**INSTITUTO TECNOLÓGICO DE AERONÁUTICA - ITA**

DIVISÃO DE CIÊNCIA DA COMPUTAÇÃO

**CES-25 – Arquiteturas para Alto Desempenho**

****

**Projeto Exame**

**Simulador de Sistema de Memória**

**Com dois níveis de cache**

**ALUNO**

Felipe Tuyama de Faria Barbosa ftuyama@gmail.com

**PROFESSORA**

Paulo André Lima de Castro pauloac@ita.br

São José dos Campos, 30 de Junho de 2016

**1. INTRODUÇÃO**

O objetivo deste projeto é implementar um simulador de sistema de memória com dois níveis de cache. Através deste, será possível ao desenvolvedor familiarizar e comparar diferentes configurações de arquitetura (quantitativamente), explorando as soluções vistas em aula para as propriedades principais das caches. Dentre elas, seu tamanho físico, estratégias e políticas de gravação e substituição de blocos.

O simulador foi implementado na linguagem de alto nível Python, uma vez que não é exigido alto desempenho para o programa, neste contexto. No contexto industrial, no entanto, seria recomendável um desenvolvimento mais eficiente, uma vez que a execução de múltiplos benchmarks pode ser demorada.

**2. DESCRIÇÃO**

Da definição de simulador: *“Para ser realizada uma simulação, é necessário construir um modelo computacional que corresponda à situação real que se deseja simular”[1]*, temos a palavra-chave “modelo computacional”.

Ou seja, é preciso descrever os comportamentos que se assemelham à situação real e também adotar premissas que representem o fenômeno a ser simulado. No caso deste simulador, temos o código fonte que descreve o comportamento do sistema de memória e também uma tabela de tempo gasto com cada operação, que representa as premissas adotadas para o sistema.

**2.1 – Descrição do tempo gasto**

Assim, preenchendo as Tabelas de tempos necessários para a realização de cada uma das operações possíveis, temos o seguinte cenário de premissas adotadas pelo simulador:

Tabela 2. Operações causadas por acesso ao Sistema de Memória.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Operação causada por | |  | Situação | |  |
| acesso ao Sist.Mem. | | Acerto em L1 | Acerto em L2 | | Acerto na Memória |
| Leitura | | 1+2=3 |  |  |  |
| Leitura | | Não(\*A) | 2+4=6 | |  |
| Leitura | | Não(\*B) | Não(\*C) | | 60 |
| Escrita | | 2+4=6 (WT) |  |  |  |
| Escrita | | Não(\*A) | 2+4=6 (WB) | |  |
| Escrita | | Não(\*D) | Não(\*E) | | 60 |
| Tabela 3. Operações causadas por subida de de bloco | | | | | |
| Operação causada |  |  | Situação | |  |
| por subida de bloco |  |  |  |  |  |
|  | Descrição |  |  | Tempo Adicional |
| (\*ID) |  |  |  |
|  |  |  |  |  |
| (\*A) | Atualização apenas em L1; L2 entrega  dados a CPU e a L1 | | | 0 | |
|  |
| (\*B) | Atualização em L1. | | | 0 | |
| (\*C) | Política de substituição WB encadeia escrita em L2, se necessário (WB). | | | 60 (para bloco sujo) - WB  0 (para bloco limpo) | |
| (\*D) | Atualização em L1. | | | 0 | |
| (\*E) | Atualização apenas em L1, pois L2 é cache WNA (escreve apenas na memória) | | | 0 (bloco limpo ou sujo) | |

(\*ID) Subida de bloco determinada por situação indicada na Tabela 2.

Na situação (\*B)/(\*C), temos uma leitura na memória principal com falhas nas caches L1 e L2. Para a cache L1, basta realizar a substituição sem preocupações adicionais, graças à política de gravação Write Through (WT). Porém, para a cache L2, deve-se checar o bit M (o qual indica bloco sujo, incoerente com o bloco na memória principal). Caso positivo, deve-se realizar o Write Back (WB), que custará o tempo de acesso de escrita à memória principal, além da leitura a ser realizada originalmente.

Esse tempo é adicional pois primeiro deve-se fazer a gravação de um dado bloco X sujo na memória (tempo adicional 60) para só então realizar a leitura do bloco Y na memória (tempo 60) cujo endereço foi solicitado, fazendo sua substituição na cache L2.

