Mapeamento de Memória Cache

Organização e Arquitetura de Computadores

Henrique Braz, Joseph Weber e Thomas Pozzer

Junho de 2019

Introdução

Este trabalho busca apresentar uma alternativa de solução ao terceiro problema da disciplina de Organização e Arquitetura de Computadores do terceiro semestre do curso de Engenharia de Software. O problema pode ser resumido da seguinte forma: dado um programa em linguagem de montagem, e seus respectivos endereços gerados. Devemos, apresentar a quantidade de acertos e erros, além do conteúdo da cache ao final da execução.

São dadas quatro configurações diferentes para a memória cache:

- Mapeamento direto, com 8 bits para tag, 4 bits para linha, 3 bits para palavra e 1 bit para byte (cache com 16 linhas, 8 palavras por linha).
- Mapeamento direto, com 8 bits para tag, 5 bits para linha, 2 bits para palavra e 1 bit para byte (cache com 32 linhas, 4 palavras por linha).
- Mapeamento associativo, com 12 bits para tag, 3 bits para palavra e 1 bit para byte (cache com 16 linhas, 8 palavras por linha).
- • Mapeamento associativo, com 13 bits para tag, 2 bits para palavra e 1 bit para byte (cache com 32 linhas, 4 palavras por linha).

1 Endereços

1.1 Endereços de hexadecimal e binário

Como já tinhamos todos os endereços da execução do programa, nosso primeiro passo foi fazer um algoritmo utilizando a linguagem Python que transformasse os endereços passados(hexadecimal) em binários como visto em sala de aula.

```
def hexa_binario(arquivo_hexa, hexa_binario):
        """
    ||        Metodo que le o arquivo inicial com os dados em hexa
    ||        param arquivo_hexa: caminho do arquivo inicial com os numeros
    ||em hexa a ser lido
    ||        param hexa_binario: caminho do arquivo onde sera salvo os
    ||dados no formato  hexa -> binario
    ||       :return: arquivo binario.txt com os dados formatados e em
    ||binario
```

```
    """
    lista_binario = []
    arq = open(arquivo_hexa, 'r')
    arq2 = open(hexa_binario,'w')
    arquivo = arq.readlines()
    a = []
    for i in range(len(arquivo)):
        a.append(arquivo[i].split(','))
        for j in range(len(a[i])):
            b = ((a[i][j]).strip())
            if b != '':
                x = bin(int(b, 16))[2:].zfill(16)
                lista_binario.append(x)
                arq2.write(b +' ---> '+x +'\n')

    arq.close()
    arq2.close()
    return(lista_binario)
```

O metodo acima retorna uma lista das posições em binário convertidos. Ele armazena essa lista em um arquivo binario.txt que utilizamos para formular os resultados dos exercicios no decorrer do relatório.

2 Mapeamento direto

Para o mapeamento direto, usamos uma lista de lista para armazenar os endereços da memória.

O programa detectará um *hit* caso encontre naquela linha a mesma tag do registrador. Em caso de *miss*, o programa irá adicionar a linha e a tag e carregará as palavras subsequentes para aquela linha.

Para a implementação no algoritmo as palavras subsequentes são definidas pela concatenação da tag com a linha, um bit validade e um bit para byte.

Uma possível implementação do algoritmo está descrita abaixo.

