Universidade Federal do Rio Grande do Sul

INF01112 - Arquitetura e Organização de Computadores I I

Trabalho I : Avaliação de Políticas de Substituição de Dados em Cache

Iagê von Linsingen - 00590289
Miguel Gil de Souza Franskowiak - 00577697
Pedro Collet Krolikowski -

Índice

Introdução

Esse trabalho tem como objetivo analisar o desempenho de duas políticas de substituição de dados em memória cache, considerando o mapeamento totalmente associativo. As políticas LRU (Least Recently Used) e LFU (Least Frequently Used) são avaliadas em três cenários distintos, com diferentes padrões de acesso à memória, de modo a tornar possível a comparação e discussão a respeito das vantagens e desvantagens de cada algoritmo.

Algoritmos

A simulação da memória cache e implementação dos algoritmos foi feita na linguagem C. As sequências de acessos à memória são representadas por vetores de caracteres, onde as letras representam diferentes blocos da memória. A cache é também um vetor de caracteres, de tamanho 6, onde são guardados os acessos e feitas as substituições de acordo com cada política. As funções chamadas na main() simulam o funcionamento da cache com as políticas de substituição LRU e LFU para cada sequência, dada como entrada. Ao final da execução de cada função, é exibido o número total de hits e misses do cenário. Abaixo são mostrados o início do código e a função main():

```c
#include <stdio.h>
#include <stdlib.h>

#define TAM_CACHE 6

// protótipos
void implementa_LRU(char cache[], char seq[], int tamSeq);
void implementa_LFU(char cache[], char seq[], int tamSeq);

// variáveis globais
int contMiss;
int contHit;

int main() {

  char cache[TAM_CACHE];
```

```
// SEQUÊNCIAS DE ACESSO
// sequencia de acessos com alta localidade temporal ("duas fases") (favorece LRU)
  char seq1[] = {.  .  .};
  int tamSeq1 = sizeof(seq1);

  // sequencia com acessos repetitivos a um subconjunto dos dados (favorece LFU)
  char seq2[] = {.  .  .};
  int tamSeq2 = sizeof(seq2);

  // caso híbrido
  char seq3[] = {.  .  .};
  int tamSeq3 = sizeof(seq3);

  printf("Sequência de acessos 1:\n");
  implementa_LRU(cache, seq1, tamSeq1);
  implementa_LFU(cache, seq1, tamSeq1);
  printf("\n");

  printf("Sequência de acessos 2:\n");
  implementa_LRU(cache, seq2, tamSeq2);
  implemenata_LFU(cache, seq2, tamSeq2);
  printf("\n");

  printf("Sequência de acessos 3:\n");
  implementa_LRU(cache, seq3, tamSeq3);
  implementa_LFU(cache, seq3, tamSeq3);

  return 0;
}
```

LRU

A política de substituição “LRU” consiste em substituir na cache elemento que está a mais tempo sem ser acessado na cache. Abaixo é mostrada a função que implementa essa política no código, dada uma sequência de acessos como entrada:

```
void implementa_LRU(char cache[], char seq[], int tamSeq) {
    contHit=0;
    contMiss=0;
```

```
    char lruPilha[TAM_CACHE]; // inicializa a pilha e cache
    for (int i = 0; i < TAM_CACHE; i++) {
        lruPilha[i] = 0;
        cache[i] = 0;
  }
  for (int i = 0; i < tamSeq; i++) {

    int hitFlag = 0;

    for (int j = 0; j < TAM_CACHE; j++) { // percorre a cache pra ver se o
elemento já está lá
      if (seq[i] == cache[j]) { // caso de HIT
        contHit += 1;
        hitFlag = 1;

        int posicPilha = -1;
        for (int k = 0; k < TAM_CACHE; k++) { // procura o elemento na pilha
          if (lruPilha[k] == seq[i]) {
            posicPilha = k;
            break;
          }
        }
        for (int k = posicPilha; k > 0; k--) { // desloca os elementos da
pilha que estão acima do elemento encontrado pra baixo
          lruPilha[k] = lruPilha[k - 1];
        }

        lruPilha[0] = seq[i]; // coloca o elemento encontrado no topo
        break;
      }
    }
    if (hitFlag == 0) { // caso de MISS (não teve hit)
      contMiss += 1;
      int vazioFlag = 0;

      for (int j = 0; j < TAM_CACHE; j++) { // se tiver espaço vazio na cache
        if (cache[j] == 0) {
          vazioFlag = 1;
          cache[j] = seq[i];
          break;
        }
      }
```

```
      if (vazioFlag == 0) { // se não tinha espaço vazio na cache -> subst.
com LRU

        for (int k = 0; k < TAM_CACHE; k++) { // percorre a cache até achar o
elemento no fundo da pilha
          if (cache[k] == lruPilha[TAM_CACHE - 1]) {
            cache[k] = seq[i];
            break;
          }
        }
      }
      for (int k = TAM_CACHE - 1; k > 0; k--) {
        lruPilha[k] = lruPilha[k - 1];
      }
      lruPilha[0] = seq[i];
    }
    hitFlag = 0;
    }
    printf("LRU: %d hits e %d misses\n", contHit, contMiss);
}
```

