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.

Core 2 Duo T7200

O simulador foi implementado na linguagem de alto nível Python, uma vez que não é exigido alto desempenho para o simulador, 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 acesso ao Sist.Mem.	Situação		
	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 por subida de bloco (*ID)	Situação	
	Descrição	Tempo Adicional
(*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.

4. Como minha implementação do sistema de memórias foi arbitrária e generalizável para N caches, eu não poderia deixar de testar um sistema com 3 níveis de cache, colocando uma cache L3 maior e mais lenta entre a cache L2 e a memória principal, diminuindo o gap entre os tempos dos níveis de memória.

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.

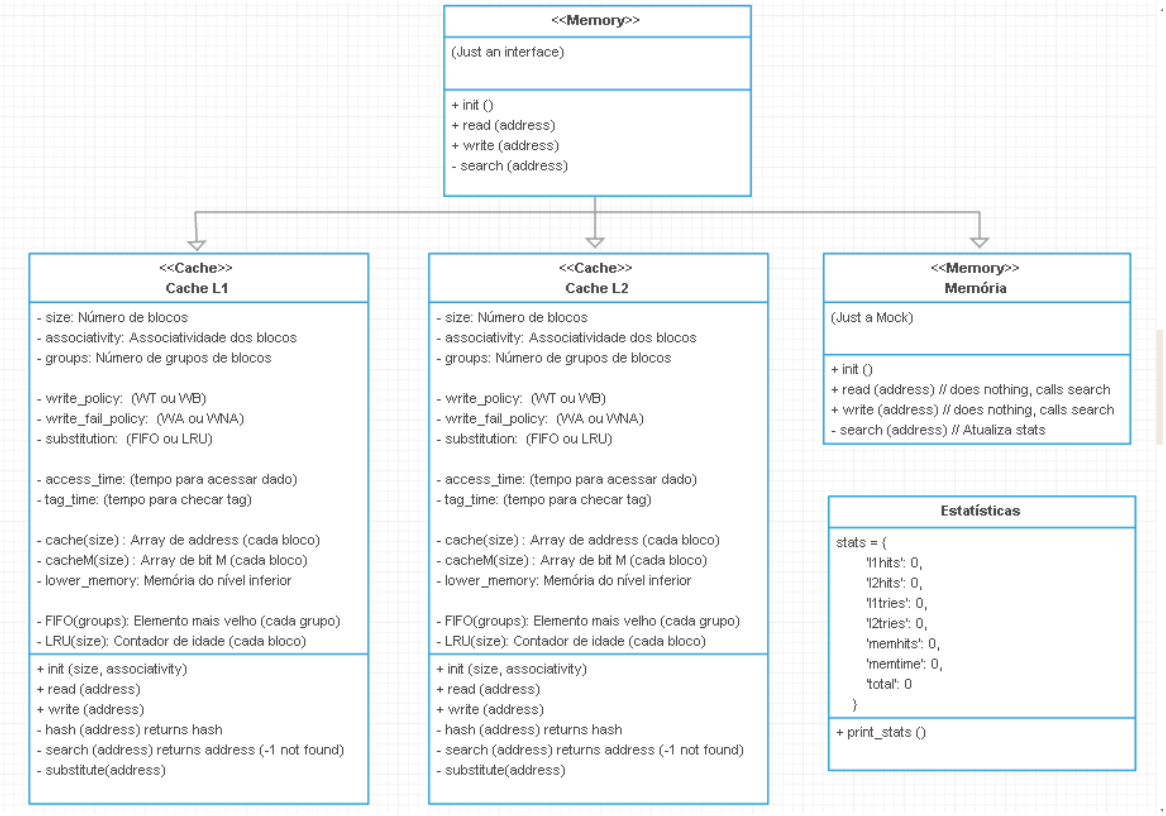


Diagrama UML com as classes do sistema de caches.

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, 512, 16)
l2.write_policy = 'WB'
l2.write_fail_policy = 'WNA'
l2.substitution = 'FIFO'
l2.hash_type = 'address'
l2.stathits = 'l2hits'
l2.stattries = 'l2tries'
l2.access_time = 4
l2.tag_time = 2
l2.lower_level = mem

# Creating Cache L1
l1 = Cache(1024, 512, 8)
l1.write_policy = 'WT'
l1.write_fail_policy = 'WA'
```

```
l1.substitution = 'FIFO'
l1.hash_type = 'address'
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).

