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

Máquina com Hierarquia de Memórias

BCC266 - Organização de Computadores

Bruno Alves Braga
Kailainy do Patrocínio
Kézia Alves Brito
Professor: Pedro Silva

Ouro Preto
16 de fevereiro de 2023

Sumário

Lista de Figuras

1 Introdução

Para este trabalho, é necessário entregar o código implementado em qualquer linguagem de programação citada (C, C++, Java ou Python) e o relatório em questão, referente ao que foi desenvolvido. O algoritmo a ser programado é uma Máquina com Hierarquia de Memórias.

1.1 Especificações do problema

O objetivo deste trabalho é codificar uma máquina com hierarquia de memória, possuindo um sistema de memória cache em três níveis. A máquina já foi pré-programada com dois níveis de cache e o método de mapeamento direto, bastando implementar o terceiro e último nível e dois métodos de mapeamento adicionais.

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: *Visual Studio Code + LiveShare*.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: *Visual Studio Code* + LaTeX *Workshop*.

1.3 Especificações da máquina

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

- Processador: Intel Core i5.
- Memória RAM: 8 GB.
- Sistema Operacional: Manjaro Linux.

1.4 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

- *-c*: para compilar o arquivo sem vincular os arquivos do tipo objeto.
- *-Wall*: para mostrar todos os possíveis *warnings* do código.
- *-o*: para vincular os arquivos do tipo objeto.
- *-lm*: para *linkar* a biblioteca *math*.

Para a execução do programa, basta digitar:

```
./exe random tamanho_da_ram tamanho_da_cache_1 tamanho_da_cache_2 tamanho_da_cache_3
./exe arquivo_de_entrada tamanho_da_cache_1 tamanho_da_cache_2 tamanho_da_cache_3
```

2 Implementação

2.1 *constants.h*

Antes de qualquer coisa, nós adicionamos o custo de acesso da *cache* 3, a qual chamamos de L3 (*level* 3), que equivale a 100, e aumentamos o custo de acesso da RAM de 100 para 1000.

Para selecionar o método de mapeamento, nós definimos a constante *MAPPING* com as seguintes opções:

```
#include <stdbool.h>

#define WORDS_SIZE 4
#define INVALID_ADD -1
#define COST_ACCESS_L1 1
#define COST_ACCESS_L2 10
#define COST_ACCESS_L3 100
#define COST_ACCESS_RAM 1000

#define MAPPING 1
// Set MAPPING to:
// 1 for direct mapping
// 2 for LRU mapping
// 3 for LFU mapping

// #define PRINT_INTERMEDIATE_RAM
#define PRINT_LOG
```

2.2 *cpu.h*

No *header* da CPU temos o Tipo Abstrato de Dados - TAD da máquina, denominado *Machine*, onde adicionamos o nível 3 da memória *cache*, assim como seu *hit* e seu *miss*.

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

2.3 *memory.h*

Já em *memory.h*, nós acrescentamos dois dados ao TAD das linhas (*Line*), ambos contadores auxiliares para os métodos de mapeamento adicionados. O inteiro *timesUsed* é responsável por manter a contagem de quantas vezes uma linha foi usada, a fim de auxiliar o mapeamento *Least Frequently Used* - LFU. No caso do LRU (*Least Recently Used*), temos o inteiro *timeInCache* com a função de guardar há quanto tempo uma linha não é usada.

```
typedef struct {
    MemoryBlock block;
    int tag;
    bool updated;
    int cost;
    int cacheHit;
    int timesUsed;
    int timeInCache;
} Line;
```

2.4 *cpu.c*

Em *cpu.c*, foram modificadas três funções a fim de adaptá-las à adição de um terceiro nível de *cache*. Na função *start*, chamamos a *startCache* para iniciar a *cache* L3 e iniciamos o *missL3* e *hitL3* com 0.

```
1   void start(Machine* machine, Instruction* instructions, int* memoriesSize)
        {
2
3       startRAM(&machine->ram, memoriesSize[0]);
4       startCache(&machine->l1, memoriesSize[1]);
5       startCache(&machine->l2, memoriesSize[2]);
6       startCache(&machine->l3, memoriesSize[3]);
7
8       machine->instructions = instructions;
9       machine->hitL1 = 0;
10      machine->hitL2 = 0;
11      machine->hitL3 = 0;
12      machine->hitRAM = 0;
13      machine->missL1 = 0;
14      machine->missL2 = 0;
15      machine->missL3 = 0;
16      machine->totalCost = 0;
17  }
```

Em *stopMachine* nós apenas chamamos a função *stopCache* para liberar a memória utilizada pela *cache* de nível 3.

```
1   void stop(Machine* machine) {
2
3       free(machine->instructions);
4
5       stopRAM(&machine->ram);
6       stopCache(&machine->l1);
7       stopCache(&machine->l2);
8       stopCache(&machine->l3);
9   }
```

Por fim, na função *printMemories*, nós chamamos a função *printc* para a *cache* L3 e adicionamos um *if* e um *for* para a impressão da mesma na tela, identando as demais impressões em "escada" com a L3.

