemory	4
2	Impressões gerais:	6
3	Análise	6
4	Conclusão	7

Lista de Figuras

1	Enter Caption	6
---	-------------------------	---

Lista de Códigos Fonte

1	Inicializando a terceira cache	2
2	Lógica em LRU	2
3	Fazendo o método LFU	3
4	Fazendo o método LFU	3
5	Lógica em LRU	4
6	Lógica em LRU	5

0.1 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc *.c -o exe -Wall -g
```

Usou-se para a compilação as seguintes opções:

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.
- *-Wall*: para mostrar todos os possível *warnings* do código.

Para a execução do programa basta digitar:

Atribuindo tamanho para memória RAM, cache L1, cache L2, cache L3

```
./exe random 10 5 2 1
```

1 Implementação

1.1 Criação da cache L3

Foi criada uma nova cache no arquivo de cpu.c com base na cache L2 e L1 que já tinham sido criadas no programa.

```
1 printf("\x1b[0;30;47m      ");
2   printc("RAM", WORDS_SIZE * 8 + 3);
3   printc("Cache L3", WORDS_SIZE * 8 + 8);
4   printc("Cache L2", WORDS_SIZE * 8 + 8);
5   printc("Cache L1", WORDS_SIZE * 8 + 8);
6
7   printf("\x1b[0m\n");
8
9
10
11   for (int i=0; i<machine->ram.size; i++) {
12       printf("\x1b[0;30;47m%5d|\x1b[0m", i);
13       for (int j=0; j<WORDS_SIZE; j++)
14           printf(" %5d |", machine->ram.blocks[i].words[j]);
15
16       if (i < machine->l3.size) { // cache 3
17           printf("|");
18           printcolored(machine->l3.lines[i].tag, machine->l3.lines[i].
19               updated);
20           for (int j=0; j<WORDS_SIZE; j++)
21               printf(" %5d |", machine->l3.lines[i].block.words[j]);
```

Código 1: Inicializando a terceira cache

1.2 Escolhendo o mapeamento

No código, escolhemos o mapeamento associativo já que seria mais fácil de implementar, já que analisa todas as linhas da cache e verifica que bloco vai sair dela. Com isso, os dois algoritmos de substituição para esse mapeamento foi o LRU e LFU.

1.2.1 LRU:

Um algoritmo de substituição que serve para tirar o bloco da cache que está mais tempo sem uso e assim substituir por outra. Com isso, criamos um contador com o nome "cache->lines[i].tempoCache" que serve de contador para contar o tempo para cada linha da cache. Caso um bloco ou linha da cache tenha sido acessado ou marcado como hit, o contador reinicia para zero já que esse passou a ser usado. Além disso, criamos uma posição auxiliar que é inicializada com 0 (posição 0 da linha) e que incrementada à medida que fica muito tempo sem ser usada na cache. O laço de repetição serve justamente para analisar cada linha/bloco da cache e incrementa cada linha da cache a cada vez que ele não é usado. Se o contador de tempo de uma determinada linha for maior que o da posição 0 da linha cache, a posição escolhida passa a ser dessa linha específica. Por fim, retorna a posição com mais tempo sem ser usado na cache para ser tirada. Além disso, o tempo reinicia com 0 quando um bloco da cache é acessado, enquanto em outros blocos é incrementado.

```
1 int posicao = 0;
2   cache->lines[0].tempoCache++;
3
4   switch (metodos){
5       case LRU:
6
7
8       for (int i =0; i < cache->size; i++){
9           if (cache->lines[i].tag != INVALID_ADD){
```

```

11
12         cache->lines[i].tempoCache = cache->lines[i].tempoCache + 1;
13
14         if (cache->lines[i].tempoCache > cache->lines[posicao].
15             tempoCache){
16             posicao = i;
17         }
18
19         if (cache->lines[i].tag == address)
20             return i;
21
22     }
23
24 }
25
26 break;

```

Código 2: Lógica em LRU

1.2.2 LFU:

A técnica do LFU consiste em escolher a posição do bloco que é a menos usada. Por causa disso, criamos um contador dentro da struct Line chamada vezesDeUso que vai ser incrementada à medida que um determinado linha (que tem o bloco) da cache é acessado ou marcado como Hit. Com isso, fazemos as verificações similares ao LRU, se a linha da posição 0 for tiver mais uso que uma das linhas da cache. A nova posição vai receber a posição da linha que tiver sido menos usada.