Na situação (\*D)/(\*E), temos uma escrita na memória principal com falhas nas caches L1 e L2. Para a cache L1, basta realizar a substituição sem preocupações adicionais, graças à política de gravação Write Through (cache sempre coerente com nível inferior). Porém a cache L2 tem política de gravação em falha Write Not Allocate (WNA), de modo não é necessário realizar substituições de bloco em L2 (dispenso verificações do bit M e possíveis operações WB).

No caso de escrita a cache L2 é WNA, de forma que não é preciso subir o bloco solicitado da memória para L2, sendo necessário somente a sua escrita na memória (tempo 60) sem tempos extras.

**2.2 Descrição do espaço gasto**

Uma observação importante é o sobre o tamanho *real* disponível para dados. Para o desenvolvimento do simulador, foi considerado o tamanho declarado no roteiro para cada uma das caches.

No entanto, foi necessário incluir um bit M (modified) adicional para cada bloco de L2 (4096KB/64bytes = 65536 blocos -> 65536 bits = 8 KB), a fim de determinar se aquele bloco sofreu modificações desde que foi trazido da memória principal, sendo assim necessária a operação de Write Back quando ocorrer sua substituição. Esse espaço extra pode ser desprezado na cache L2, porém tratando-se da cache L1, qualquer espaço é significativo, tornando a solução WT mais atraente.

Seria possível também adicionar um bit de validade aos blocos (útil em multiprocessadores e na inicialização da cache, marcando posições de memória “vagas”). No entanto esse bit seria utilizado somente no começo do sistema, gastando espaço adicional e ainda acrescentando ineficiência (busca constante de posições livres “inválidas’). Assim, a implementação do LRU utilizando FIFO (First In, First Out) resolve bem esse quesito, podendo dispensar a utilização deste bit.

A implementação de FIFO é outro aspecto a se considerar. A melhor solução em espaço é usar as posições da cache como uma fila circular, mantendo em memória a última posição de bloco alocado (de forma que a posição seguinte será sempre a do bloco mais antigo, facilitando a substituição). Essa estratégia usará espaço adicional da ordem O(log n), em que n é o número de conjuntos de blocos da cache em questão (65536 blocos -> 4096 conjuntos -> log(4096) = 12 bits para cache L2 e 1024 blocos -> 128 conjuntos -> log(128) = 7 bits para a cache L1).

No simulador existem variáveis adicionais, como “Política de gravação”, “Política de gravação na falha”, “tempo de acesso”, “tempo de tag”, “memória inferior” e “status de hit”, que apenas usam o espaço em memória no simulador, não se aplicando a este quesito de espaço despendido com o sistema de memória em si.

**2.3 Otimização sugerida**

1. Logo de cara, a solução de implementar LRU usando FIFO me deixou um pouco intrigado por existirem técnicas mais eficientes de se proceder, como o algoritmo do envelhecimento (usando contador) ou mesmo uma pilha dos os blocos da cache. O problema desta solução é o gasto adicional de espaço da cache em uma aplicação prática, mas otimizando bastante a estratégia de substituição de blocos.

2. Devo anotar aqui também a necessidade de criação de uma função de Hash, para dividir os blocos de cache em conjuntos de associatividade. Usando o operador resto com divisor igual ao número de blocos, notei um resultado muito insatisfatório, de modo que procurei uma solução ótima que distribuísse igualitariamente os blocos entre os conjuntos. Não sei se esse aspecto se encaixa como otimização, mas notei melhoria significativa dos resultados após esta prática.

3. Ao rodar o programa pela primeira vez, notei uma lerdeza descomunal na execução. Foram 6 minutos no total para processar todos os dados de entrada. Investigando o problema, notei que a leitura do arquivo de entrada não era o fator limitante (ordem de poucos segundos), assim notei que meu programa faz chamadas recursivas e que no início de das operações de leitura/escrita, procuro um dado endereço em um vetor, operação O(n), em que n é o tamanho da cache, da ordem de milhar. Ataquei este problema, procurando o endereço somente nas posições do conjunto associativo referente ao endereço (obtido graças ao Hash que implementei). A complexidade foi reduzida para O(m), em que m é a associatividade da cache, da ordem de unidades. Conclusão, meu simulador ficou 25 vezes mais rápido, sendo executado em poucos segundos.

