Universidad de Costa Rica

IE-0521

Estructuras de Computadoras Digitales II

Profesor: Katherine Núñez Solano

Tarea Programada 2

Estudiante:
Andrés Alvarado

Carné:
B30313

24-11-2017

1. Introducción

Es necesario tener una coherencia de cache para múltiples procesadores, ya que si no la hay pueden haber lecturas de datos antiguos por lo que la información estaría errónea.

Hay dos maneras para esta coherencia. Se puede hacer mediante snooping, o coherencia basada en directorios. Existen varios protocolos para la coherencia basada en directorios, como MSI, MESI, MOSI, MOESI, etc. Para esta tarea se quería simular una coherencia en base de directorios con un protocolo tipo MESI. Este protocolo se basa en que cada dato ingresado en la cache puede tener 4 estados, Modificado (M), Exclusivo (E), Compartido (S) o Invalido (I).

1.1. Explicación del Protocolo

Al inicio si hay datos en el cache del procesador estos tendrían un estado de invalido ya que estos datos no se pueden utilizar. Al acceder un nuevo dato este tendrá un estado de Exclusivo, ya que no hay otro procesador con el dato leído. Este lo puede modificar las veces que el quiera y seguirá en estado Exclusivo. Cuando otro procesador hace una lectura de este dato, el procesador 1, pasa de Exclusivo a Compartido, ya que otro procesador lo esta utilizando, y el procesador 2 toma este valor directamente del procesador 1, sin tener que hacer lectura a memoria, ya que el Procesador 1 estaba en Exclusivo. Por lo que ambos procesadores tendrán el mismo valor y ambos en estado Compartido. Si el Procesador 2 hace un cambio a este valor, este dato pasa de Compartido a Modificado, y en el procesador 1 pasa de Compartido a Invalido, ya que el dato que este tenga ya no es verdadero.

2. Programa

Para hacer esta simulación se utilizó un programa en C++ que recibe un archivo texto con una dirección de memoria y un carácter L o S indicando si esta leyendo o salvando.

Este programa depende de 3 archivos, el main.cc, cache.cc y cache.h. El main.cc es el programa principal que hace la lectura del archivo de texto, lo interpreta y se lo manda al cache.cc para que haga el protocolo MESI con todos los datos que recibe.

Las funciones que valen la pena mencionar son las siguientes:

- PrRd : Se realiza una solicitud de lectura a un bloque
- PrWr : Se realiza una solicitud de escritura a un bloque
- BusRd : Implica que se realizó una lectura fallida en otro CPU. Se envia una señal para revisar los demás caches. En caso que otro cache posea el mismo bloque, pasan a estado S, en caso contrario estado E.
- BusRdx : Implica que se realizó una lectura en otro CPU pero este no poseía el bloque anteriormente.
- BusUpgr : Se da cuando un CPU realiza una escritura, se deben revisar si los demás caches poseen el mismo bloque. En ese caso se invalidan.

1. Condición I

a) Si se realiza una lectura (PrRd):

- Se activa la señal BusRd. Se revisan los demás caches, si poseen el mismo bloque, el estado cambia a S, de lo contrario cambia a E
- Se trae el dato de la memoria principal o de otro caché.

b) Si se realiza una Escritura (PrWr)
Se activa la señal BusRdX. * Estado cambia a "Mz se invalidan las posiciones de los demás caches que posean este bloque.

2. Condición S

a) Si se realiza una lectura (PrRd):
PrRd es un Hit (no se activa ningún bus). Simplemente se lee el dato

b) Si se realiza una Escritura (PrWr):
PrWr cambia de estado a "M", activa el BusUpgr e invalida las demás posiciones de cache.

3. Condición E
No es necesario activar ningún Bus.

a) Si se realiza una Lectura (PrRd):
En lectura (PrRd) el estado se mantiene igual (E).

b) Si se realiza una Escritura (PrWr):
En escritura se cambia el estado a "M".

