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

Trabalho Prático II

BCC266 - Organização de Computadores

Vitor Oliveira Diniz
Maria Luiza Aragão
Jéssica Machado
Professor: Pedro Silva

Ouro Preto
16 de fevereiro de 2023

Sumário

Lista de Figuras

Lista de Códigos Fonte

1 Introdução

A memória cache funciona como uma biblioteca de acesso rápido que existe dentro de computadores e dispositivos móveis. Ela tem o objetivo de guardar dados, informações e processos temporários acessados com frequência e assim agilizar o processo de uso no momento em que são requisitados pelo usuário.

1.1 Especificações do problema

Para este trabalho prático, deveríamos, a partir dos caches 1 e 2 que foram disponibilizados pelo professor, implementar a memória cache 3 e fazer a movimentação de dados entre os diferentes tipos de cache.

1.2 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: Visual Studio Code LaTeXWorkshop.

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *CLANG*: ferramentas de análise estática do código.
- *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: Ryzen 7-5800H.
- Memória RAM: 16 Gb.
- Sistema Operacional: Arch Linux x86_64.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc main.c -c -Wall
gcc mmu.c -c -Wall
gcc memory.c -c -Wall
gcc instruction.c -c -Wall
gcc cpu.c -c -Wallnewline gcc generator.c -c -Wall
gcc main.o mmu.o memory.o instruction.o cpu.o generator.o -o exe -g
```

Usou-se para a compilação as seguintes opções:

- *-Wall*: para mostrar todos os possível *warnings* do código.
- *-c*: Para compilar o arquivo sem linkar os arquivos para obtermos um arquivo do tipo objeto.
- *-o*: Compilar para um arquivo do tipo output (saída).

Para a execução do programa basta digitar um dos exemplos:

```
./exe random [TAMANHO DA RAM] [L1] [L2] [L3]
```

2 Desenvolvimento

Seguindo as boas práticas de programação, implementamos uma memória cache particionada em linhas e simulamos o mapeamento associativo para a troca das mesmas com a memória RAM. Usamos as políticas LFU (Last Frequently Used) e LRU (Last Recently Used) como métodos de otimização.

2.1 Operações

A seguir entraremos em detalhe sobre as principais funções utilizadas no programa e as coisas que implementamos.

2.1.1 Médodos de mapeamento

Define o tipo de método que vai ser usado para a saída dos resultados.

```
1
2       // 1 MAPEAMENTO DIRETO
3       // 2  LRU (Least Recently Used)
4       // 3  LFU (Least Frequently Used)
5
6       #define SUBSTITUTION_METHOD 3
```

Código 1: Definição do tipo de método

2.1.2 TAD Machine

Adicionamos cache L3, o hitL3 (que conta quantas vezes processador achou o dado na cache) e o missL3 (que conta quantas vezes o processador precisou buscar o dado na memória ou até no disco).

```
1
2       typedef struct {
3           Instruction* instructions;
4           RAM ram;
5           Cache l1; // cache L1
6           Cache l2; // cache L2
7           Cache l3; // cache L3
8           int hitL1, hitL2, hitL3, hitRAM;
9           int missL1, missL2, missL3;
10          int totalCost;
11          } Machine;
```

Código 2: TAD Machine

2.1.3 run

Inicializa o contador PC com valor 0 e, enquanto o opcode de machine->instructions[PC] for diferente de -1 (condição de parada), printamos a quantidade de vezes que o processador achou (hit) o dado na L1, L2, L3 e RAM, a quantidade de vezes que o processador precisou buscar o dado na memória ou até no disco (miss), e o custo total.

```
1       void run(Machine* machine) {
2           int PC = 0; // Program Counter
3           while(machine->instructions[PC].opcode != -1) {
4               executeInstruction(machine, PC++);
5               printf("\tL1:(%6d, %6d) | L2:(%6d, %6d) | L3:(%6d, %6d) | RAM:(%6d
                    ) | COST: %d\n",
6                   machine->hitL1, machine->missL1,
7                   machine->hitL2, machine->missL2,
8                   machine->hitL3, machine->missL3,
9                   machine->hitRAM,
10                  machine->totalCost);
11          }
```

```
}
```

Código 3: Funçao run

2.1.4 printMemories

Printa as informações da RAM, se está atualizado é a cor verde e, se não, cor vermelha.