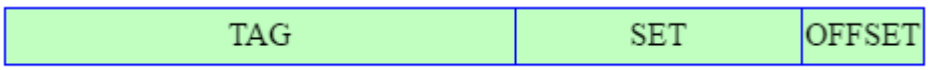
Já o LRU (sigla usada no sentido algoritmo do contador ou envelhecimento) percorre as posições do conjunto associativo, já incrementando seus contadores ao mesmo tempo que determina o valor máximo deste vetor (o elemento mais velho do grupo). Em seguida zera a idade do elemento mais velho, uma vez que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

$$hash(address) = \left(\sum_{i}^{digits(address)} digit_i * i \right) \%(\#groups)$$

```
    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.

$$Taxa_{acerto} = \frac{\#hits[Para\ dada\ cache\ ou\ memória]}{\#Total\ de\ todas\ as\ operações}$$

$$Taxa_{sucesso} = \frac{\#sucessos}{(\#sucessos + \#falhas)}\ [Para\ dada\ cache\ ou\ memória]$$

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]:

- Qual o tempo efetivo considerando dois níveis de caches ?
  - Tef = h * Tc + (1 – h) * [h_2 * Tc_2 + (1-h_2)*Tm]
  - Como se mede a taxa de acerto h_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:

Aumento de desempenho ocorrido, devido a uma melhoria E.

$$speedup(E) = \frac{ex\_time(\ )}{ex\_time(E)} = \frac{performance(E)}{performance(\ )}$$

Cálculo do SpeedUp do simulador do sistema de memória.

O cálculo das estatísticas segue conforme o código em Python abaixo, descrito na classe referente às estatísticas do simulador:

```
    def print_stats(self):
        u"""Exibe as estatísticas da execução do benchmark."""
        print ""
        print "Estatísticas: " + str(stats.stats)

        l1_hit_rate = 1.0 * self.stats['l1hits'] / self.stats['total']
        l2_hit_rate = 1.0 * self.stats['l2hits'] / self.stats['total']
        mem_hit_rate = 1.0 * self.stats['memhits'] / self.stats['total']
        print "L1  hit rate: " + str(l1_hit_rate)
        print "L2  hit rate: " + str(l2_hit_rate)
        print "Mem hit rate: " + str(mem_hit_rate)

        l1_success_rate = 1.0 * self.stats['l1hits'] / self.stats['l1tries']
        l2_success_rate = 1.0 * self.stats['l2hits'] / self.stats['l2tries']
        mem_success_rate = 1.0
        print "L1  success rate: " + str(l1_success_rate)
        print "L2  success rate: " + str(l2_success_rate)
        print "Mem success rate: " + str(mem_success_rate)

        l1time = (l1.tag_time + l1.access_time)
        l2time = (l2.tag_time + l2.access_time)
        effective_time = (
            l1_success_rate * l1time + (1.0 - l1_success_rate) * (
                l2_success_rate * l2time + (1.0 - l2_success_rate) * (
                    mem.access_time
                )))
        print "Effective Time: " + str(effective_time)
        print "Mem Total Time: " + str(self.stats['memtime'] / 1000.0)
```

Algoritmo para cálculo das estatísticas do Sistema.

4.2 Resultados para simulação otimizada (Hash)

Enfim, vamos ao resultado do programa para a configuração original sugerida. Queria dizer desde já que os resultados apresentam um desempenho bem ruim e conflitante com os valores comparados com a turma, o que me leva a crer que talvez esteja algo mal configurado no meu programa. Não consegui descobrir exatamente o problema, de modo que mostro a saída para a dada configuração:

```
u"""Escopo global."""

# Auxiliar times vector
times = []

# Creating Statistics
stats = Statistics()

# Creating Memory
mem = Memory()
mem.access_time = 60

main()
```

```
# Creating Cache L1
l1 = Cache(1024, 512, 8)
l1.write_policy = 'WT'
l1.write_fail_policy = 'WA'
l1.substitution = 'FIFO'
l1.hash_type = 'address'
l1.stathits = 'l1hits'
l1.stattries = 'l1tries'
l1.access_time = 2
l1.tag_time = 1
l1.lower_level = l2
```