2.1 Tag: 9 bits, linha: 4 bits, palavra: 3 bits, bitByte: 1 bit

2.1.1 Estatísticas

Quantidade de *Hits*: **250**
Quantidade de *Miss*: **14**
Porcentagem de acerto: **94.7**

```
def exercicio1(lista_binario,exercicio1):
"""
//     Funcao que faz a analise da cash
//     :param lista_binario: lista convertida de hexa para binario
//     :param exercicio1: caminho do arquivo.txt onde sera salvo
//     os dados da analise
//     utiliza da funcao organiza_palavra para inserir os ultimos
//     bits nas posicoes do bloco
//     utiliza da funcao cria_cash para criar a cash vazia,
```

```
//    com 16 linhas e 4 casas
    """
    cash = []
    cash = cria_cash1(16,4) #cria a cash com 16 linhas
    indice = 0
    hit = 0
    miss = 0
    arq = open(exercicio1,'w')
    arq.write('\nlinha, bit_validade, tag, palavra0, palavra1, palavra2, 
    for i in range(len(lista_binario)):
        x = str(lista_binario[i])
        tag = x[0:8]
        linha  = x[8:12]
        palavra = x[0:15]
        b_byte = x[15]
        for j in range(len(cash)):

            if linha in cash[j]:
                indice = j #achei a linha: cash[indice]

        if cash[indice][2] != tag:

            cash[indice][1] = '1'
            cash[indice][2] = tag
            cash[indice][3] = organiza_palavra(palavra,3)[0]
            cash[indice][4] = organiza_palavra(palavra,3)[1]
            cash[indice][5] = organiza_palavra(palavra,3)[2]
            cash[indice][6] = organiza_palavra(palavra,3)[3]
            cash[indice][7] = organiza_palavra(palavra,3)[4]
            cash[indice][8] = organiza_palavra(palavra,3)[5]
            cash[indice][9] = organiza_palavra(palavra,3)[6]
            cash[indice][10] = organiza_palavra(palavra,3)[7]
            cash[indice][11] = b_byte
            cash[indice][12] = 'Miss'
            miss += 1

        elif cash[indice][2] == tag:

            cash[indice][11] = b_byte
            cash[indice][12] = 'Hit'
            hit += 1
```

2.2 Tag: 8 bits, linha: 5 bits, palavra: 2 bits, bitByte: 1 bit

2.2.1 Estatísticas

Quantidade de *Hits*: **241**
Quantidade de *Miss*: **23**
Porcentagem de acerto: **91.29**

```
def exerciocio2(lista_binario,exercicio2):
```

```
    """
//    Metodo que faz a analise da cash
//    :param lista_binario: lista convertida de hexa para binario
//    :param exercicio2: caminho do arquivo.txt onde sera salvo os
//dados da analise
//    utiliza do metodo organiza_palavra para inserir os ultimos bits
//nas posicoes do bloco
//    utiliza do m todo cria_cash para criar a cash vazia
    """
    cash = []
    cash = cria_cash2(32,5) #cria a cash com 32
    indice = 0
    hit = 0
    miss = 0
    arq = open(exercicio2,'w')
    arq.write('\nlinha, bit_validade, tag, palavra0, palavra1, palavra2,
    for i in range(len(lista_binario)):
        x = str(lista_binario[i])
        tag = x[0:8]
        linha  = x[8:13]
        palavra = x[0:15]
        b_byte = x[15]
        for j in range(len(cash)):

            if linha in cash[j]:
                indice = j #achei a linha: cash[indice]

        if cash[indice][2] != tag:

            cash[indice][1] = '1'
            cash[indice][2] = tag
            cash[indice][3] = organiza_palavra(palavra,2)[0]
            cash[indice][4] = organiza_palavra(palavra,2)[1]
            cash[indice][5] = organiza_palavra(palavra,2)[2]
            cash[indice][6] = organiza_palavra(palavra,2)[3]
            cash[indice][7] = b_byte
            cash[indice][8] = 'Miss'
            miss += 1

        elif cash[indice][2] == tag:

            cash[indice][7] = b_byte
            cash[indice][8] = 'Hit'
            hit += 1
```

3 Mapeamento Associativo

Para a construção do mapeamento associativo utiliza uma lista de listas para armazenar os endereços da memória.

As palavras carregadas serão obtidas da mesma maneira que no mapeamento direto, porém agora concatenando apenas tag e incremento.

Inicialmente o algoritmo verifica se a cache está cheia e se estiver cheia, criamos uma condição de busca pela TAG. O programa acusa *hit* caso encontre a tag na cache e *miss* caso não encontre.

Caso a cache esteja cheia, ele utiliza a politica de substituição abaixo para sobrescrever a posição com a nova informação.