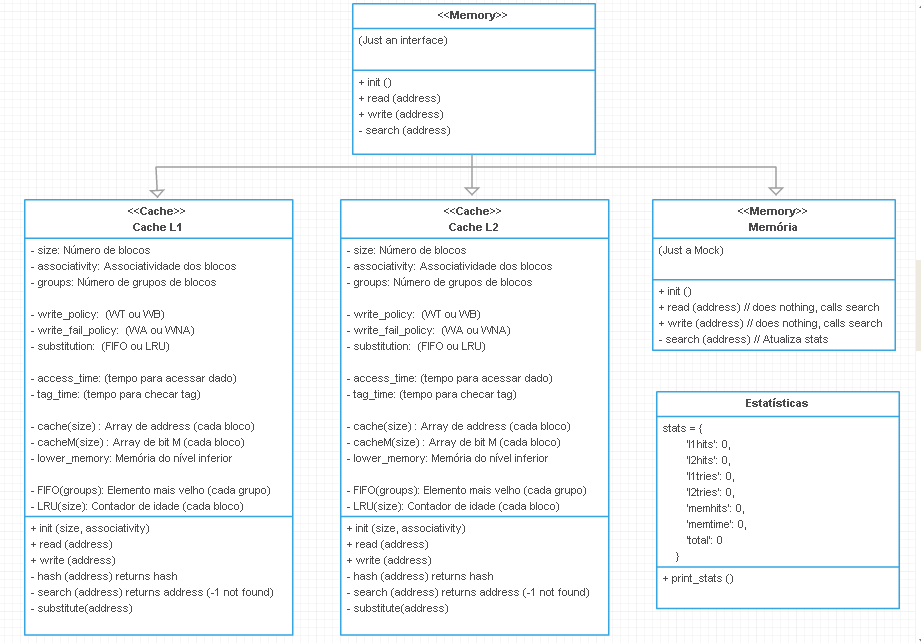
**3. IMPLEMENTAÇÃO**

**3.1 Implementação Classes**

Gostaria de nesta seção explicitar alguns detalhes sobre a arquitetura da implementação do simulador em si, que julguei serem relevantes.

Foram criadas três classes no simulador:

1. Cache: É a classe mais detalhada, com todas as informações e comportamentos do simulador de memória. Permite facilmente descrever L1 e L2 e é generalizável para um sistema com N níveis de cache.
2. Memória: É apenas um Mock que não realiza nada, apenas atualiza as estatísticas e garante o acerto de escrita/leitura como nível-piso do sistema.
3. Estatísticas: Possui as estatísticas do simulador e faz os cálculos necessários para exibir as informações do sistema.

****

**Diagrama UML com as classes do sistema de caches.**

**3.2 Implementação de Rotinas**

Também foi criada um trecho de código global para inicialização do sistema conforme as características propostas, além de uma rotina Main para desempenhar a interpretação do arquivo de entrada e chamar as funções desejadas para o sistema, além de comandar a escrita de estatísticas ao término do programa. Segue abaixo os trechos de código.

Gostaria de citar em especial o suo do vetor auxiliar de tempos, “times”. A cada operação solicitada, ele insere o tempo gasto para aquela tarefa nesta lista, de forma que tarefas sequenciais são seus elementos e os tempos das tarefas desempenhadas em paralelo são conservados, em vez de somados. Ao término de cada operação, é tomado o valor máximo desta lista, que é o tempo gasto com a operação (sempre o tempo do mais lento). Em caso de operações que envolvem tempos extras (vide tabela da seção anterior), o caso é devidamente tratado, da seguinte forma: o tempo de memória é devidamente incrementado e então este vetor é esvaziado, somando os tempos sequenciais em vez de considera-los paralelos.

u"""Escopo global para chamada da main."""

# Auxiliar times vector

times = []

# Creating Statistics

stats = Statistics()

# Creating Memory

mem = Memory()

mem.access\_time = 60

# Creating Cache L2

l2 = Cache(65536, 16)

l2.write\_policy = 'WB'

l2.write\_fail\_policy = 'WNA'

l2.substitution = 'FIFO'

l2.stathits = 'l2hits'

l2.stattries = 'l2tries'

l2.access\_time = 4

l2.tag\_time = 2

l2.lower\_level = mem

# Creating Cache L1

l1 = Cache(1024, 8)

l1.write\_policy = 'WT'

l1.write\_fail\_policy = 'WA'

l1.substitution = 'LRU'

l1.stathits = 'l1hits'

l1.stattries = 'l1tries'

l1.access\_time = 2

l1.tag\_time = 1

l1.lower\_level = l2

main()