```
# Creating Cache L2
l2 = Cache(65536, 512, 16)
l2.write_policy = 'WB'
l2.write_fail_policy = 'WNA'
l2.substitution = 'FIFO'
l2.hash_type = 'address'
l2.stathits = 'l2hits'
l2.stattries = 'l2tries'
l2.access_time = 4
l2.tag_time = 2
l2.lower_level = mem
```

```
**************************************
*                                    *
*  Simulador de Sistema de Memória   *
*                                    *
**************************************
# # # # # # # # # # # # # # # # # # # #
Estatísticas: {'l1hits': 504575, 'l2hits': 155637, 'l1tries': 1000000,
'memtime': 26864439, 'memhits': 409926, 'total': 1000000, 'l2tries': 554981}
L1  hit rate: 0.504575
L2  hit rate: 0.155637
Mem hit rate: 0.409926
L1  success rate: 0.504575
L2  success rate: 0.28043662756
Mem success rate: 1.0
Effective Time: 23.7367179247
Mem Total Time: 26864.439
[Finished in 12.1s]
```

Saída 1 – Sem melhorias.

Primeiramente, mudei o cálculo de hash, de address para usar o operador % como descrito na seção interior, obtendo um salto significativo de desempenho (mas ainda bem inferior ao esperado), para as seguintes configurações:

```
u"""Escopo global."""

# Auxiliar times vector
times = []

# Creating Statistics
stats = Statistics()

# Creating Memory
mem = Memory()
mem.access_time = 60

main()
```

```
# Creating Cache L1
l1 = Cache(1024, 512, 8)
l1.write_policy = 'WT'
l1.write_fail_policy = 'WA'
l1.substitution = 'FIFO'
l1.hash_type = 'address'
l1.stathits = 'l1hits'
l1.stattries = 'l1tries'
l1.access_time = 2
l1.tag_time = 1
l1.lower_level = l2
```

```
# Creating Cache L2
l2 = Cache(65536, 512, 16)
l2.write_policy = 'WB'
l2.write_fail_policy = 'WNA'
l2.substitution = 'FIFO'
l2.hash_type = 'address'
l2.stathits = 'l2hits'
l2.stattries = 'l2tries'
l2.access_time = 4
l2.tag_time = 2
l2.lower_level = mem
```

```
**************************************
*                                    *
*  Simulador de Sistema de Memória   *
*                                    *
**************************************
# # # # # # # # # # # # # # # # # # #
Estatísticas: {'l1hits': 529837, 'l2hits': 398564, 'l1tries': 1000000,
'memtime': 12096408, 'memhits': 138372, 'total': 1000000, 'l2tries': 532432}
L1  hit rate: 0.529837
L2  hit rate: 0.398564
Mem hit rate: 0.138372
L1  success rate: 0.529837
L2  success rate: 0.748572587673
Mem success rate: 1.0
Effective Time: 10.7939297889
Mem Total Time: 12096.408
[Finished in 8.7s]
```

Saída 2 – Otimização do hash usando operador %.

Percebendo o grande efeito da escolha de um hash apropriado, tentei então usar a operador que eu inventei para este cálculo, esperando uma melhoria ainda mais significativa para o sistema de memória. Eis o resultado, que agora pode-se dizer “respeitável”:

```
u"""Escopo global."""

# Auxiliar times vector
times = []

# Creating Statistics
stats = Statistics()

# Creating Memory
mem = Memory()
mem.access_time = 60

main()
```

```
# Creating Cache L1
l1 = Cache(1024, 512, 8)
l1.write_policy = 'WT'
l1.write_fail_policy = 'WA'
l1.substitution = 'FIFO'
l1.hash_type = 'address'
l1.stathits = 'l1hits'
l1.stattries = 'l1tries'
l1.access_time = 2
l1.tag_time = 1
l1.lower_level = l2
```

```
# Creating Cache L2
l2 = Cache(65536, 512, 16)
l2.write_policy = 'WB'
l2.write_fail_policy = 'WNA'
l2.substitution = 'FIFO'
l2.hash_type = 'address'
l2.stathits = 'l2hits'
l2.stattries = 'l2tries'
l2.access_time = 4
l2.tag_time = 2
l2.lower_level = mem
```