4. Condición M
No se activa ningún Bus.
Ni lectura ni escritura cambian el estado (M).

Main.cc:

```
#include <stdlib.h>
#include <assert.h>
#include <fstream>
#include <sstream>
#include <string>
#include <istream>
#include <vector>
#include <iomanip>
#include "cache.h"
using namespace std;

void MESI(vector<Cache *> cacheArray, int proc_num, ulong address, char op, int
    num_proc)
{
  //For INVALID State
  if (cacheArray[proc_num]->findLine(address) == NULL)
  { //PrRd --- For Read(C) : INVALID ---> SHARED | For Read(!C) : INVALID --->
   EXCLUSIVE | FOR Write : INVALID ---> MODIFIED
    bool C = false;
    for (int i = 0; i < num_proc; i++)
    {
```

```
20        if (i != proc_num && cacheArray[i]->findLine(address) != NULL)
21        {
22          C = true;
23        }
24      }
25      if (op == 'L')
26      {
27        if (C == true)
28        {
29          cacheArray[proc_num]->Access(address, op);
30          (cacheArray[proc_num]->findLine(address))->setFlags(SHARED);
31          //Cache to Cache Transfer when Line exists in another processor and State
    INVALID-->SHARED
32          cacheArray[proc_num]->num_of_cache_to_cache_transfer++;
33          //Generate BusRD to turn modified to shared
34          for (int i = 0; i < num_proc; i++)
35          {
36            if (i != proc_num && cacheArray[i]->findLine(address) != NULL)
37            {
38              if ((cacheArray[i]->findLine(address))->getFlags() == MODIFIED)
39              {
40                cacheArray[i]->writeBack(address);
41                //cacheArray[proc_num]->num_of_chache_to_cache_transfer++;
42              }
43              else if ((cacheArray[i]->findLine(address))->getFlags() == EXCLUSIVE)
44              {
45                cacheArray[i]->num_of_interventions++;
46                //cacheArray[proc_num]->num_of_chache_to_cache_transfer++;
47              }
48              (cacheArray[i]->findLine(address))->setFlags(SHARED);
49            }
50          }
51        }
52        else
53        {
54          cacheArray[proc_num]->Access(address, op);
55          (cacheArray[proc_num]->findLine(address))->setFlags(EXCLUSIVE);
56        }
57        return;
58      }
59      else
60      { //Invalid ---> Modified
61        cacheArray[proc_num]->Access(address, op);
62        (cacheArray[proc_num]->findLine(address))->setFlags(MODIFIED);
63        //Generates BusRDx to invalidate other caches
64        for (int i = 0; i < num_proc; i++)
65        {
66          if (i != proc_num && cacheArray[i]->findLine(address) != NULL)
67          {
68            if ((cacheArray[i]->findLine(address))->getFlags() == SHARED)
69            {
70              cacheArray[i]->num_of_invalidations++;
71              cacheArray[proc_num]->num_of_cache_to_cache_transfer++;
72            }
73            else if ((cacheArray[i]->findLine(address))->getFlags() == MODIFIED)
74            {
```

```
75              cacheArray[i]->writeBack(address);
76            }
77            (cacheArray[i]->findLine(address))->invalidate();
78          }
79        }
80        return;
81      }
82    }
83    //For Exclusive State
84    if ((cacheArray[proc_num]->findLine(address))->getFlags() == EXCLUSIVE)
85    { //PrRd --- For Read : EXCLUSIVE ---> EXCLUSIVE | FOR Write : EXCLUSIVE --->
       MODIFIED
86      if (op == 'L')
87      {
88        cacheArray[proc_num]->Access(address, op);
89        (cacheArray[proc_num]->findLine(address))->setFlags(EXCLUSIVE);
90        return;
91      }
92      else
93      { //EXCLUSIVE ---> MODIFIED
94        cacheArray[proc_num]->Access(address, op);
95        (cacheArray[proc_num]->findLine(address))->setFlags(MODIFIED);
96        return;
97      }
98    }
99    //For SHARED State
100   if ((cacheArray[proc_num]->findLine(address))->getFlags() == SHARED)
101   {
102     /*READ*/
103     if (op == 'L')
104     {
105       cacheArray[proc_num]->Access(address, op);
106       return;
107     }
108     /*WRITE*/
109     else
110     { //Shared  ----> Modificado
111       cacheArray[proc_num]->Access(address, op);
112       (cacheArray[proc_num]->findLine(address))->setFlags(MODIFIED);
113       //Generates BusUPGR to invalidate other caches
114       for (int i = 0; i < num_proc; i++)
115       {
116         if (i != proc_num && cacheArray[i]->findLine(address) != NULL)
117         {
118           if ((cacheArray[i]->findLine(address))->getFlags() == SHARED)
119           {
120             cacheArray[i]->num_of_invalidations++;
121           }
122           (cacheArray[i]->findLine(address))->invalidate();
123         }
124       }
125       return;
126     }
127   }
128   //Modificado
129   if ((cacheArray[proc_num]->findLine(address))->getFlags() == MODIFIED)
```

```
130   {
131     if (op == 'L')
132     {
133       cacheArray[proc_num]->Access(address, op);
134       (cacheArray[proc_num]->findLine(address))->setFlags(MODIFIED);
135       return;
136     }
137     else
138     {
139       cacheArray[proc_num]->Access(address, op);
140       (cacheArray[proc_num]->findLine(address))->setFlags(MODIFIED);
141       return;
142     }
143   }
144 
145   return;
146 }
147 
148 int main(int argc, char *argv[])
149 {
150 
151   ifstream fin;
152   FILE *pFile;
153 
154   int pro;
155   uchar op;
156   uint addr;
157 
158   if (argv[1] == NULL)
159   {
160     printf("input format: ");
161     printf("./smp_cache  <trace_file> \n");
162     exit(0);
163   }
164 
165   int cache_size = 16000; //16Kb
166   int cache_assoc = 16;
167   int blk_size = 16;
168   int num_processors = 4;
169   char *fname = (char *)malloc(20); //Archivo de lectura
170   fname = argv[1];
171   vector<Cache *> cacheArray;
172 
173   for (int i = 0; i < num_processors; i++)
174   {
175     Cache *c = new Cache(cache_size, cache_assoc, blk_size);
176     cacheArray.push_back(c);
177   }
178 
179   pFile = fopen(fname, "r");
180 
181   if (pFile == 0)
182   {
183     printf("Error en la lectura, no es un formato aceptado\n");
184     exit(0);
185   }
```

```
186
187   //print las estadisticas //
188   int cont = 0;
189   while (!feof(pFile))
190   {
191     pro = cont %4;
192     cont = cont + 1;
193     fscanf(pFile, "%x %c \n", &addr, &op);
194     MESI(cacheArray, pro, addr, op, num_processors);
195   }
196
197   fclose(pFile);
198
199   for (int i = 0; i < num_processors; i++)
200   {
201     cout << "============ Simulation results (Cache " << i << ") ============" <<
      endl;
202     cout << "01. number of reads:                " << cacheArray[i]->getReads() <<
      endl;
203     cout << "02. number of read misses:          " << cacheArray[i]->getRM() << endl;
204     cout << "03. number of writes:               " << cacheArray[i]->getWrites() <<
      endl;
205     cout << "04. number of write misses:         " << cacheArray[i]->getWM() << endl;
206     cout << "05. total miss rate:                " << fixed << setprecision(2) << (
     cacheArray[i]->getWM() + cacheArray[i]->getRM()) * 100.0 / (cacheArray[i]->
     getReads() + cacheArray[i]->getWrites()) << '%' << endl;
207   }
208   return 0;
209 }
```

