Instituto Tecnológico de Aeronáutica

Divisão de Engenharia de Computação

CES-33 Sistemas Operacionais

Projeto Exame

Sistemas de Hierarquia de Memória

Alunos:
Gabriel Ilharco Magalhães

Prof. responsável:
Paulo André Castro

30 de junho de 2016

1 Objetivo

O presente trabalho tem por objetivo a criação e implementação de um simulador de sistema de memórias. O código desenvolvido permite que sejam criados sistemas com um número qualquer de níveis de cache, com parâmetros variáveis (tamanho, tamanho do bloco, associatividade, tempo de acesso de 1 palavra, tempo de acesso de 1 tag) e políticas de gravação variáveis. Em todas as memórias, a política de substiuição de blocos foi implementado através de fila (FIFO).

Para a implementação do simulador, utilizou-se a linguagem C++, que foi escolhida por sua grande velocidade de execução.

O programa desenvolvido foi testado para duas arquiteturas similares ao do Core 2 Duto T7200, ambas com dois níveis de cache, nas quais pode-se determinar as taxas de acerto em cada nível utilizando um programa de benchmark.

2 Descrição

O sistema implementado foi testado para 2 níveis de cache e um endereçamento de 32 bits. Abaixo segue a descrição dos parâmetros de cada um dos níveis de memória, bem como as políticas de gravação e substituição.

Tamanho da Cache L1	32KB Dados (+32KB Instruções)
Tamanho do bloco (ou linha) em L1	64 bytes
Política de Gravação em L1	Write Through
Política de Substituição L1	FIFO
Associatividade L1	Associativo de 8 vias
Tempo de Acesso de 1 Palavra em L1	2 clocks
Tempo de Comparação de 1 Tag	1 (clock)
Outras características L1	Write Allocate
Tamanho da Cache L2	4096KB
Tamanho do bloco (ou linha) em L2	64 bytes
Política de Gravação em L2	Write Back
Política de Substituição L2	FIFO
Associatividade L2	Associativo de 16 vias
Tempo de Acesso de 1 Palavra em L2	4 clocks
Tempo de Comparação de 1 Tag	2 (clocks)
Outras características L2	Write Not Allocate
Tempo de acesso à memória	60 (clocks)

Para esses dados, seguem a tabela abaixo mostra a descrição das situações possíveis que ocorrem no conjunto de memórias quando ocorre uma nova operação. Preferiu-se juntar as tabelas propostas no roteiro em uma única, para mais fácil visualização do tempo total gasto em cada situação e a descrição do que acontece em cada um dos componentes.

Considerou-se que, com o write trough, tenta-se gravar simultaneamente na cache atual e na memória inferior. Também, considerou-se que todas as buscas ocorrem em paralelo. Implementou-

se a aproximação do LRU utilizando a política FIFO (First In - First Out). Além disso, foi considerado que todas as tags em um segmento podem ser comparadas simultaneamente.

Para controle, é necessário que, para cada bloco, seja armazenado a sua tag. Desse modo, como em L1 existem 32KB/64B = 512 blocos, e cada tag possui 52 bits (utilizando endereçamento de 64 bits, 6 bits são correspondentes ao offset, pois cada bloco possui 64 bytes, e 6 bits são correspondentes ao set, pois cada set contém 8*64 = 512 bytes, então L1 possui 32K/512 = 64 sets), temos um total de memória de controle para L1 de 512*52=26.6KB.

Analogamente, para L2, existem 4GB/64B = 64K blocos, e cada tag possui 52 bits (utilizando endereçamento de 64 bits, 6 bits correspondem ao offset, pois cada bloco possui 64 bytes, e 6 bits são correspondentes ao set, pois cada set contem 16*64=1KB, então L1 possui 64K/1K = 64 sets), temos um total de memória de controle para L2 de 64K*64=4GB.

Como sugestão de alteração às configurações do sistema original, sugeriu-se aumentar a associatividade da cache do segundo nível, de 16 vias para 64 vias.