```
**************************************
*                                    *
*  Simulador de Sistema de Memória   *
*                                    *
**************************************
# # # # # # # # # # # # # # # # # # # #
Estatísticas: {'l1hits': 668328, 'l2hits': 341092, 'l1tries': 1000000,
'memtime': 7804017, 'memhits': 66293, 'total': 1000000, 'l2tries': 406705}
L1  hit rate: 0.668328
L2  hit rate: 0.341092
Mem hit rate: 0.066293
L1  success rate: 0.668328
L2  success rate: 0.838671764547
Mem success rate: 1.0
Effective Time: 6.8844511595
Mem Total Time: 7804.017
[Finished in 12.5s]
```

Saída 3 – Otimização do hash usando minha função inventada.

4.4 Resultados para simulação otimizada (tempo de execução)

Outro aspecto interessante que devo notar foi a otimização do tempo de execução do simulador. No início, estava realizando a busca de endereço em todas as posições da cache, levando a um procedimento demorado. Agora observo somente parte dela, a partir do grupo associativo indicado pelo Hash calculado.

Usando o algoritmo ineficiente de busca temos o seguinte resultado:

```
    def search(self, address):
        u"""Procura endereço na cache."""
        stats.stats[self.stattries] += 1
```

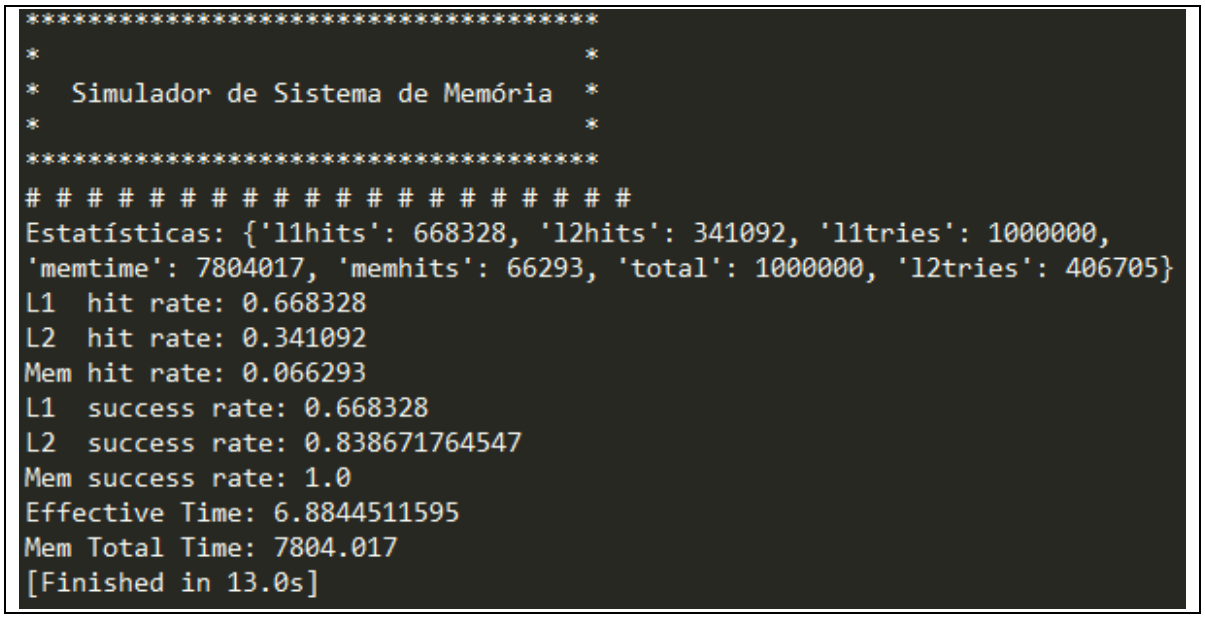
```
        if address in self.cache:
            return self.cache.index(address)
        return -1
```