Cache.cc:

```
1 #include <stdlib.h>
2 #include <assert.h>
3 #include "cache.h"
4 using namespace std;
5
6 Cache::Cache(int s, int a, int b)
7 {
8   ulong i, j;
9   reads = readMisses = writes = 0;
10  writeMisses = writeBacks = currentCycle = 0;
11
12  size = (ulong)(s);
13  lineSize = (ulong)(b);
14  assoc = (ulong)(a);
15  sets = (ulong)((s / b) / a);
16  numLines = (ulong)(s / b);
17  log2Sets = (ulong)(log2(sets));
18  log2Blk = (ulong)(log2(b));
19
20  num_of_cache_to_cache_transfer = 0;
21  num_of_mem_trans = 0;
22  num_of_interventions = 0;
23  num_of_invalidations = 0;
24  num_of_flushes = 0;
25
```

```
26    tagMask = 0;
27    for (i = 0; i < log2Sets; i++)
28    {
29      tagMask <<= 1;
30      tagMask |= 1;
31    }
32
33    cache = new cacheLine *[sets];
34    for (i = 0; i < sets; i++)
35    {
36      cache[i] = new cacheLine[assoc];
37      for (j = 0; j < assoc; j++)
38      {
39        cache[i][j].invalidate();
40      }
41    }
42  }
43
44  void Cache::Access(ulong addr, uchar op)
45  {
46    currentCycle++;
47
48    if (op == 'S')
49      writes++;
50    else
51      reads++;
52
53    cacheLine *line = findLine(addr);
54    if (line == NULL) //miss
55    {
56      if (op == 'S')
57        writeMisses++;
58      else
59        readMisses++;
60
61      cacheLine *newline = fillLine(addr);
62      if (op == 'S')
63        newline->setFlags(MODIFIED); //Pasa al estado modificado despues de la
       escritura
64    }
65    else
66    {
67      //Como hubo un hit se modifica a Modificado
68      updateLRU(line);
69      if (op == 'S')
70        line->setFlags(SHARED);
71    }
72  }
73
74  cacheLine *Cache::findLine(ulong addr)
75  {
76    ulong i, j, tag, pos;
77
78    pos = assoc;
79    tag = calcTag(addr);
80    i = calcIndex(addr);
```

```
81
82    for (j = 0; j < assoc; j++)
83      if (cache[i][j].isValid())
84        if (cache[i][j].getTag() == tag)
85        {
86          pos = j;
87          break;
88        }
89    if (pos == assoc)
90      return NULL;
91    else
92      return &(cache[i][pos]);
93  }
94
95  void Cache::updateLRU(cacheLine *line)
96  {
97    line->setSeq(currentCycle);
98  }
99
100 cacheLine *Cache::getLRU(ulong addr)
101 {
102   ulong i, j, victim, min;
103
104   victim = assoc;
105   min = currentCycle;
106   i = calcIndex(addr);
107
108   for (j = 0; j < assoc; j++)
109   {
110     if (cache[i][j].isValid() == 0)
111       return &(cache[i][j]);
112   }
113   for (j = 0; j < assoc; j++)
114   {
115     if (cache[i][j].getSeq() <= min)
116     {
117       victim = j;
118       min = cache[i][j].getSeq();
119     }
120   }
121   assert(victim != assoc);
122
123   return &(cache[i][victim]);
124 }
125
126 cacheLine *Cache::findLineToReplace(ulong addr)
127 {
128   cacheLine *victim = getLRU(addr);
129   updateLRU(victim);
130
131   return (victim);
132 }
133
134 // Agrega nueva linea
135 cacheLine *Cache::fillLine(ulong addr)
136 {
```

```
137   ulong tag;
138
139   cacheLine *victim = findLineToReplace(addr);
140   assert(victim != 0);
141   if (victim->getFlags() == MODIFIED || victim->getFlags() == SHARED_MODIFIED)
142     writeBack(addr);
143
144   tag = calcTag(addr);
145   victim->setTag(tag);
146   victim->setFlags(MODIFIED);
147
148   return victim;
149 }
150
151 void Cache::printStats()
152 {
153   //imprime las estadisticas obtenidas
154 }
```