```
    void printMemories(Machine* machine) {
        printf("\x1b[0;30;47m     ");
        printc("RAM", WORDS_SIZE * 8 - 1);
        printc("Cache L3", WORDS_SIZE * 8 + 6);
        printc("Cache L2", WORDS_SIZE * 8 + 6);
        printc("Cache L1", WORDS_SIZE * 8 + 6);
        printf("\x1b[0m\n");



        for (int i=0;i<machine->ram.size;i++) {
            printf("\x1b[0;30;47m%5d|\x1b[0m", i);
            for (int j=0;j<WORDS_SIZE;j++)
                printf(" %5d |", machine->ram.blocks[i].words[j]);

            if (i < machine->l3.size) {
                printf("|");
                printcolored(machine->l3.lines[i].tag, machine->l3.lines[i].
                    updated);
                for (int j=0;j<WORDS_SIZE;j++)
                        printf(" %5d |", machine->l3.lines[i].block.words[j]);

                if (i < machine->l2.size) {
                    printf("|");
                    printcolored(machine->l2.lines[i].tag, machine->l2.lines[i
                        ].updated);
                    for (int j=0;j<WORDS_SIZE;j++)
                        printf(" %5d |", machine->l2.lines[i].block.words[j]);
                    if (i < machine->l1.size) {
                        printf("|");
                        printcolored(machine->l1.lines[i].tag, machine->l1.
                            lines[i].updated);
                        for (int j=0;j<WORDS_SIZE;j++)
                            printf(" %5d |", machine->l1.lines[i].block.words[
                                j]);
                    }
                }
            }
            printf("\n");
        }
    }
```

Código 4: Função printMemories

2.1.5 startCache

Aloca dinamicamente as linhas e inicia a cache com o valor inválido (-1).

```
1   // ALOCA DINAMICAMENTE A CACHE COM UM TAMANHO ESPECIFICO
2   void startCache(Cache* cache, int size) {
3       cache->lines = (Line*) malloc(sizeof(Line) * size);
4       cache->size = size;
5
6       // INICIA A CACHE COM UM VALOR INVALIDO -1
7       for (int i=0;i<size;i++){
8           cache->lines[i].tag = INVALID_ADD;
9           cache->lines[i].timeOnCache = 0;
10          cache->lines[i].timesUsed = 0;
11      }
12  }
```

Código 5: startCache

2.1.6 TAD Line

Adicionamos as variáveis timesUsed, do tipo inteiro, para contar quantas vezes a variável passada para a cache foi usada, e a timeOnCache o tempo que informação ficou sem ser acessada.

```
1
2   typedef struct {
3       MemoryBlock block; // BLOCO DE MEMORIA, QUE CONTEM 1 VETOR DE PALAVRAS
            , O BLOCO REPRESENTA CACHE L1 L2 L3 E RAM
4       int tag; /* Address of the block in memory RAM */
5       bool updated;
6       int cost; // custo de acesso a CACHE/RAM
7       int cacheHit;
8       int timesUsed;
9       int timeOnCache;
10  } Line;
```

Código 6: TAD Line

2.1.7 memoryCacheMapping

Essa função é responsável por checar qual tipo de mapeamento estamos utilizando e, dependendo do mapeamento, iremos aplicar uma política diferente. No caso do mapeamento direto, retornamos o possível endeeço da cache e a verificação de sua existência é feita no MMUSearchOnMemorys. No caso da LRU, percorremos o vetor sempre checando qual endereço está a mais tempo na cache e salvando seu índice, caso encontremos o endereço que procuramos retornamos seu índice, se a substituição vai acontecer ou não depende do MMUSearchOnMemorys. No LFU acontece a mesma coisa que o LRU, apenas mudando a condição de checagem para o bloco menos usado.