```

1
2     case LFU:
3
4     for (int i =0; i < cache->size; i++){
5         if (cache->lines[i].vezesDeUso < cache->lines[posicao].vezesDeUso)
6             {
7                 posicao = i;
8             }
9
10        if (cache->lines[i].tag == address){
11            return i;
12        }
13
14    }
15
16    return posicao;
17    break;

```

Código 3: Fazendo o método LFU

Uma linha ou um dado quando é encontrado é como se ela estivesse sendo usada para acesso, por isso pensamos melhor colocarmos o contador dentro da função UptadeMachineInfos.

```

1
2     void updateMachineInfos(Machine* machine, WhereWasHit* whereWasHit,
3         Line *line, int cost) {
4     switch (*whereWasHit) {
5         case L1Hit: // se o hit for na l1, incrementa
6             machine->hitL1 += 1;
7             break;
8
9         case L2Hit:
10             machine->hitL2 += 1;
11             machine->missL1 += 1;

```

```

11         break;
12
13     case L3Hit:
14         machine->hitL3 += 1;
15         machine->missL1 += 1;
16         machine->missL2 += 1;
17         break;
18
19     case RAMHit:
20         machine->hitRAM += 1;
21         machine->missL1 += 1;
22         machine->missL2 += 1;
23         machine->missL3 += 1;
24         break;
25
26 }
27
28 line->tempoCache = 0;
29 line->vezesDeUso = line->vezesDeUso + 1;
30 machine->totalCost += cost;
31 }

```

Código 4: Fazendo o método LFU

1.2.3 MMU SearchMemory

Usamos a técnica do write back, se o bloco mudar, apenas a cache é alterada, a memória principal não. Inicializamos a função com 3 posições em cada cache por meio da função de Mapeamento Associativo. Com isso, fazemos as verificações em cada cache para encontrar o endereço do bloco.

Primeiro, verificamos se o bloco de memória (endereço) está na cache L1. Caso esteja, será contado o custo, marcado o cacheHit por meio da função UptadeMachineInfos e retornar a linha em L1 Segundo, caso o endereço do bloco esteja numa determinada posição em L2, trazer essa linha para L1, porém soma antes o custo que teve para acessar L1 e L2 (custo maior), além de marcar cacheHit em L2 e miss em L1 Terceiro, caso o endereço do bloco esteja numa determinada posição em L3, essa posição de L3 vai passar para L2 e retorna a linha correspondente. Além disso, marca cacheHit em L3 e miss em L1 e L2, seu custo acaba ficando maior que em L2 também.

```

1  if (cache1[l1posicao].tag == add.block) {
2
3
4      cost = COST_ACCESS_L1;
5      cache1[l1posicao].updated = false;
6      *whereWasHit = L1Hit;
7      cache1[l1posicao].vezesDeUso++;
8      updateMachineInfos(machine, whereWasHit, &(amp;cache1[l1posicao]), cost);
9
10
11     return &(cache1[l1posicao]);
12
13 }
14 else if (cache2[l2posicao].tag == add.block) {
15     /* Bloco est na memoria cache l2 (em cada linha)*/
16
17     cache2[l2posicao].tag = add.block;
18     cache2[l2posicao].updated = false;
19
20     cost = COST_ACCESS_L1 + COST_ACCESS_L2;
21     cache2[l2posicao].vezesDeUso++;
22
23
24     *whereWasHit = L2Hit;

```

