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

Paulo Eduardo Costalonga Armani, Kaique Oliveira
Professor: Pedro Henrique Lopes Silva

Ouro Preto
20 de julho de 2023

Sumário

Lista de Tabelas

Lista de Códigos Fonte

1 Introdução

O objetivo é criar um algoritmo que funcione como um simulador do mapeamento associativo, permitindo a troca de linhas entre caches e a memória RAM, seguindo o padrão da hierarquia de memória encontrada em sistemas contemporâneos de computação. Essa implementação possibilitará a análise de diversas políticas de gerenciamento de memória e a avaliação de seu impacto no desempenho global do sistema.

1.1 Especificações do problema

É necessário desenvolver um modelo de computação capaz de emular a hierarquia de memória existente em sistemas de computação atuais. Esse modelo deve ser capaz de simular o acesso tanto à memória principal quanto à memória cache. Para atingir esse objetivo, é essencial criar funções em linguagem C que possibilitem a leitura e escrita de dados na memória, além de funções dedicadas à implementação das políticas de gerenciamento de cache.

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code. [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *Live Share*: ferramenta usada para *pair programming* à distância.
- *Valgrind*: ferramentas de análise dinâmica do código.
- *Google Meet*: ferramenta usada para comunicação da dupla

1.4 Especificações da máquina

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

- Processador: AMD® Ryzen 5 5500u with radeon graphics × 12
- Memória RAM: 8GB
- Sistema Operacional: Ubuntu 22.04.2 LTS

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar o seguinte comando para rodar o arquivo *Makefile* disponível:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

- *-o*: para definir o arquivo de saída.

[1] VScode está disponível em https://code.visualstudio.com/
[2] Disponível em https://www.overleaf.com/

- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.

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

- *-c*: para compilação do código e geração dos arquivos objetos.

Para entrar com um arquivo já existente basta digitar:

```
./exe file 'arquivo.in' 'tamanhoL1' 'tamanhoL2 ' 'tamanhoL3'
```

Para executar o comando random:

```
../exe random 'tamanhoRAM' 'tamanhoL1' 'tamanhoL2' 'tamanhoL3'
```

2 Implementação

O desenvolvimento foi realizado utilizando da técnica de pair programming, onde a dupla em conjunto programou ativamente no código ao mesmo tempo. O uso das ferramentas Live Share para o compartilhamento de código e do Google Meet para a comunicação em equipe foi o que viabilizou o uso da técnica citada. 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) e FIFO (first-in first-out).

2.1 Cache L3

A seguir a cache L3:

```
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;
```

Código 1: 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.

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.

2.2 Mapeamento Associativo

Para implementar o mapeamento associativo das memórias, foi adicionado um switch dentro da função 'memoryCacheMapping', no mmu.c, que seleciona a política de mapeamento com base no valor da variável 'mapppolicyType'. Os casos de 1 a 4 representam as diferentes políticas de mapeamento disponíveis.
A seguir o código da função:

```
int memoryCacheMapping(int address, Cache *cache)
{
    int mappPolicyType = 1;

    switch (mappPolicyType)
    {
    // Mapeamento direto
    case 1:
        return directMapping(address, cache);

    // Least Recently Used (LRU)
    case 2:
        return lru(address, cache);

    // Least Frequently Used (LFU)
    case 3:
        return lfu(address, cache);

    // First In First Out (FIFO)
    case 4:
        return fifo(address, cache);

    default:
        return -1;
    }
}
```

Código 2: Função de memoryCacheMapping.

2.3 LRU

A seguir o código da função:

```
int lru(int address, Cache *cache)
{
    int leastRecentlyUsed = 0;

    for (int i = 0; i < cache->size; i++)
    {
        // Verifica se o bloco áest na linha atual da cache
        if (cache->lines[i].tag == address)
            return i;

        // Encontra a linha menos recentemente usada na cache
        if (cache->lines[i].timeInCache > cache->lines[leastRecentlyUsed].
            timeInCache)
            leastRecentlyUsed = i;
    }

    return leastRecentlyUsed;
}
```

Código 3: Função LRU(Least Recently Used).

Essa função lru é responsável por implementar a política de substituição Least Recently Used (LRU) em uma cache. Essa política determina que o bloco de memória a ser substituído é aquele que foi menos recentemente utilizado, ou seja, aquele que não foi acessado há mais tempo.

A função recebe dois parâmetros: o address (endereço de memória) que se deseja mapear na cache e um ponteiro para a estrutura de dados Cache, que representa a cache onde o mapeamento será realizado.

Dentro da função, é criada uma variável inteira chamada leastRecentlyUsed e inicializada com o valor 0. Essa variável será utilizada para armazenar o índice da linha da cache que contém o bloco menos recentemente utilizado.