```
1
2   int memoryCacheMapping(int address, Cache* cache) {
3
4   int index = 0;
5
6   switch(SUBSTITUTION_METHOD){
7       //DIRECT MAPPING
8       case 1:
9           return address % cache->size;
10          break;
11      //LRU METHOD (Least Recently Used)
12      case 2:
13          cache->lines[0].timeOnCache++;
14          for( int i = 0; i < cache->size; i++){
```

```
15
16                  cache->lines[i].timeOnCache++;
17
18                  if(cache->lines[i].timeOnCache > cache->lines[index].
                        timeOnCache)
19                      index = i;
20
21                  if(cache->lines[i].tag == address)
22                      return i;
23
24              }
25              return index;
26              break;
27
28          //LFU METHOD (Least Frequently Used)
29          case 3:
30              for( int i = 0; i < cache->size; i++){
31                  if(cache->lines[i].timesUsed < cache->lines[index].timesUsed)
32                      index = i;
33
34                  if(cache->lines[i].tag == address)
35                      return i;
36              }
37
38              return index;
39              break;
40      }
41
42      return address % cache->size;
43  }
```

Código 7: Função memoryCacheMapping

2.1.8 updateMachineInfos

Modificamos os valores de hit e miss da cache l3, contamos quantas a linha foi usada e o custo.

```
1
2       void updateMachineInfos(Machine* machine, Line* line) {
3
4
5           switch (line->cacheHit) {
6               case 1:
7                   machine->hitL1 += 1;
8                   break;
9
10              case 2:
11                  machine->hitL2 += 1;
12                  machine->missL1 += 1;
13                  break;
14
15              case 3:
16                  machine->hitL3 += 1;
17                  machine->missL1 += 1;
18                  machine->missL2 += 1;
19                  break;
20
21              case 4:
22                  machine->hitRAM += 1;
23                  machine->missL1 += 1;
24                  machine->missL2 += 1;
25                  machine->missL3 += 1;
```

```
26                  break;
27          }
28
29          line->timeOnCache = 0;
30          line->timesUsed++;
31
32          machine->totalCost += line->cost;
33      }
```

Código 8: Função updateMachineInfos

2.1.9 MMUSearchOnMemorys

Inicialmente, essa função pega a possível posição do bloco que queremos na CACHE e conferimos se o que queremos está realmente na RAM. Se estiver, atualizamos o cache hit referente aonde ele se encontra, o custo de acesso da memória, a nossa máquina e retornamos a linha que o endereço se encontra. Se não encontrarmos o endereço na cache, fazemos as movimentações necessárias entre as memórias, levando em consideração o mapeamento escolhido e, caso precise, levamos o endereço da cache para RAM.