**Trecho global de código com as inicializações.**

def main():

u"""Rotina main do Simulador de Memória."""

print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

print "\* \*"

print "\* Simulador de Sistema de Memória \*"

print "\* \*"

print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"

total = 0

# Para cada linha do arquivo de entrada:

with open('../gcc.trace') as infile:

for line in infile:

# Barra de progresso

if total % 50000 == 0:

print "#",

total += 1

# Interpreta endereço e operação

address = int(line[:8], 16)

operation = line[9]

# Realiza operação selecionada

del times[:]

if operation == 'R':

l1.read(address)

elif operation == 'W':

l1.write(address)

# Operações realizads em paralelo

stats.stats['memtime'] += max(times)

# Imprime estatísticas

stats.stats['total'] = total

stats.print\_stats()

**Rotina main com o loop principal do programa.**

**3.3 Implementação LRU**

Outro detalhe importante é a determinação do slot da cache que será substituído, seja através do algoritmo FIFO ou do algoritmo do envelhecimento LRU. A ideia é primeiro determinar o grupo de blocos onde ocorrerá a substituição, usando a função Hash, em seguida, verifica-se o tipo da substituição.

Para o caso da FIFO, basta pegar o elemento apontado por FIFO para o grupo calculado (sempre aponta para o elemento mais velho), em seguida incrementa esse ponteiro usando fila circular (volta ao início da fila quando chega ao final).

Já o LRU (sigla usada no sentido algoritmo do contador ou envelhecimento) percorre as posições do conjunto associativo, já incrementando seus contadores ao mesmo tempo que determina o valor máximo deste vetor (o elemento mais velho do grupo). Em seguida zera a idade do elemento mais velho, uma vez que ele será substituído.

def substitute(self, address):

u"""Substituição de conjunto de blocos na Cache."""

group = self.hash(address)

if self.substitution == 'FIFO':

slot = group \* self.associativity + self.FIFO[group]

self.FIFO[group] = (self.FIFO[group] + 1) % self.associativity

elif self.substitution == 'LRU':

start = group \* self.associativity

stop = (group + 1) \* self.associativity

slot = start

for i in range(start, stop):

if self.LRU[slot] < self.LRU[i]:

slot = i

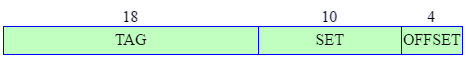
self.LRU[i] = self.LRU[i] + 1

self.LRU[slot] = 0

**Algoritmos LRU para substituição de blocos.**

**3.4 Implementação Hash**

A implementação padrão de Hash é a partir do endereço consultado realizar o mascaramento de seus bits, determinando uma SET referente ao conjunto de blocos daquele dado endereço[5].



def hash(self, address):

u"""Cálculo do hash do conjunto do bloco de Cache."""

hash = address % (self.block\_size \* self.size)

hash = hash / self.block\_size

hash = hash / self.associativity

self.hashes[hash] = self.hashes[hash] + 1

return hash

**Algoritmos de Hash com mascaramento de bits.**

No entanto, por apresentar baixa performance, adotei uma variante que pode ser desenvolvido usando o operador % da seguinte forma (não necessariamente fazendo mascaramento de bits, mas determinando uma boa regra – simples e rápida para a associação dos blocos da cache):

def hash(self, address):

u"""Cálculo do hash do conjunto do bloco de Cache."""

return address % self.groups

**Algoritmos de Hash usando operador resto %.**

Gostaria de mencionar também o algoritmo do Hash que inventei. Procurei realizar o cálculo mais aleatório possível sobre o número do endereço, de forma a minimizar o número de colisões nos grupos de blocos da cache. Utilizei o seguinte cálculo:

def hash(self, address):

u"""Cálculo do hash do conjunto do bloco de Cache."""

hash = i = 1

while address > 0:

hash = hash + (address % 10) \* i

address = address / 10

i = i + 1

return hash % self.groups

**Algoritmo para o cálculo do Hash.**

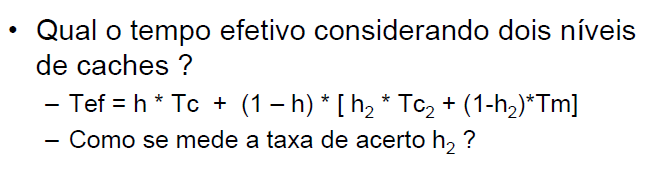
**4. RESULTADOS**

**4.1 Critérios para resultados & avaliação**

Como medida de desempenho do sistema, temos as estatísticas de acerto e de sucesso do acesso a cada uma das caches. Atribui um significado diferente a cada um destes: a taxa de acerto é a razão entre o número de hits (de uma dada cache/memória) e o número total de todas operações realizadas. A taxa de sucesso é a razão entre o número de hits (de uma dada cache/memória) e o número total de acessos (tentativas com sucesso ou com falha) àquela mesma dada cache/memória.