Operação	L1	L2	Memória	Tempo total	Descrição
Leitura	Hit	-	-	1+2=3	Acerto em L1. L1 entrega dados à CPU. Não há tempo adicional
Leitura	Miss	Hit	-	2+4=6	Erro em L1, acerto em L2. L2 entrega dados à CPU e à L1. Note que, devido à política Write-Trough de L1, não é possível que a substituição de bloco gere gravação em L2.
Leitura	Miss	Miss	Hit	60(+60)	Erro em L1 e L2, acerto na memória principal. Memória entrega dados à CPU, à L1 e à L2. Devido à política Write-Trough de L1, não é possível que uma substituição de bloco gere gravação em L2. Devido à política Write-Back de L2, é possível que uma substituição de bloco em L2 gere gravação na memória (se o bloco substituído estiver sujo). Nesse caso, temos 60 clocks de tempo adicional.
Escrita	Hit	Hit	-	2+4=6	Acerto em L1. L1 entrega dados à CPU. Devido à política de Write Through, necessariamente faço gravação em L2 (tempo pago: 2+4, considerando operação em paralelo) Note que, como o bloco já está em L2, não existe substituição em L2, então não é possível pagarmos tempo adicional

Operação	L1	L2	Memória	Tempo total	Descrição
Escrita	Miss	Hit	-	2+4=6	Erro em L1, acerto em L2. CPU entrega dados à L1 e L2 simultaneamente. Devido à política Write-Back de L2, não há tempo adicional de acesso à memória principal. Devido à política Write-Allocate de L1, o bloco também é gravado em L1 (tempo adicional: 0, considerando operação em paralelo). Note que uma possível substituição em L1 não pode causar gravação em L2, devido à política de Write-Through.
Escrita	Miss	Miss	Hit	60	Erro em L1 e L2, acerto na memória principal. Devido à política Write-Not-Allocate de L2, o bloco não é gravado em L2. Devido à política Write-Allocate de L1, o bloco é gravado em L1 (tempo adicional: 0, considerando operação em paralelo).
Escrita	Hit	Miss	Hit	60	Acerto em L1. Devido à política Write-Trough de L1, tenta gravar em L2, o que causa falha e escrita na memória principal (tempo adicional: 60). Devido à política Write-Not-Allocate de L2, o bloco não é gravado em L2.

Para cada uma das configurações (L2 com associatividade de 16 vias e de 64 vias), testou-se o programa com o benchmark fornecido, e mediu-se o tempo total de execução e as estatísticas de cache miss para cada um dos níveis de cache. Os resultados e análise seguem na seção seguinte.

O código contendo a implementação das operações descritas pela tabela acima é mostrado a seguir

Código principal da aplicação:

```
#include <stdio.h>
#include "Memory.h"

int main () {
  fflush(stdin);
  Memory * null = NULL;
  Memory * RAM = new Memory(
    (long long) 4294967296, // long long size (2^32)
    64,                     // long long block_size
    (long long) 4294967296, // long long main_memory_size
    4194304,                // long long associativity_set_size
    null,                   // Memory * lower_level_memory
    60,                     // double word_access_time
    0,                      // double tag_compare_time
```

```
    WRITE_BACK,                 // int write_miss_policy
    WRITE_ALLOCATE              // int write_hit_policy
  );
  Memory * L2 = new Memory(
    4194304,                    // long long size
    64,                         // long long block_size
    (long long) 4294967296,     // long long main_memory_size
    64,                         // long long associativity_set_size
    RAM,                        // Memory * lower_level_memory
    4,                          // double word_access_time
    2,                          // double tag_compare_time
    WRITE_BACK,                 // int write_miss_policy
    NO_WRITE_ALLOCATE           // int write_hit_policy
  );
  Memory * L1 = new Memory(
    32768,                      // long long size
    64,                         // long long block_size
    (long long) 4294967296,     // long long main_memory_size
    8,                          // long long associativity_set_size
    L2,                         // Memory * lower_level_memory
    2,                          // double word_access_time
    1,                          // double tag_compare_time
    WRITE_TROUGH,               // int write_miss_policy
    WRITE_ALLOCATE              // int write_hit_policy
  );

  FILE * file = fopen("gcc.trace", "r");
  long long address;
  char rw;
  double time = 0;
  bool success;
  while (fscanf (file, "%llx %c", &address, &rw) != EOF) {
    if (rw == 'W' || rw == 'w') {
      L1->writePage(address, success, time);
    }
    else {
      L1->getBlock(address, success, time);
    }
  }
  printf ("\n\n-----------------------------------------\nTotal time: %g\n", time);
  printf ("L1 miss rate: %.4lf\n", (double)L1->misses/L1->total_op);
  printf ("L2 miss rate: %.4lf\n", (double)L2->misses/L2->total_op);
  return 0;
}
```