```
**************************************
*                                    *
*  Simulador de Sistema de Memória   *
*                                    *
**************************************
# # # # # # # # # # # # # # # # # # # #
Estatísticas: {'l1hits': 668328, 'l2hits': 341092, 'l1tries': 1000000,
'memtime': 7804017, 'memhits': 66293, 'total': 1000000, 'l2tries': 406705}
L1  hit rate: 0.668328
L2  hit rate: 0.341092
Mem hit rate: 0.066293
L1  success rate: 0.668328
L2  success rate: 0.838671764547
Mem success rate: 1.0
Effective Time: 6.8844511595
Mem Total Time: 7804.017
[Finished in 481.9s]
```

Saída 1 – Sem otimização. Surpreendentes 480s.

Para o algoritmo eficiente, temos a seguinte melhoria:

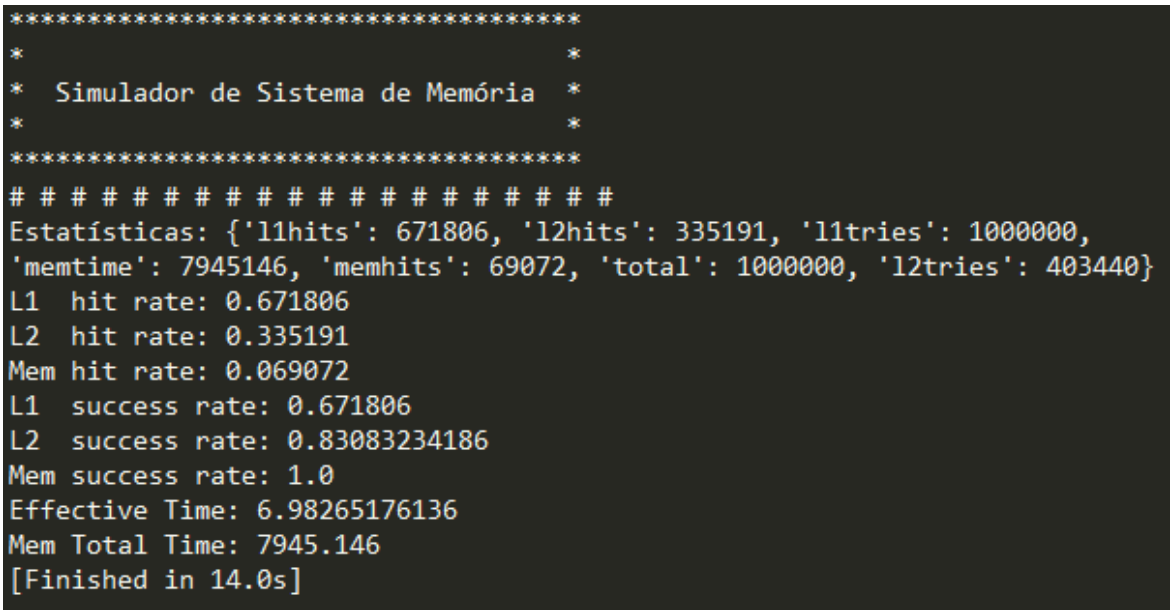
```
    def search(self, address):
        u"""Procura endereço na cache."""
        stats.stats[self.stattries] += 1

        # Setting the searching range
        group = self.hash(address)
        start = group * self.associativity
        stop = (group + 1) * self.associativity

        for i in range(start, stop):
            if address == self.cache[i]:
                return i
        return -1
```



Saída 2 – Resultado usando cache inventada. Meros 13s.

4.5 Resultados para simulação otimizada - LRU

Como proposta de melhoria da performance do sistema de memória, proponho o uso de LRU usando contador, em vez do algoritmo simplista FIFO. Eis os resultados usando esse novo LRU, usando a cache inventada:



Saída 1 – Resultado usando cache inventada e LRU contador.

4.6 Resultados para simulação otimizada – Cache L3

Pessoalmente, eu não poderia deixar de incluir essa comparação. Usando a fonte [2], observei no slide 15 do capítulo de memórias uma tela do software CPU-Z com algumas informações de Cache L3. Decidi adaptá-las um pouco, estimando as informações que faltavam:

Cache L3:

- Tamanho: 6 MB
- Tamanho do bloco: 64 bytes
- Política de gravação: WB
- Política de substituição: FIFO
- Associatividade de blocos: 32 vias
- Tempo de acesso: 10 clocks
- Tempo de tag: 5 clocks
- Write Not Allocate

OBS: Mudei a cache L2 para WA, agora que seu nível inferior não é mais a memória.

```
u"""Escopo global."""