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

```
22
23                  if(i < machine->l2.size) {
24                      printf("|");
25                      printcolored(machine->l2.lines[i].tag, machine->l2.lines[i
                            ].updated);
26
27                      for(int j = 0; j < WORDS_SIZE; j++)
28                          printf(" %5d |", machine->l2.lines[i].block.words[j]);
29
30                      if(i < machine->l1.size) {
31                          printf("|");
32                          printcolored(machine->l1.lines[i].tag, machine->l1.
                                lines[i].updated);
33
34                          for(int j = 0; j < WORDS_SIZE; j++)
35                              printf(" %5d |", machine->l1.lines[i].block.words[
                                    j]);
36                      }
37                  }
38              }
39
40              printf("\n");
41          }
42      }
```

2.5 *main.c*

Na função *main*, a quantidade de argumentos recebidos aumentou de cinco para seis, a fim de receber o tamanho da *cache* L3. Ao vetor *memoriesSize* foi adicionada uma posição para comportar esse tamanho, digitado pelo usuário.

```
1       int main(int argc, char**argv) {
2
3           srand(1507);
4
5           if(argc != 6) {
6               printf("Numero de argumentos invalidos! Sao 6.\n");
7               printf("Linha de execucao: ./exe TIPO_INSTRUCAO [TAMANHO_RAM|
                    ARQUIVO_DE_INSTRUCOES] TAMANHO_L1 TAMANHO_L2 TAMANHO_L3\n");
8               printf("\tExemplo 1 de execucao: ./exe random 10 2 4 6\n");
9               printf("\tExemplo 2 de execucao: ./exe file arquivo_de_instrucoes.
                    txt\n");
10              return 0;
11          }
12
13          int memoriesSize[4];
14          Machine machine;
15          Instruction *instructions;
16
17          memoriesSize[1] = atoi(argv[3]);
18          memoriesSize[2] = atoi(argv[4]);
19          memoriesSize[3] = atoi(argv[5]);
20
21          if(strcmp(argv[1], "random") == 0) {
22              memoriesSize[0] = atoi(argv[2]);
23              instructions = generateRandomInstructions(memoriesSize[0]);
24          }
25
26          else if(strcmp(argv[1], "file") == 0) {
27              instructions = readInstructions(argv[2], memoriesSize);
28          }
```

```
29
30          else {
31              printf("Invalid option.\n");
32              return 0;
33          }
```

Além disso, chamamos a função *sleep*, da biblioteca *unistd.h*, para simular o tempo de inicialização e parada da máquina.

```
1
2           printf("Starting machine...\n");
3           sleep(1);
4           start(&machine, instructions, memoriesSize);
5
6           if(memoriesSize[0] < 10)
7               printMemories(&machine);
8
9           run(&machine);
10
11          if(memoriesSize[0] < 10)
12              printMemories(&machine);
13
14          stop(&machine);
15
16          printf("Stopping machine...\n");
17          sleep(1);
18          return 0;
19      }
```

2.6 *memory.c*

Em *memory.c*, na função *startCache*, inicializamos os contadores *timesUsed* e *timeInCache* como 0 no laço de repetição que antes apenas inicializava as tags das linhas.

```
1       void startCache(Cache* cache, int size) {
2
3           cache->lines = (Line*) malloc(sizeof(Line) * size);
4           cache->size = size;
5
6           for(int i = 0; i < size; i++) {
7               cache->lines[i].tag = INVALID_ADD;
8               cache->lines[i].timesUsed = 0;
9               cache->lines[i].timeInCache = 0;
10          }
11      }
```

2.7 *mmu.c*

O arquivo *mmu.c* foi o que mais sofreu modificações no geral. Na função *memoryCacheMapping*, nós criamos um *switch* baseado na nossa constante *MAPPING*, onde o *case* 1 realiza o mapeamento direto, o *case* 2 o mapeamento LRU e o *case* 3 o LFU.

```
1       int memoryCacheMapping(int address, Cache* cache) {
2
3           switch(MAPPING) {
4
5               case 1:
6                   return address % cache->size;
7                   break;
```