```

25     updateMachineInfos(machine, whereWasHit, &(amp;cache3[l2posicao]), cost);
      // hit em l2
26     moveLine(cache2, l2posicao, cache1, l1posicao);
27     return &(cache2[l2posicao]);
28 }
29
30 else if (cache3[l3posicao].tag == add.block){
31
32     /*Bloco est na mem ria cache 3*/
33     cache3[l3posicao].tag = add.block;
34     cache3[l3posicao].updated = false;
35     cost = COST_ACCESS_L1 + COST_ACCESS_L2 + COST_ACCESS_L3;
36     *whereWasHit = L3Hit;
37     cache3[l3posicao].vezesDeUso++;
38     updateMachineInfos(machine, whereWasHit, &(amp;cache3[l3posicao]), cost);
39     moveLine(cache3, l3posicao, cache2, l2posicao);
40     moveLine(cache2, l2posicao, cache1, l1posicao);
41     return &(cache3[l3posicao]);
42
43 }

```

Código 5: Lógica em LRU

E por último, caso o bloco de memória não esteja em nenhuma das caches, é necessário olhar na memória RAM. Com isso, é necessário ter as verificações, já que pode ter momentos em que as caches estão cheias e precisam ser movidas por meio do mapeamento. Caso a não se pode colocar o bloco da RAM em L1 devido ao fato de que a posição está ocupada, é necessário que a posição de L1 saia e vá para L2. Isso ocorre tanto de L2 para L3 e a L3 para RAM, ocorrendo de maneira circular.

```

1  else {
2
3      if (canOnlyReplaceBlock(cache1[l1posicao]) == false) {
4          if (canOnlyReplaceBlock(cache2[l2posicao]) == false) {
5              if (canOnlyReplaceBlock(cache3[l3posicao]) == false){
6                  RAM[cache3[l3posicao].tag] = cache3[l3posicao].block;
7              }
8
9              cache3[l3posicao] = cache2[l2posicao];
10             cache3[l3posicao].tempoCache = 0;
11         }
12
13         cache2[l2posicao] = cache1[l1posicao];
14         cache2[l2posicao].tempoCache = 0;
15
16     }
17
18     cache1[l1posicao].block = RAM[add.block];
19     cache1[l1posicao].tag = add.block;
20     cache1[l1posicao].updated = false;
21     cost = COST_ACCESS_L1 + COST_ACCESS_L2 + COST_ACCESS_L3 +
          COST_ACCESS_RAM;
22     cache1[l1posicao].tempoCache = 0;
23     cache1[l1posicao].cacheHit = 4;
24
25
26
27 }
28
29 updateMachineInfos(machine, &(cache1[l1posicao]), cost);
30 return &(cache1[l1posicao]);

```

Código 6: Lógica em LRU

Tabela testando de forma randômica						
	Cache1	Cache 2	Cache 3	RAM	Instructions	
MRU1	1	2	3	10	10	
MRU2	3	4	5	9	8	
MRU3	6	8	9	10	5	
MRU4	4	5	6	5	6	

	Cache1Hit	Cache2Hit	Cache3Hit	RAMHit		
MRU1	0	3	1	10		
MRU2	5	0	0	17		
MRU3	10	0	0	12		
MRU4	11	0	0	11		

Figura 1: Enter Caption

2 Impressões gerais:

Tivemos muita dificuldade para entender o funcionamento do programa e o que era pedido no trabalho. A parte que tivemos mais dificuldade foi fazer a verificação caso o bloco de memória não esteja em nenhuma das caches, pois teríamos que fazer uma movimentação das linhas das três caches para colocar o bloco de memória da RAM em L1. Além disso, fizemos uma alteração na struct de WhereHasHit, pois estava dando erro na compilação do código e então criamos uma variável cacheHit dentro da struct Line e então aplicamos na função UptadeMachineInfos para fazer a verificação de onde ocorreu o hit.

3 Análise

Tempo de execução do programa: 0.00017s a 0.000023s Ordem de complexidade: $O(n)$

4 Conclusão

O trabalho foi bem complexo de realizar. Tiveos várias dificuldades na compilação e formatação das caches. Além disso, tivemos muita dificuldade para resolver o problema da divisão pois o programa só estava atribuindo valores aleatórios para as memórias, entretanto não ocorria as subtrações sucessivas e a máquina finalizava.

Referências