# Auxiliar times vector
times = []

# Creating Statistics
stats = Statistics()

# Creating Memory
mem = Memory()
mem.access_time = 60

main()
```

```
# Creating Cache L1
l1 = Cache(1024, 512, 8)
l1.write_policy = 'WT'
l1.write_fail_policy = 'WA'
l1.substitution = 'FIFO'
l1.hash_type = 'address'
l1.stathits = 'l1hits'
l1.stattries = 'l1tries'
l1.access_time = 2
l1.tag_time = 1
l1.lower_level = l2
```

```
# Creating Cache L2
l2 = Cache(65536, 512, 16)
l2.write_policy = 'WB'
l2.write_fail_policy = 'WA'
l2.substitution = 'FIFO'
l2.hash_type = 'bigcomplex'
l2.stathits = 'l2hits'
l2.stattries = 'l2tries'
l2.access_time = 4
l2.tag_time = 2
l2.lower_level = l3
```

```
# Creating Cache L3
l3 = Cache(98304, 512, 32)
l3.write_policy = 'WB'
l3.write_fail_policy = 'WNA'
l3.substitution = 'FIFO'
l3.hash_type = 'bigcomplex'
l3.stathits = 'l3hits'
l3.stattries = 'l3tries'
l3.access_time = 10
l3.tag_time = 5
l3.lower_level = mem
```

```
**************************************
*                                    *
*  Simulador de Sistema de Memória   *
*                                    *
**************************************
# # # # # # # # # # # # # # # # # # # #
Estatísticas: {'memhits': 46363, 'total': 1000000, 'l1hits': 668328,
'l2hits': 355729, 'memtime': 6888189, 'l3tries': 56727, 'l1tries': 1000000,
'l2tries': 406705, 'l3hits': 12810}
L1  hit rate: 0.668328
L2  hit rate: 0.355729
L3  hit rate: 0.01281
Mem hit rate: 0.046363
L1  success rate: 0.668328
L2  success rate: 0.87466099507
L3  success rate: 0.225818393358
Mem success rate: 1.0
Effective Time: 5.81743188118
Mem Total Time: 6888.189
[Finished in 13.1s]
```

Saída – Otimização com Cache L3.

5. ANÁLISE

5.1 Sobre o uso do Hash

Primeiramente, comparo os resultados obtidos mudando a função de Hash (que determina a qual conjunto associativo cada bloco da cache pertence). Observo uma forte variação dos resultados perante esta escolha, de modo que acredito que deve haver algo de incoerente com o cálculo usando mascaramento de bits do endereço, devido ao péssimo desempenho obtido.

Fiquei surpreso com a melhoria usando o operador %. O tempo de memória foi reduzido pela metade e o número de success rate de L2 praticamente dobrou, indicando que sua utilização foi melhorada neste processo. O cálculo deste Hash é bem simplista, tanto que a execução fica 4 segundos mais rápida que os demais, neste caso.

Meu hash aleatório, então, levou o sistema de memória ao ápice da utilização dos grupos de blocos, quase cortando novamente pela metade o tempo de memória (o tempo efetivo do sistema fica quase igual ao tempo de L2), culminando as taxas de sucesso das caches em 66% e 84%, respectivamente. Esses resultados ainda são inferiores aos obtidos

por minha turma, mas acredito estar bem próximo do Hash idealizado pela atividade, que maximizaria estes números.

Mas o que eu acho realmente essencial de detalhar é o hit rate da memória. Essa porcentagem otimizada de 6,6% indica o quanto a memória principal foi realmente empregada dentre o número total de operações. Observa-se uma diminuição brutal desse número em relação ao Hash original (41%), o que indica que os níveis de cache estão de fato desempenhando o papel que deveriam realizar: evitar o uso de memória principal lenta.

5.2 Sobre a performance do simulador

Talvez não seja tão importante para o conteúdo da matéria, mas foi muito relevante para o projeto, uma vez que a execução extremamente lenta antes da otimização poderia inviabilizar a execução do projeto e tornar o feedback do simulador muito lento.

A escolha de uma política de posicionamento de blocos associativa de conjunto permitiu assim uma busca mais eficiente da cache em nível de software, com um speedUp de (480)/(13) = 37, podendo chegar até (480)/(8.7) = 55 (no caso de hash simples), além de otimizar o sistema de posicionamento como intermediário do completamente associativo (mais complexa e extremamente lenta – percorrer cache toda na busca de software) e mapeamento direto (mais simples, mas com possível ineficiência (dois blocos disputando posição)).

5.3 Sobre o LRU

Considero a utilização do LRU usando contador a melhoria sugerida para aprimorar o sistema de cache. Intuitivamente, o algoritmo me parece mais inteligente que o uso de uma FIFO simplista que pode vir a descartar um dado muito utilizado na cache, sem considerar sua frequência de uso.

No entanto, os resultados mostraram uma melhoria bem pobre ao se adotar esse algoritmo, quase equivalente ao FIFO. O espaço adicional gasto para armazenar as estruturas dos contadores (espaço físico na própria cache) realmente não vale a pena, tornando essa proposta mais complexa, demorada e ineficiente.

5.4 Sobre a cache L3

Dado o alto acerto da cache L2 para o exemplo otimizado com cache inventada, acho que o acréscimo da cache L3 apenas traria um pequeno benefício para o sistema em geral. As operações iniciais do sistema resultam em falhas nas três caches de toda forma, sendo imprescindível o uso de memória no inicial. Após essa situação, acredito ficar quase 100% da informação retida cache L3, pois há 1 milhão de endereços lidos para 100 mil blocos na cache L3. Dada o uso constante de mesmos endereços, esse espaço é mais que suficiente para o dado benchmark.

Minha tentativa de obter um bom Hash para L3 foi outro fator significante para seu desempenho. Para este caso não busquei exaustivamente a solução perfeita como fiz para as caches L1 e L2, de modo que o resultado teve um baixo success rate. O tempo efetivo de memória fica entre os tempos de L1 e L2, achei isso muito interessante, pois mostra que o tempo médio das operações é bem reduzido pelo sistema.

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 otimização proposta. Achei divertido brincar com os valores esperando melhorias de resultado, a criação da cache L3 me mostrou como o número de possibilidades é vasto, seria legal aplicar uma AI para otimizar estes parâmetros, talvez usando algoritmo genético ou algo do tipo.

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.

Discuti bastante com meus colegas resultados e metodologias do simulador, o que me deixou bastante frustrado, por não conseguir alcançar os mesmos resultados que eles. Não consegui identificar eventuais erros no meu código, por mais que eu me esforçasse. Mas acredito talvez estar no cálculo do Hash usando mascaramento de bits, pois meu resultado ficou horrível, sendo que funcionou perfeitamente para meus colegas (talvez seja a solução ideal para os hashes).

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

Vou incluir em anexo o código fonte do projeto. A variante com cache L3 será incluída em anexo, mas não aqui para não ocupar demasiado espaço.