Nos demais mapeamentos, caso o *adress* seja encontrado em uma linha, o índice da mesma é retornado. Para o LRU, temos a variável *leastRecentlyUsed*, que é inicializada com 0 e atualizada no

laço de repetição caso haja uma linha sem ser usada a mais tempo. Essa comparação é feita por meio do contador *timeInCache*.

```
            case 2:

                int leastRecentlyUsed = 0;

                for(int i = 0; i < cache->size; i++) {

                    if(cache->lines[i].tag == address)
                        return i;

                    if(cache->lines[i].timeInCache > cache->lines[
                        leastRecentlyUsed].timeInCache) {
                        leastRecentlyUsed = i;
                    }
                }

                return leastRecentlyUsed;
                break;
```

No caso do LFU, a variável *leastFrequentlyUsed* é também inicializada com 0, porém, é atualizada na repetição se houver uma linha que foi utilizada menos vezes, o que é contabilizado por *timesUsed*.

```
            case 3:

                int leastFrequentlyUsed = 0;

                for(int i = 0; i < cache->size; i++) {

                    if(cache->lines[i].tag == address)
                        return i;

                    if(cache->lines[i].timesUsed < cache->lines[
                        leastFrequentlyUsed].timesUsed) {
                        leastFrequentlyUsed = i;
                    }
                }

                return leastFrequentlyUsed;
                break;
        }
    }
```

Em *updateMachineInfos* atualizamos os *cache hits* e *cache misses* para cada *case*, adequando-os à inserção da *cache* L3, além de resetar o contador *timeInCache* e incrementar o *timesUsed*.

```
    void updateMachineInfos(Machine* machine, Line* line) {

        switch(line->cacheHit) {

            case 1:
                machine->hitL1 += 1;
                break;

            case 2:
                machine->hitL2 += 1;
                machine->missL1 += 1;
                break;

            case 3:
                machine->hitL3 += 1;
                machine->missL1 += 1;
```

```
                machine->missL2 += 1;
                break;

            case 4:
                machine->hitRAM += 1;
                machine->missL1 += 1;
                machine->missL2 += 1;
                machine->missL3 += 1;
                break;
        }

        line->timeInCache = 0;
        line->timesUsed += 1;
        machine->totalCost += line->cost;
    }
```

Por fim, na função *mmuSearchOnMemorys*, nós adicionamos a *cache* L3, chamando a função *usleep* para simular o tempo de procura em cada *cache*.

```
    Line* MMUSearchOnMemorys(Address add, Machine* machine) {

        usleep(40000);
        int l1pos = memoryCacheMapping(add.block, &machine->l1);

        usleep(100000);
        int l2pos = memoryCacheMapping(add.block, &machine->l2);

        usleep(200000);
        int l3pos = memoryCacheMapping(add.block, &machine->l3);
```

Incorporamos três repetições (*for*) para incrementar o contador *timeInCache* em cada linha das 3 *caches* após o mapeamento, já que, para mapear, é necessário percorrer todas as linhas.

```
        Line* cache1 = machine->l1.lines;
        Line* cache2 = machine->l2.lines;
        Line* cache3 = machine->l3.lines;
        MemoryBlock* RAM = machine->ram.blocks;

        for(int i = 0; i < machine->l1.size; i++) {
            machine->l1.lines[i].timeInCache += 1;
        }

        for(int i = 0; i < machine->l2.size; i++) {
            machine->l2.lines[i].timeInCache += 1;
        }

        for(int i = 0; i < machine->l3.size; i++) {
            machine->l3.lines[i].timeInCache += 1;
        }
```

Atualizamos os custos dentro de cada condicional que checa os *cache hits*.

```
        if(cache1[l1pos].tag == add.block) {

            cache1[l1pos].cost = COST_ACCESS_L1;
            cache1[l1pos].cacheHit = 1;
        }

        else if(cache2[l2pos].tag == add.block) {

            cache2[l2pos].tag = add.block;
            cache2[l2pos].updated = false;
            cache2[l2pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2;
            cache2[l2pos].cacheHit = 2;
```

```
13
14              updateMachineInfos(machine, &(cache2[l2pos]));
15              return &(cache2[l2pos]);
16          }
17
18          else if(cache3[l3pos].tag == add.block) {
19
20              cache3[l3pos].tag = add.block;
21              cache3[l3pos].updated = false;
22              cache3[l3pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2 +
                    COST_ACCESS_L3;
23              cache3[l3pos].cacheHit = 3;
24
25              updateMachineInfos(machine, &(cache3[l3pos]));
26              return &(cache3[l3pos]);
27          }
```