Em seguida, é iniciado um loop for que percorre todas as linhas da cache. O loop começa com i igual a 0 e continua até que i seja menor que o tamanho da cache (cache-¿size).

Dentro do loop, a função realiza duas tarefas principais:

Verifica se o bloco de memória com o endereço address já está presente na linha atual da cache. Isso é feito comparando a tag do bloco presente na linha (cache-¿lines[i].tag) com o endereço address. Se o bloco for encontrado, a função retorna o índice da linha (i), indicando que não é necessário substituir o bloco, pois ele já está presente na cache.

Encontra a linha que contém o bloco menos recentemente utilizado na cache. Para fazer isso, a função compara o valor do contador de tempo (timeInCache) do bloco na linha atual (cache-¿lines[i].timeInCache) com o valor do contador de tempo do bloco armazenado na linha leastRecentlyUsed. Se o contador de tempo do bloco atual for maior do que o contador de tempo do bloco armazenado na variável leastRecentlyUsed, então atualizamos leastRecentlyUsed para o valor de i, indicando que encontramos uma linha com um bloco menos recentemente utilizado.

Após percorrer todas as linhas da cache, a função retorna o valor de leastRecentlyUsed, que representa o índice da linha que contém o bloco menos recentemente utilizado na cache. Esse índice será utilizado para substituir o bloco antigo pelo novo quando necessário.

2.4 LRU

A seguir o código da função:

```
int lfu(int address, Cache *cache)
{
    int leastFrequentlyUsed = 0;

    for (int i = 0; i < cache->size; i++)
    {
        // Verifica se o bloco áest na linha atual da cache
        if (cache->lines[i].tag == address)
            return i;

        // Encontra a linha menos frequentemente usada na cache
        if (cache->lines[i].timesUsed < cache->lines[leastFrequentlyUsed].
            timesUsed)
            leastFrequentlyUsed = i;
    }

    return leastFrequentlyUsed;
}
```

Código 4: Função LFU(Least Frequently Used).

Essa função lfu implementa a política de substituição Least Frequently Used (LFU) em uma cache. Essa política determina que o bloco de memória a ser substituído é aquele que foi menos frequentemente utilizado, ou seja, aquele que teve o menor número de acessos ao longo do tempo.

A função recebe dois parâmetros: o address (endereço de memória) que se deseja mapear na cache e um ponteiro para a estrutura de dados Cache, que representa a cache onde o mapeamento será realizado.

Dentro da função, é criada uma variável inteira chamada leastFrequentlyUsed e inicializada com o valor 0. Essa variável será utilizada para armazenar o índice da linha da cache que contém o bloco menos frequentemente utilizado.

Em seguida, é iniciado um loop for que percorre todas as linhas da cache. O loop começa com i igual a 0 e continua até que i seja menor que o tamanho da cache (cache-¿size).

Dentro do loop, a função realiza duas tarefas principais:

Verifica se o bloco de memória com o endereço address já está presente na linha atual da cache. Isso é feito comparando a tag do bloco presente na linha (cache-¿lines[i].tag) com o endereço address. Se o bloco for encontrado, a função retorna o índice da linha (i), indicando que não é necessário substituir o bloco, pois ele já está presente na cache.

Encontra a linha que contém o bloco menos frequentemente utilizado na cache. Para fazer isso, a função compara o número de vezes que o bloco na linha atual foi usado (cache-¿lines[i].timesUsed) com o número de vezes que o bloco armazenado na linha leastFrequentlyUsed foi usado (cache-¿lines[leastFrequentlyUsed].timesUsed). Se o número de vezes que o bloco atual foi usado for menor do que o número de vezes que o bloco armazenado em leastFrequentlyUsed foi usado, então atualizamos leastFrequentlyUsed para o valor de i, indicando que encontramos uma linha com o bloco menos frequentemente utilizado.

Após percorrer todas as linhas da cache, a função retorna o valor de leastFrequentlyUsed, que representa o índice da linha que contém o bloco menos frequentemente utilizado na cache. Esse índice será utilizado para substituir o bloco antigo pelo novo quando necessário.

2.5 FIFO

A seguir o código da função:

```
int fifo(int address, Cache *cache)
{
    int oldestLine = 0;

    // Procura por um çespao vazio na cache ou pelo çendereo que áest sendo
       procurado
    for (int i = 0; i < cache->size; i++)
    {
        if (!cache->lines[i].updated || cache->lines[i].tag == address)
            return i;

        // Encontra a linha mais antiga na cache
        if (cache->lines[i].timeInCache < cache->lines[oldestLine].timeInCache
           )
            oldestLine = i;
    }

    // Nenhum çespao vazio e çendereo ãno áest na cache, substitui a linha
       mais antiga
    return oldestLine;
}
```

Código 5: Função FIFO(First In First Out).