3.1 Política utilizada

A política de substituição de dados na cache utilizada foi *Contador*, caracterizada por substituir a posição da cache apontada pela posição *contador + 1*. Esse contador é iniciado com 0 e toda a vez que ele é utilizado ele vai incrementado 1 até a ultima posição da cache. Quando chega na posição final, ele retorna ao inicio da cache, ou seja, zera o contador.

3.2 Tag: 12 bits, palavra: 3 bits, bitByte: 1 bit

3.2.1 Estatísticas

Quantidade de *Hits*: **252**
Quantidade de *Miss*: **12**
Porcentagem de acerto: **98.43**

3.3 O algoritmo

```
def exercicio3(lista_binario,exercicio3):
    """
//    Metodo que faz a analise da cash
//    :param lista_binario: lista convertida de hexa para binario
//    :param exercicio2: caminho do arquivo.txt onde sera salvo os
//dados da analise
//    utiliza do metodo organiza_palavra para inserir os ultimos bits
//nas posicoes do bloco
//    utiliza do metodo cria_cash para criar a cash vazia
//    palavra0, palavra1, palavra2, palavra3,palavra4, palavra5,
//palavra6, palavra7
    """
    cash = []
    cash = cria_cash3(16,4) #cria a cash com 16 linhas
    hit = 0
    miss = 0
    cont2 = 0
    arq = open(exercicio3,'w')
    arq.write('\nlinha,␣tag,␣linha,␣palavra0,␣palavra1,␣palavra2,␣palavra
    for i in range(len(lista_binario)):
        x = str(lista_binario[i])
        tag = x[0:12]
        palavra = x[0:15]
        b_byte = x[15]
```

```
cont = 0

for k in range(len(cash)):
    if cash[k][1] != '':
        cont += 1

if cont == len(cash) and tag not in cash:
    cash[cont2][1] = tag
    cash[cont2][3] = organiza_palavra(palavra,3)[0]
    cash[cont2][4] = organiza_palavra(palavra,3)[1]
    cash[cont2][5] = organiza_palavra(palavra,3)[2]
    cash[cont2][6] = organiza_palavra(palavra,3)[3]
    cash[cont2][7] = organiza_palavra(palavra,3)[4]
    cash[cont2][8] = organiza_palavra(palavra,3)[5]
    cash[cont2][9] = organiza_palavra(palavra,3)[6]
    cash[cont2][10] = organiza_palavra(palavra,3)[7]
    cash[cont2][11] = b_byte
    cash[cont2][12] = 'Miss'
    miss += 1
    if cont2 < len(cash):
        cont2 += 1
    else:
        cont2 = 0
    cont = 0

else:
    for j in range(len(cash)):
        if tag == cash[j][1]:
            cash[j][11] = b_byte
            cash[j][12] = 'Hit'
            hit += 1
            break

        elif cash[j][1] == '':
            cash[j][1] = tag
            cash[j][3] = organiza_palavra(palavra,3)[0]
            cash[j][4] = organiza_palavra(palavra,3)[1]
            cash[j][5] = organiza_palavra(palavra,3)[2]
            cash[j][6] = organiza_palavra(palavra,3)[3]
            cash[j][7] = organiza_palavra(palavra,3)[4]
            cash[j][8] = organiza_palavra(palavra,3)[5]
            cash[j][9] = organiza_palavra(palavra,3)[6]
            cash[j][10] = organiza_palavra(palavra,3)[7]
            cash[j][11] = b_byte
            cash[j][12] = 'Miss'
            miss += 1
            break
```

3.4 Tag: 13 bits, palavra: 2 bits, bitByte: 1 bit

3.4.1 Estatísticas

Quantidade de *Hits*: **243**
Quantidade de *Miss*: **21**
Porcentagem de acerto: **94.92**