**Cálculo da taxa de sucesso do simulador de memória.**

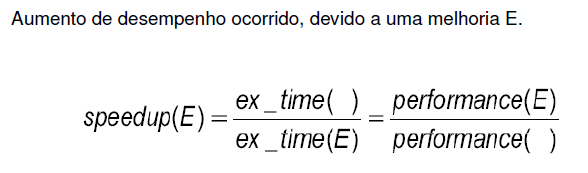
Temos também o tempo de memória, que indica o tempo total de execução do simulador com base no benchmark fornecido. É um número alto que isolado aparentemente não tem muito sentido. Mas pode ser muito útil para comparar com a melhoria sugerida e também para o cálculo do tempo efetivo do sistema, que abordo no próximo parágrafo.

Outro parâmetro para a medição de desempenho que também foi calculado é o tempo efetivo do sistema de memória (valor intermediário dos tempos de cache e memória), usando a seguinte fórmula para dois níveis de cache [2]: 

**Cálculo do tempo efetivo de um sistema de memória.**

Respondendo à pergunta do enunciado, a taxa de acerto h2 é dada pela razão entre o número total de hits na cache L2 e o número de acessos (requisições, sendo acertos e falhas) à cache L2. Ou seja, dado que L1 falhou, a taxa de sucesso na cache L2.

Como otimização do próprio simulador, temos o tempo gasto para simular o sistema de cache para o dado benchmark, que muitas vezes pode ser alto e inviabilizar uma comparação em tempo real. Assim, a sugestão citada na Seção 2 tem uma melhoria de desempenho que pode ser mensura pelo cálculo do SpeedUp do simulador:



**Cálculo do SpeedUp do simulador do sistema de memória.**

**4.2 Resultados para simulação original**

Respondendo à pergunta do enunciado, a taxa de acerto h2 é dada pela razão entre o número total de hits na cache L2 e o número de acessos (requisições, sendo acertos e falhas) à cache L2. Ou seja, dado que L1 falhou, a taxa de sucesso na cache L2.

**4.3 Resultados para simulação otimizada**

Respondendo à pergunta do enunciado, a taxa de acerto h2 é dada pela razão entre o número total de hits na cache L2 e o número de acessos (requisições, sendo acertos e falhas) à cache L2. Ou seja, dado que L1 falhou, a taxa de sucesso na cache L2.

**5. ANÁLISE**

Comparando os resultados obtidos para cada modelo de sistema de memória simulado, observamos que

**6. CONCLUSÃO**

Este projeto permitiu aprender na prática o funcionamento de um sistema de memória e as estratégias adotadas para otimizar sua performance. Particularmente achei bem interessante aplicar o simulador para um arquivo benchmark relativamente grande, podendo obter e visualizar resultados quantitativos da aplicação de uma nova política ou otimização proposta.

Sobre o grau de dificuldade do desenvolvimento, acredito estar bem acima da média para um projeto com tão pouco prazo, dado o nível de detalhamento do sistema de memória. Porém, a facilidade dos resultados cobrados (bem simplificadas) compensam o nível de complexidade do projeto em geral.

**7. REFERÊNCIAS BIBLIOGRÁFICAS**

[1] Wikipedia : <https://pt.wikipedia.org/wiki/Simula%C3%A7%C3%A3o>

[2] Slides de aula: <http://www.comp.ita.br/~pauloac/>

[3] Stackoverflow (sempre): <http://stackoverflow.com/>

[4] Memória Cache - UFPB: <http://producao.virtual.ufpb.br/books/edusantana/introducao-a-arquitetura-de-computadores-livro/livro/livro.chunked/ch05s07.html>

[5] Cache Addressing: <https://www.d.umn.edu/~gshute/arch/cache-addressing.xhtml>

**8. APÊNDICE - Código**