```
1
2       Line* MMUSearchOnMemorys(Address add, Machine* machine) {
3       // Strategy => write back
4
5       // Direct memory map
6       int l1pos = memoryCacheMapping(add.block, &machine->l1);
7       int l2pos = memoryCacheMapping(add.block, &machine->l2);
8       int l3pos = memoryCacheMapping(add.block, &machine->l3);
9
10
11      Line* cache1 = machine->l1.lines;
12      Line* cache2 = machine->l2.lines;
13      Line* cache3 = machine->l3.lines;
14
15      // adicionar linha da L3
16      MemoryBlock* RAM = machine->ram.blocks;
17
18      if (cache1[l1pos].tag == add.block) {
19          /* Block is in memory cache L1 */
20          // ESTA FALTANDO TAG E UPDATED
21          cache1[l1pos].cost = COST_ACCESS_L1;
22          cache1[l1pos].cacheHit = 1;
23      } else if (cache2[l2pos].tag == add.block) {
24          /* Block is in memory cache L2 */
25          cache2[l2pos].tag = add.block;
26          cache2[l2pos].updated = false;
27          cache2[l2pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2;
28          cache2[l2pos].cacheHit = 2;
29          updateMachineInfos(machine, &(cache2[l2pos]));
30
31          return &(cache2[l2pos]);
32      } else if (cache3[l3pos].tag == add.block){
33          /* Block is in memory cache L3 */
34          cache3[l3pos].tag = add.block;
35          cache3[l3pos].updated = false;
36          cache3[l3pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2 + COST_ACCESS_L3;
37          cache3[l3pos].cacheHit = 3;
38
39          updateMachineInfos(machine, &(cache3[l3pos]));
40          return &(cache3[l3pos]);
41
42      } else {
43          /* Block only in memory RAM, need to bring it to cache and manipulate
                the blocks */
44          //fazer o mapeamento para decidir quem tirar da ram.
45          l2pos = memoryCacheMapping(cache1[l1pos].tag, &machine->l2); /* Need
                to check the position of the block that will leave the L1 */
46          l3pos = memoryCacheMapping(cache2[l2pos].tag, &machine->l3); /* Need
                to check the position of the block that will leave the L1 */
47
48
49          if (!canOnlyReplaceBlock(cache1[l1pos])) {
```

```
50              /* The block on cache L1 cannot only be replaced, the memories
                    must be updated */
51              if (!canOnlyReplaceBlock(cache2[l2pos])){
52                  /* The block on cache L2 cannot only be replaced, the memories
                        must be updated */
53                  if(!canOnlyReplaceBlock(cache3[l3pos])){
54                      /* The block on cache L2 cannot only be replaced, the
                            memories must be updated */
55                      RAM[cache3[l3pos].tag] = cache3[l3pos].block;
56                  }
57                  cache3[l3pos] = cache2[l2pos];
58                  cache3[l3pos].timeOnCache = 0;
59
60
61              }
62
63              cache2[l2pos] = cache1[l1pos];
64              cache2[l2pos].timeOnCache = 0;
65
66          }
67
68
69          cache1[l1pos].block = RAM[add.block];
70          cache1[l1pos].tag = add.block;
71          cache1[l1pos].updated = false;
72          cache1[l1pos].cost = COST_ACCESS_L1 + COST_ACCESS_L2 + COST_ACCESS_L3
                + COST_ACCESS_RAM;
73
74          //se passa da l1 para a l2 eu deveria zerar ?
75          cache1[l1pos].cacheHit = 4;
76          cache1[l1pos].timeOnCache = 0;
77
78
79      }
80      updateMachineInfos(machine, &(cache1[l1pos]));
81      return &(cache1[l1pos]);
82  }
```

Código 9: Função MMUSearchOnMemorys

2.2 Função Main

Na função main adicionamos mais um espaço a memoriesSize para a l3.

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

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

    if (argc != 6) {
        printf("Numero de argumentos invalidos! Sao 6.\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;
    }

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

    memoriesSize[1] = atoi(argv[3]);
    memoriesSize[2] = atoi(argv[4]);
    memoriesSize[3] = atoi(argv[5]);


    if (strcmp(argv[1], "random") == 0) {
        memoriesSize[0] = atoi(argv[2]);
        instructions = generateRandomInstructions(memoriesSize[0]);
    } 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] < 10)
        printMemories(&machine);
    run(&machine);
    if (memoriesSize[0] < 10)
        printMemories(&machine);
    stop(&machine);
    printf("Stopping machine...\n");
    return 0;
}
```

Código 10: Main

3 Impressões Gerais

Primeiramente, nos reunimos no discord para a leitura e compreensão do documento e arquivos disponibilizados para a realização do trabalho. O código disponibilizado estava bem legível o que facilitou um pouco, mas nas aulas faltaram demonstrações/implementações dos métodos de mapeamento. Também encontramos dificuldade de entender a tabela que estava no PDF, porém, esse problema foi resolvido posteriormente com certa facilidade. Um dos pontos positivos foi quando o professor disponibilizou tempo de aula para sanar nossas dúvidas. Para a documentação utilizamos o LaTeX, o que não nos trouxe dificuldade devido ao prévio conhecimento adquirido com trabalhos anteriores. Descobrimos uma extensão bastante útil do VS Code: o Live Share, que nos permitiu editar o código simultaneamente.