Código implementado das memórias:

```
//
// Created by gabrielilharco on 17/06/2016
//

#include "Memory.h"

Block::Block(long long address) : address(address) {
  dirty = false;
}

Block::Block() : address(0) {
  dirty = false;
}

Memory::Memory (
  long long size,
  long long block_size,
  long long main_memory_size,
  int associativity_set_size,
  Memory * lower_level_memory,
  double word_access_time,
  double tag_compare_time,
  int write_miss_policy,
  int write_hit_policy
) :
  size(size),
  block_size(block_size),
  main_memory_size(main_memory_size),
  associativity_set_size(associativity_set_size),
  lower_level_memory(lower_level_memory),
  word_access_time(word_access_time),
  tag_compare_time(tag_compare_time),
  write_miss_policy(write_miss_policy),
  write_hit_policy(write_hit_policy)
{
  misses = 0;
  total_op = 0;
  blocks = vector<Block>(size/block_size);
  // initializing blocks
  for (int i = 0; i < size/block_size; i++) {
    if (!lower_level_memory)
      blocks[i] = Block(getBlockAddress(block_size*i));
    else
```

```
        blocks[i] = -1;
    }
    // initializing indexes for substituition
    for (int i = 0; i < (size/block_size)/associativity_set_size; i++)
      current_set_index.push_back(0);
}

Block Memory::getBlock(long long page_address, bool& success, double& time) {
    total_op++;
    // get block address from page address
    long long block_address = getBlockAddress(page_address);
    // check if the block exists in this memory
    int block_index = getBlockFromCurrentMemory(block_address, success, time);
    Block block = blocks[block_index];
    if (success) {
      // cache hit, just update variables and return block
      time += word_access_time + tag_compare_time;
      return block;
    }
    misses++;
    // if memory is a cache memory (not the lowest level)
    if (lower_level_memory) {
      // cache miss, we need to update the memory
      // get block from lower level memory
      Block block = lower_level_memory->getBlock(page_address, success, time);
      // store block in current memory
      // also deals with the case of a block leaving current memory;
      substitute(block, success, time);
      // return the block
      return block;
    }
    // lowest level memory
    time += word_access_time;
    return Block(block_address);

}

Block Memory::writePage(long long page_address, bool& success, double& time) {
    // get block address from page address
    long long block_address = getBlockAddress(page_address);
    // check if the block exists in this memory
    int block_index = getBlockFromCurrentMemory(block_address, success, time);
    Block block = blocks[block_index];
    // make block dirty
```

```
  if (lower_level_memory)
    blocks[block_index].dirty = true;
  if (success) {
    //cache hit
    // if memory is write trough, we need to write it in lower level memory
    if (write_hit_policy == WRITE_TROUGH && lower_level_memory)
      lower_level_memory->writePage(page_address, success, time);
    else
      time += word_access_time + tag_compare_time;
    // return the block
    return block;
  }
  // cache miss
  if (lower_level_memory) {
    // get block from lower level memory
    Block block2 = lower_level_memory->writePage(page_address, success, time);
    // if write miss policy is write allocate, store block in current memory
    // also deals with the case of a block leaving current memory;
    if (write_miss_policy == WRITE_ALLOCATE) {
      // substitute
      substitute(block2, success, time);
    }
    // return the block
    return block2;
  }
  //printf ("RAM hit\n"); fflush(stdin);
  return Block(block_address);
}

void Memory::substitute (Block block, bool& success, double& time) {
  // number of sets on current memory
  long long n_sets = (size/block_size)/associativity_set_size;
  // get in which set this blocks belong to
  long long set_address =  (block.address*block_size)/(main_memory_size/n_sets)
  // beginning of associativity set
  long long start = set_address*associativity_set_size;
  // block index
  long long block_index = start + current_set_index[set_address]%block_size;
  // get block to be substituted
  Block old_block = blocks[block_index];

  // put new block
  blocks[block_index] = block;
  // a new block is never dirty
```

```
  blocks[block_index].dirty = false;
  // increment current_set_index;

  // if write hit policy is write-back, check if block is dirty
  if (write_hit_policy == WRITE_BACK && old_block.dirty && lower_level_memory &&
    // we need to make sure lower level memory updates it
    lower_level_memory->writePage(getSamplePage(old_block.address), success, tim
  }

  current_set_index[set_address]++;

}

// gets index of block (if it exists) on current memory
int Memory::getBlockFromCurrentMemory(long long block_address, bool& success, d
  // number of sets on current memory
  long long n_sets = (size/block_size)/associativity_set_size;
  // get in which set this blocks belong to
  long long set_address =  (block_address*block_size)/(main_memory_size/n_sets)
  // beginning of associativity set
  long long start = set_address*associativity_set_size;
  // end of associativity set
  long long end = start + associativity_set_size;
  // we need to check for every possible position
  for (long long i = start; i < end; i++) {
    if (blocks[i].address == block_address) {
      // success, just return the block
      success = true;
      return i;
    }
  }
  // block not in current memory. Return dummy block
  success = false;
  return -1;
}

// gets the address of the block that colong longains a page passed as paramete
long long Memory::getBlockAddress(long long page_address) {
  long long number_of_blocks = main_memory_size/block_size;
  int tag_size = log2(number_of_blocks);
  int memory_size = log2(main_memory_size);
  return page_address >> memory_size-tag_size;
}
```

```
long long Memory::getSamplePage(long long block_address) {
  long long n_sets = (size/block_size)/associativity_set_size;
  return (size/n_sets) * block_address/(main_memory_size/n_sets);
}
```