```
# !/usr/bin/env python
# -*- coding: utf-8 -*-
u"""Simulador de Sistema de Memória."""
# Professor: Paulo André (PA)
# Disciplina: CES-25
# Autor: Felipe Tuyama


class Cache (object):
    u"""Classe Cache."""

    def __init__(self, size, block_size, associativity):
        u"""Inicialização da Cache."""
        self.size = size
        self.block_size = block_size
        self.associativity = associativity
        self.groups = size / associativity
        self.cache = [-1] * size
        self.cacheM = [False] * size
        self.FIFO = [0] * self.groups
        self.LRU = [0] * size

    def hash(self, address):
        u"""Cálculo do hash do conjunto do bloco de Cache."""
        if self.hash_type == 'simple':
            return address % self.groups
```

```
        elif self.hash_type == 'address':
            hash = address % (self.block_size * self.size)
            hash = hash / self.block_size
            hash = hash / self.associativity
            return hash
        elif self.hash_type == 'complex':
            hash = i = 1
            while address > 0:
                hash = hash + (address % 10) * i + i
                address = address / 10
                i = i + 1
            return hash % self.groups
        elif self.hash_type == 'bigcomplex':
            hash = i = 1
            while address > 0:
                hash = (hash + (address % 10) * i) * i
                address = address / 10
                i = i + 1
            return hash % self.groups

    def search(self, address):
        u"""Procura endereço na cache."""
        stats.stats[self.stattries] += 1

        # Setting the searching range
        group = self.hash(address)
        start = group * self.associativity
        stop = (group + 1) * self.associativity

        for i in range(start, stop):
            if address == self.cache[i]:
                return i
        return -1

    def substitute(self, address, write):
        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

        # Verfica se o bloco está sujo
        if self.cacheM[slot] and self.write_policy == 'WB':
            # Penalidade: Tempo extra para gravação
            stats.stats['memtime'] += max(times)
            del times[:]
            # O Bloco deve ser gravado no nível inferior
            self.lower_level.write(address)