4 Análise

Após a análise da parte já disponibilizada do código, identificamos implementações de funções que já havíamos aplicado no trabalho anterior, como a soma, subtração e movimentação de dados. Tendo isso em vista, construímos o raciocínio da aplicação da cache L3, seguindo a mesma linha de pensamento das memórias já implementadas anteriormente. Partimos então para o mapeamento associativo e as movimentações de troca, ponderando sobre as políticas LFU (Last Frequently Used) e LRU (Last Recently Used).

4.1 Análise de Complexidade

Na função memoryCacheMapping, a sua complexidade vai depender do método de mapeamento escolhido. Para o mapeamento direto, como não percorremos o vetor, já que a posição na cache depende unicamente do final de seu endereço, sua complexidade será $\mathcal{O}(1)$.

Na política LRU, na função memoryCacheMapping, percorremos o vetor para encontrar o bloco que está a mais tempo sem ser utilizado, por isso teremos uma complexidade de $\mathcal{O}(n)$.

Na política LFU, na memoryCacheMapping, também percorremos o vetor para encontrar o bloco que está a mais tempo sem ser utilizado, por isso teremos uma complexidade de $\mathcal{O}(n)$.

	Cache 1	Cache 2	Cache 3	Taxa C1 %	Taxa C2 %	Taxa C3 %	Taxa RAM %	Taxa de Disco %	Tempo de execução (unidade hipotética)
M1	8	16	32	23.3	14.2	19.8	100	-	14729773
M2	32	64	128	34.2	24.6	35.6	100	-	9201573
M3	16	64	256	33.9	29.4	21.1	100	-	10309703
M4	8	32	128	22.6	15.4	16.6	100	-	15163653
M5	16	32	64	33.2	25	27.5	100	-	10287173

Figura 1: Mapeamento Direto

	Cache 1	Cache 2	Cache 3	Taxa C1 %	Taxa C2 %	Taxa C3 %	Taxa RAM %	Taxa de Disco %	Tempo de execução (unidade hipotética)
M1	8	16	32	22.6	61.3	19.9	100	-	6806003
M2	32	64	128	68.9	47.0	13.2	100	-	3995943
M3	16	64	256	30.0	75.5	24.54	100	-	3770643
M4	8	32	128	23.5	69.4	8.8	100	-	5992523
M5	16	32	64	29.5	74.5	6.1	100	-	4762923

Figura 2: LRU

	Cache 1	Cache 2	Cache 3	Taxa C1 %	Taxa C2 %	Taxa C3 %	Taxa RAM %	Taxa de Disco %	Tempo de execução (unidade hipotética)
M1	8	16	32	34.0	21.0	64.5	100	-	5994873
M2	32	64	128	82.4	6.2	13.7	100	-	3943933
M3	16	64	256	82.1	6.1	24.4	100	-	3583753
M4	8	32	128	34.1	71.7	11.4	100	-	4676673
M5	16	32	64	82.1	3.7	6.2	100	-	4448313

Figura 3: LFU

5 Conclusão

Com este trabalho, aprendemos sobre a memória CACHE e movimentação de dados entre diferentes tipos de cache, dependendo do tipo de mapeamento usado. Entre as políticas de mapeamento, o mapeamento direto é, sem dúvidas, o pior método de todos. Já entre o LFU e LRU, os resultados adquiridos na comparação entre o cache hit e cache miss foram semelhantes. Uma das dificuldades encontradas foi a falta de uma saída teste para efeitos de comparação para que a gente pudesse ver se a saída do nosso programa está correta. Dito isso, achamos a dinâmica do trabalho excelente e bem didática possamos aprender de um jeito palpável esses conceitos.