Enfim, acrescentamos a L3 no *else*, resetamos o *timeInCache* e atualizamos o *cacheHit* de 3 para 4.

```
1           else {
2               if(!canOnlyReplaceBlock(cache1[l1pos])) {
3                   if (!canOnlyReplaceBlock(cache2[l2pos])) {
4                       if (!canOnlyReplaceBlock(cache3[l3pos])) {
5                           RAM[cache3[l3pos].tag] = cache3[l3pos].block;
6                       }
7                       cache3[l3pos] = cache2[l2pos];
8                       machine->l3.lines[l3pos].timeInCache = 0;
9                   }
10                  cache2[l2pos] = cache1[l1pos];
11                  machine->l2.lines[l2pos].timeInCache = 0;
12              }
13              cache1[l1pos].block = RAM[add.block];
14              machine->l1.lines[l1pos].timeInCache = 0;
15              cache1[l1pos].tag = add.block;
16              cache1[l1pos].updated = false;
17              cache1[l1pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2 +
                    COST_ACCESS_L3 + COST_ACCESS_RAM;
18              cache1[l1pos].cacheHit = 4;
19          }
20
21          updateMachineInfos(machine, &(cache1[l1pos]));
22          return &(cache1[l1pos]);
23      }
```

3 Impressões gerais

A partir da realização deste trabalho, fomos capazes de entender um pouco mais sobre a hierarquia de memória, especificamente sobre a *cache* e a RAM, e sobre as formas de mapeamento de memória.

De forma geral, para o funcionamento do código, alteramos algumas funções já existentes a fim de incluir a *cache* L3, e todas as funções subsequentes foram também adaptadas para os três níveis de *cache*. Escolhemos desenvolver, como dois mapeamentos adicionais, o LRU (*Least Recently Used*) e o LFU (*Least Frequently Used*).

4 Análise

Para realizar os experimentos variando o tamanho das *caches* e avaliando a quantidade de *cache hits* e *cache misses*, nós utilizamos o arquivo de teste *instrucao.in*, obtendo os seguintes resultados:

	Cache 1	Cache 2	Cache 3
M1	8	16	32
M2	32	64	128
M3	16	64	256
M4	8	32	128
M5	16	32	64

Figura 1: Resultados do Mapeamento Direto I

Taxa C1 %	Taxa C2 %	Taxa C3 %	Taxa RAM %	Taxa de Disco%	Tempo de exec.
~23	~14	~19,7	~53	x	14729773
~34	~24,6	~35,5	~32	x	9201573
~34	~30	~21	~36,7	x	10309703
~22,5	~15	~16,5	~55,6	x	15163653
~33	~25	~27	~36	x	10287173

Figura 2: Resultados do Mapeamento Direto II

	Cache 1	Cache 2	Cache 3
M1	8	16	32
M2	32	64	128
M3	16	64	256
M4	8	32	128
M5	16	32	64

Figura 3: Resultados do LRU I

Taxa C1 %	Taxa C2 %	Taxa C3 %	Taxa RAM %	Taxa de Disco%	Tempo de exec.
~23	~64	~13	~44	x	6899823
~82	~6	~13	~14	x	3961043
~30	~75	~24	~13	x	3785053
~23	~69	~8	~22	x	6200483
~29	~74	~6	~17	x	4784303

Figura 4: Resultados do LRU II

	Cache 1	Cache 2	Cache 3
M1	8	16	32
M2	32	64	128
M3	16	64	256
M4	8	32	128
M5	16	32	64

Figura 5: Resultados do LFU I

Taxa C1 %	Taxa C2 %	Taxa C3 %	Taxa RAM %	Taxa de Disco%	Tempo de exec.
~34	~21	~64	~18,5	x	5994873
~82	~6	~13,7	100	x	3943933
~82	~6	~24	100	x	3583753
~34	~71	~11	100	x	4676673
~82	~3	~6	100	x	4448313

Figura 6: Resultados do LFU II

5 Conclusão

A realização do trabalho foi enfastiante e complexa, além dos casos de teste serem confusos. O que melhorou em relação ao primeiro trabalho, entretanto, foram as explicações do professor sobre o trabalho, já que foram disponibilizadas duas aulas para tirarmos dúvidas, além das explicações dadas em aula.