        # Substituição do bloco sem traumas
        self.cache[slot] = address
        self.cacheM[slot] = write

    def read(self, address):
        u"""Operação de leitura na Cache."""
        # Busca endereço na Cache
        index = self.search(address)
        if index > 0:
            # Realização da leitura na Cache
            stats.stats[self.stathits] += 1
            times.append(self.tag_time + self.access_time)
        else:
            # Continua a busca no nível inferior
            times.append(self.tag_time)
            self.lower_level.read(address)
            # Traz o bloco para a Cache
            self.substitute(address, False)

    def write(self, address):
        u"""Operação de escrita na Cache."""
        # Busca endereço na Cache
        index = self.search(address)
        if index > 0:
            stats.stats[self.stathits] += 1
            # Política de Gravação Write Through
            if self.write_policy == 'WT':
                # Realização da escrita na Cache
                times.append(self.tag_time + self.access_time)
                self.cacheM[index] = True
                # Realiza escrita no nível inferior também
                self.lower_level.write(address)
            # Política de Gravação Write Back
```

```
            elif self.write_policy == 'WB':
                # Realização da escrita na Cache
                times.append(self.tag_time + self.access_time)
                self.cacheM[index] = True
        else:
            # Política de Gravação Write Allocate
            if self.write_fail_policy == 'WA':
                # Grava bloco no nível inferior
                times.append(self.tag_time)
                self.lower_level.write(address)
                # Traz o bloco para a Cache
                self.substitute(address, True)
            # Política de Gravação Write Not Allocate
            elif self.write_fail_policy == 'WNA':
                # Não traz o bloco do nível inferior
                times.append(self.tag_time)
                self.lower_level.write(address)


class Memory (object):
    u"""Memória RAM principal."""

    def __init__(self):
        u"""Inicialização da memória."""

    def search(self, address):
        u"""Procura endereço na memória."""
        times.append(self.access_time)
        stats.stats['memhits'] += 1

    def read(self, address):
        u"""Operação de leitura na memória."""
        # Busca endereço na memória
        self.search(address)

    def write(self, address):
        u"""Operação de escrita na memória."""
        # Busca endereço na memória
        self.search(address)


class Statistics (object):
    u"""Estatísticas do programa para a sua execução."""

    stats = {
        'l1hits': 0,
```

```
            'l2hits': 0,
            'l1tries': 0,
            'l2tries': 0,
            'memhits': 0,
            'memtime': 0,
            'total': 0
        }

        def print_stats(self):
            u"""Exibe as estatísticas da execução do benchmark."""
            print ""
            print "Estatísticas: " + str(stats.stats)

            l1_hit_rate = 1.0 * self.stats['l1hits'] /
self.stats['total']
            l2_hit_rate = 1.0 * self.stats['l2hits'] /
self.stats['total']
            mem_hit_rate = 1.0 * self.stats['memhits'] /
self.stats['total']
            print "L1  hit rate: " + str(l1_hit_rate)
            print "L2  hit rate: " + str(l2_hit_rate)
            print "Mem hit rate: " + str(mem_hit_rate)

            l1_success_rate = 1.0 * self.stats['l1hits'] /
self.stats['l1tries']
            l2_success_rate = 1.0 * self.stats['l2hits'] /
self.stats['l2tries']
            mem_success_rate = 1.0
            print "L1  success rate: " + str(l1_success_rate)
            print "L2  success rate: " + str(l2_success_rate)
            print "Mem success rate: " + str(mem_success_rate)

            l1time = (l1.tag_time + l1.access_time)
            l2time = (l2.tag_time + l2.access_time)
            effective_time = (
                l1_success_rate * l1time + (1.0 - l1_success_rate) * (
                    l2_success_rate * l2time + (1.0 - l2_success_rate)
* (
                        mem.access_time)))
            print "Effective Time: " + str(effective_time)
            print "Mem Total Time: " + str(self.stats['memtime'] /
1000.0)


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()

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, 512, 16)
l2.write_policy = 'WB'
l2.write_fail_policy = 'WNA'
l2.substitution = 'FIFO'
l2.hash_type = 'bigcomplex'
l2.stathits = 'l2hits'
l2.stattries = 'l2tries'
l2.access_time = 4
l2.tag_time = 2
l2.lower_level = mem

# Creating Cache L1
l1 = Cache(1024, 512, 8)
l1.write_policy = 'WT'
l1.write_fail_policy = 'WA'
l1.substitution = 'FIFO'
l1.hash_type = 'complex'
l1.stathits = 'l1hits'
l1.stattries = 'l1tries'
l1.access_time = 2
l1.tag_time = 1
l1.lower_level = l2

main()
```