3.5 O algoritmo

```
def exercicio4(lista_binario,exercicio3):
    """
//    Metodo que faz a analise da cash
//    :param lista_binario: lista convertida de hexa para binario
//    :param exercicio2: caminho do arquivo.txt onde sera salvo os
//dados da analise
//    utiliza do metodo organiza_palavra para inserir os ultimos bits
//nas posicoes do bloco
//    utiliza do metodo cria_cash para criar a cash vazia
//    palavra0, palavra1, palavra2, palavra3,palavra4, palavra5,
//palavra6, palavra7,
    """
    cash = []
    cash = cria_cash4(32,5) #cria a cash com 32
    hit = 0
    miss = 0
    cont2 = 0
    arq = open(exercicio3,'w')
    arq.write('\nlinha, tag, linha, palavra0, palavra1, palavra2, bit par
    for i in range(len(lista_binario)):
        x = str(lista_binario[i])
        tag = x[0:13]
        palavra = x[0:15]
        b_byte = x[15]
        cont = 0

        for k in range(len(cash)):
            if cash[k][1] != '':
                cont += 1

        if cont == len(cash) and tag not in cash:
            cash[cont2][1] = tag
            cash[cont2][3] = organiza_palavra(palavra,3)[0]
            cash[cont2][4] = organiza_palavra(palavra,3)[1]
            cash[cont2][5] = organiza_palavra(palavra,3)[2]
            cash[cont2][6] = organiza_palavra(palavra,3)[3]
            cash[cont2][7] = b_byte
            cash[cont2][8] = 'Miss'
            miss += 1
            if cont2 < len(cash):
                cont2 += 1
```

```
            else:
                cont2 = 0
            cont = 0

        else:
            for j in range(len(cash)):
                if tag == cash[j][1]:
                    cash[j][7] = b_byte
                    cash[j][8] = 'Hit'
                    hit += 1
                    break

                elif cash[j][1] == '':
                    cash[j][1] = tag
                    cash[j][3] = organiza_palavra(palavra,3)[0]
                    cash[j][4] = organiza_palavra(palavra,3)[1]
                    cash[j][5] = organiza_palavra(palavra,3)[2]
                    cash[j][6] = organiza_palavra(palavra,3)[3]
                    cash[j][7] = b_byte
                    cash[j][8] = 'Miss'
                    miss += 1
                    break

        arq.write('\n'+str(cash[j])+'\n')

    arq.write('\nCash, resultado final: \n')
    arq.write('\nlinha, tag, linha, palavra0, palavra1, palavra2, palavra
    arq.write('\n'+str(cash)+'\n')
    arq.write('\nTotal de Cash Hit:\n')
    arq.write(str(hit) +'\n')
    arq.write('Total de Cash Miss:\n')
    arq.write(str(miss))
    arq.close()
```

Conclusão

Diante do problema apresentado e da interpretação para a resolução do mesmo, se desenvolveu um algoritmo para a automatização de alguns passos necessários para a conclusão da tarefa.

A obtenção dos endereços em hexadecimal gerados através do simulador Viking, a conversão destes endereços para binário e os mapeamentos diretos e associativos (sendo o LRU a política adotada para este último mapeamento), foram encontrados em menos de um segundo, o que seria impossível se a tarefa fosse realizada a mão.

Como resultado, foram encontrados 252 *Hits* e 36 *Miss* no primeiro mapeamento direto (tag: 9 bits, linha: 4 bits, palavra: 2 bits), com uma porcentagem de acerto de 87,5%. No segundo mapeamento direto (tag: 9 bits, linha: 5 bits, palavra: 1 bit), a quantidade de *Hits* foi de 226 e a quantidade de *Miss* chegou a 62, gerando uma porcentagem de acerto um pouco menor, de 78,5%.

Já nos mapeamentos associativos, o primeiro (tag: 13 bits, palavra: 2 bits) obteve 243 *Hits* e 45 *Miss*, o que acarretou em 84,4% de acerto. Por fim, para o último mapeamento

(tag: 14 bits, palavra: 1 bit), também associativo, foram encontrados apenas 204 $Hits$ e 84 $Miss$, tendo este a menor porcentagem de acerto dentre os 4 mapeamentos analisados, com 70,8%.