Essa função fifo implementa a política de substituição First In First Out (FIFO) em uma cache. Essa política determina que o bloco de memória a ser substituído é aquele que foi o primeiro a ser inserido na cache, ou seja, o bloco mais antigo.

A função recebe dois parâmetros: o address (endereço de memória) que se deseja mapear na cache e um ponteiro para a estrutura de dados Cache, que representa a cache onde o mapeamento será realizado.

Dentro da função, é criada uma variável inteira chamada oldestLine e inicializada com o valor 0. Essa variável será utilizada para armazenar o índice da linha da cache que contém o bloco mais antigo.

Em seguida, é iniciado um loop for que percorre todas as linhas da cache. O loop começa com i igual a 0 e continua até que i seja menor que o tamanho da cache (cache-¿size).

Dentro do loop, a função realiza duas tarefas principais:

Verifica se a linha atual da cache está vazia (!cache-¿lines[i].updated) ou se contém o bloco com o endereço address. Se uma dessas condições for verdadeira, significa que encontramos um espaço vazio na cache ou que o bloco já está presente na cache com o endereço desejado. Em qualquer um desses casos, a função retorna o índice da linha (i), indicando que não é necessário substituir o bloco, pois ele já está presente na cache ou há espaço disponível.

Encontra a linha que contém o bloco mais antigo na cache. Para fazer isso, a função compara o tempo que o bloco na linha atual foi inserido na cache (cache-¿lines[i].timeInCache) com o tempo que o bloco armazenado na linha oldestLine foi inserido na cache (cache-¿lines[oldestLine].timeInCache). Se o tempo de inserção do bloco atual for menor do que o tempo de inserção do bloco armazenado em oldestLine, então atualizamos oldestLine para o valor de i, indicando que encontramos a linha com o bloco mais antigo.

Após percorrer todas as linhas da cache, a função retorna o valor de oldestLine, que representa o índice da linha que contém o bloco mais antigo na cache. Esse índice será utilizado para substituir o bloco antigo pelo novo quando necessário.

3 Impressões Gerais

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

Em relação às coisas que mais agradaram a dupla, 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 a dupla, 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.

4 Análises

Foram usados arquivos de teste para realizar os experimentos.

Para executar o código, os seguintes comandos foram rodados:

Comando de entrada por arquivo

```
$ ./exe file tests/1.in 2 4 6
```

Sendo 2 o tamanho da Cache L1, 4 o tamanho da Cache L2 e 6 o tamanho da Cache L3.

4.1 Resumo dos experimentos

Tempo (em segundos)	Consumo de memória (em bytes)
0,20	286,232

Tabela 1: Resumo dos testes da entrada por arquivo.

Observando os resultados obtidos, listamos algumas vantagens e desvantagens das políticas.

4.2 LRU (Least Recently Used)

4.2.1 Ordem de Complexidade:

Inserção e busca O(n), atualização O(1) para mover o bloco para o início da lista.

4.2.2 Vantagens:

Tende a se comportar bem em situações onde há uma alta taxa de repetição de acesso a alguns blocos específicos.
Pode reduzir a quantidade de substituições desnecessárias, melhorando potencialmente o desempenho geral da cache.

4.2.3 Desvantagens:

Implementar o rastreamento do uso de todos os blocos pode ser mais custoso em termos de hardware ou complexidade do código.
Pode não ser tão eficiente em situações em que a repetição de acesso é menos pronunciada e a carga de trabalho apresenta padrões de acesso não uniformes.

4.3 LFU (Least Frequently Used)

4.3.1 Ordem de Complexidade:

Inserção, busca e atualização O(1).

4.3.2 Vantagens:

Pode funcionar bem em situações onde a frequência de acesso a diferentes blocos é mais uniforme e equilibrada.
Reduz a necessidade de rastrear a ordem temporal exata de acesso, focando apenas na frequência de uso dos blocos.

4.3.3 Desvantagens:

A contagem de frequência de uso de todos os blocos pode ser mais custosa em termos de hardware ou complexidade do código.
Em cargas de trabalho com padrões de acesso variáveis, pode não ser tão eficiente quanto outras políticas.

4.4 FIFO (First In First Out)

4.4.1 Ordem de Complexidade:

Inserção O(1), busca O(n). Não requer atualização quando um bloco é acessado.

4.4.2 Vantagens:

Implementação simples, exigindo apenas a manutenção da ordem de chegada dos blocos na cache.
Pode funcionar bem em situações em que a ordem temporal de chegada dos blocos é relevante para o desempenho.

4.4.3 Desvantagens:

Não considera a frequência de acesso aos blocos, o que pode levar a substituições desnecessárias de blocos frequentemente utilizados.
Pode não ser tão eficiente em situações onde a ordem temporal de chegada não é um fator importante para o desempenho.

5 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.