3 Resultados e análise

Executou-se o programa para o benchmark com ambas as configurações, tanto para L2 com associatividade de 16 vias e com associatividade de 64 vias. Com essa alteração, esperava-se menos falhas em L2, o que deveria diminuir o tempo de execução do benchmark. A seguir, seguem as estatísticas para cada uma das configurações.

Associatividade de 16 vias (L2)

- Total time: $1.89e+07$ (clocks)
- L1 miss rate: 62.71%
- L2 miss rate: 42.23%

Associatividade de 64 vias (L2)

- Total time: $0.93e+07$ (clocks)
- L1 miss rate: 62.71%
- L2 miss rate: 11.18%

Conforme esperado, a taxa de falhas em L2 diminuiu, o que fez com que o tempo total de execução caisse consideravelmente (em torno de 50%).

4 Conclusões

O presente trabalho foi importante para consolidar os conceitos de memória e seu gerenciamento, políticas de gravação e substituição e outros conceitos importantes de sistemas de memória.

Particularmente, considerei o projeto de dificuldade moderada, porém muito instrutivo. Como sugestão, deixo o alinhamento com a disciplina de CES-33, na qual fizemos dois projetos relacionados à memória - o curso de CES-25 é bem extenso e possui uma grande gama de assuntos que renderiam ótimos projetos. Tenho ciência de que os projetos surgiram no final do semestre, mas acredito que, para um próximo ano, fosse interessante que fosse cobrado outro assunto no projeto de CES-25, para uma maior cobertura da matéria.

No geral, fica um elogio à ideia de realizar um projeto como exame, e a ideia de se fazer mais laboratórios durante o semestre. É muito interessante realmente programar o que se vê na teoria, e nos faz entender o conteúdo programático de uma maneira mais profunda e persistente.

Apêndice: Arquivo header do projeto (.h)

```
//
// Created by gabrielilharco on 17/06/16.
//

#ifndef MMS_MEMORY_H
#define MMS_MEMORY_H

#include <vector>
#include <math.h>

using namespace std;

// Write miss techniques
#define WRITE_ALLOCATE    1
#define NO_WRITE_ALLOCATE   2

// Write hit techniques
#define WRITE_TROUGH    1
#define WRITE_BACK      2

class Block {
public:
  bool dirty;
  long long address;

  Block(long long address);
  Block();
};

class Memory {
private:
  // last operation status and time
  bool last_op_status;
  long long last_op_time;
  // current index for each set (for block replacement)
  vector<long long> current_set_index;

  void substitute (Block block, bool& success, double& time);
  int getBlockFromCurrentMemory(long long block_address, bool& success, double&
  long long getBlockAddress(long long page_address);
  long long getSamplePage(long long block_address);
public:
```

```
    int misses;
    int total_op;

    // memory blocks
    vector<Block> blocks;

    // memory size in bytes
    long long size;
    // block size in bytes
    long long block_size;
    // main memory size in bytes
    long long main_memory_size;
    // N for a N-way set associative policy
    int associativity_set_size;  // 1 for diret mapping and <size> for fully asso
    // lower level memory
    Memory * lower_level_memory;
    // time necessary to access a word in the memory
    double word_access_time;
    // time necessary to compare two tags
    double tag_compare_time;
    // write miss policy
    int write_miss_policy;
    // write hit policy
    int write_hit_policy;

    Memory(
      long long size,
      long long block_size,
      long long main_memory_size,
      int associativity_set_size,
      Memory * lower_level_memory,
      double word_access_time,
      double tag_compare_time,
      int write_miss_policy,
      int write_hit_policy
    );
    Block getBlock(long long page_address, bool& success, double& time);
    Block writePage(long long page_address, bool& success, double& time);

};

#endif //MMS_MEMORY_H
```