Visão Geral do código:

1. A sub-rotina recebe como parâmetros os vetores da cache, sequência de acessos e um inteiro contendo a quantidade de elementos presentes na sequência.
2. É utilizada uma estrutura de pilha para armazenar os itens da cache em ordem, com o mais recente sempre no topo (posição zero). Essa forma é ineficiente e tem alta complexidade de tempo, mas serve o propósito de contar as quantidades de hits e misses
3. A função percorre o vetor de sequência de forma iterativa até que todos os seus elementos tenham sido acessados e trazidos para cache em algum momento.

LFU

A política de substituição “LFU” consiste em substituir na cache o elemento que foi menos acessado ao longo da execução de acessos. Abaixo está a função que implementa essa política no código:

```
void implementa_LFU(char cache[], char seq[], int tamSeq) {
  // ZERA os hits e os misses
  contMiss = 0;
  contHit = 0;
  // variáveis de controle
  int flag_hit, flag_vazia;
```

```
int indice_menor;

  // 65 = 'A' em ASCII 0
  int cont[26];

  // ZERA todas as posições do vetor de contadores
  for (int i = 0; i < 26; i++) {
    cont[i] = 0;
  }

  // ZERA todas as posições do CACHE
  for (int i = 0; i < TAM_CACHE; i++) {
    cache[i] = 0;
  }

  // LOOP principal de leitura
  for (int i = 0; i < tamSeq; i++) {
    flag_hit = 0;
    flag_vazia = 0;
    indice_menor = 0;

    // INCREMENTA a contagem no vetor de contagem
    cont[seq[i] - 65]++; //(65 = 'A' em ASCII)

    // LOOP de verificação da cache (tentativa de hit)
    for (int j = 0; j < TAM_CACHE; j++) {
      if (seq[i] == cache[j]) {
        contHit++;
        flag_hit = 1;
        break;
      }
    }

    if (flag_hit == 0) {
        // LOOP para inserção de dado (caso haja uma posição vazia na cache)
        for (int i = 0; i < TAM_CACHE; i++) {
            if (cache[i] == 0){
              cache[i] = seq[i];
              contMiss++;
              flag_vazia = 1;
              break;
            }
        }
```

```
        if (flag_vazia == 0){
            // LOOP de remoção de um dado da cache (para inserção de um novo)
            indice_menor = 0;
            for(int j = 1; j < TAM_CACHE; j++){
                if(cont[cache[j] - 65] < cont[cache[indice_menor] - 65]){
                    indice_menor = j;
                }
            }
            cache[indice_menor] = seq[i]; // insere o novo dado na cache
            contMiss++;
        }
    }
  }
  printf("LFU: %d hits e %d misses\n", contHit, contMiss);
}
```

Visão Geral do código:

1. Função recebe como os mesmos parâmetros de implementa_LRU: vetor da cache, vetor de sequência de acessos e inteiro com a quantidade de elementos no vetor.
2. É criado um vetor que armazena a quantidade de acessos de cada letra, com cada índice do vetor correspondendo a uma letra.
3. Também percorre o vetor de sequência de forma iterativa até que todos os seus elementos tenham sido acessados e trazidos para cache em algum momento. Caso haja empate na frequência dos elementos na cache durante a substituição, o algoritmo sempre substitui o elemento na menor posição dentro da cache.

Cenários e Resultados

O desempenho da cache com diferentes políticas de substituição depende muito do padrão de acesso à memória de cada aplicação. Diferentes sequências de acesso podem favorecer, ou tornar as políticas ineficientes. Assim, para esse trabalho, foram elaborados três casos de teste com características distintas para a avaliação das políticas LRU e LFU.

Caso 1: Alta localidade temporal

O primeiro cenário simula uma aplicação com alta localidade temporal, isso é, uma sequência de acessos à memória onde certos elementos são acessados repetidas vezes durante um curto período de tempo.

```
char seq1[] = {
      'A', 'B', 'C', 'D', 'E', 'F',
      'A', 'C', 'B', 'F', 'D', 'E', 'B', 'A', 'F', 'E', 'C', 'D', 'A', 'F', 'B', 'E',
      'C', 'D', 'A', 'F', 'B', 'E', 'D', 'A', 'F', 'C', 'E', 'B', 'F', 'A', 'E', 'C',
      'B', 'D', 'F', 'A', 'C', 'E', 'G', 'H', 'I', 'J', 'K', 'L',
      'G', 'I', 'H', 'L', 'J', 'K', 'H', 'G', 'L', 'K', 'I', 'J', 'G', 'L', 'H', 'K',
      'I', 'J', 'G', 'K', 'H', 'L', 'J', 'G', 'L', 'I', 'K', 'H', 'G', 'J', 'L', 'H',
      'I', 'K', 'L', 'G', 'J', 'H'
  };
```

Essa sequência de 88 acessos possui alta localidade temporal em duas etapas distintas, sendo a primeira metade dos acessos entre os blocos ‘A’ ao ‘F’, e a outra metade entre os blocos ‘G’ ao ‘L’. Os resultados da implementação com LRU e LFU são:

```
Sequência de acessos 1:
LRU: 76 hits e 12 misses
LFU: 39 hits e 49 misses
```

A política LRU se mostra claramente superior nesse cenário por conseguir se adaptar mais rapidamente a essa mudança de localidade, enquanto a LFU leva em consideração a frequência histórica dos acessos, desfavorecendo a entrada do novo conjunto de blocos na cache.

Caso 2: Acesso repetitivo a um subconjunto reduzido de dados

O segundo cenário simula uma aplicação em que um certo subconjunto dos dados é acessado com maior frequência, com ocasionais longas sequências de acesso a outros dados não frequentes.

```
char seq2[] = { 'A', 'B', 'C', 'D', 'E', 'F', 'A', 'B', 'C', 'D', 'A', 'B', 'C', 'D',
'A', 'B', 'C', 'D', 'A', 'A', 'B', 'B', 'C', 'C', 'D', 'D', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'A', 'B',
'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B', 'C', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'A', 'B', 'C', 'D', 'A',
'B', 'C', 'D', 'D', 'C', 'B' };
```

Na sequência elaborada para esse caso, também com 88 elementos, os blocos ‘A’ ao ‘D’ são acessados repetidamente, e existem duas “varreduras” onde são acessados uma sequência de blocos do ‘G’ ao ‘Z’. Os resultados da implementação com LRU e LFU são:

```
Sequência de acessos 2:
LRU: 34 hits e 54 misses
LFU: 42 hits e 46 misses
```

Nesse cenário, a política LFU se mostrou superior justamente por preservar a frequência histórica dos acessos, sendo capaz de reconhecer que os blocos ‘A’ ao ‘D’ são os mais recorrentes na aplicação e devem permanecer na cache se possível.

Caso 3: Híbrido

O último caso elaborado contém padrões que favorecem e desfavorecem ambas as políticas na mesma sequência. O intuito é observar como se comportam os algoritmos em um cenário misto, onde tanto localidade temporal quanto frequência de acesso afetam diretamente a proporção de hits e misses, e as diferenças de desempenho não são óbvias.

```
char seq3[] = {
      'A','B','C','A','B','C','D','A','B','C','E','F','A','B','C','D',
      'X','Y','Z','X','X','Y','X','X','Z','X','Y','X','X','Z','X','X',
      'M','N','O','M','P','N','O','Q','M','N','O','P','Q','R','M','N',
      'A','B','C','A','D','E','A','B','C','D','E','F','A','B','C','D',
      'X','Y','Z','X','Y','Z','X','X','Y','Z','X','Y','Z','M','N','O',
      'P','Q','R','S','T','U','V','W'
  };
```

Essa sequência com 88 blocos possui seções com alta localidade temporal (‘A’ ao ‘F’), favorecendo LRU, e partes favorecendo a LFU, em que os blocos ‘X’, ‘Y’ e ‘Z’ são acessados repetidamente. Também contém varreduras com os elementos ‘M’ ao ‘W’. Os resultados foram:

```
Sequência de acessos 3:
LRU: 53 hits e 35 misses
LFU: 31 hits e 57 misses
```

Nesse cenário, a LRU performou melhor por reagir rapidamente às mudanças nos padrões de acesso, embora o seu ganho de desempenho contra a LFU tenha sido menor em relação ao caso 1, projetado especialmente para favorecer a política Least Recently Used. Observa-se que para essa quantidade de elementos, consideravelmente pequena e sem espaço para sequências muito longas de acessos repetidos (em que LFU se sai melhor), a LRU torna-se mais vantajosa.