Cache.h:

```
1  #ifndef CACHE_H
2  #define CACHE_H
3
4  #include <cmath>
5  #include <iostream>
6
7  typedef unsigned long ulong;
8  typedef unsigned char uchar;
9  typedef unsigned int uint;
10
11 enum
12 {
13   INVALID = 0,
14   VALID,
15   DIRTY,
16   SHARED,
17   MODIFIED,
18   EXCLUSIVE,
19   SHARED_CLEAN,
20   SHARED_MODIFIED,
21   EMPTY
22 };
23
24 class cacheLine
25 {
26 protected:
27   ulong tag;
28   ulong Flags; // 0:INVALID, 1:SHARED, 2:MODIFIED
29   ulong seq;
30
31 public:
32   cacheLine()
33   {
34     tag = 0;
35     Flags = 0;
36   }
```

```
37   ulong getTag() { return tag; }
38   ulong getFlags() { return Flags; }
39   ulong getSeq() { return seq; }
40   void setSeq(ulong Seq) { seq = Seq; }
41   void setFlags(ulong flags) { Flags = flags; }
42   void setTag(ulong a) { tag = a; }
43   void invalidate()
44   {
45     tag = 0;
46     Flags = INVALID;
47   }
48   bool isValid() { return ((Flags) != INVALID); }
49 };
50
51 class Cache
52 {
53 protected:
54   ulong size, lineSize, assoc, sets, log2Sets, log2Blk, tagMask, numLines;
55   ulong reads, readMisses, writes, writeMisses, writeBacks;
56
57   cacheLine **cache;
58   ulong calcTag(ulong addr) { return (addr >> (log2Blk)); }
59   ulong calcIndex(ulong addr) { return ((addr >> log2Blk) & tagMask); }
60   ulong calcAddr4Tag(ulong tag) { return (tag << (log2Blk)); }
61
62 public:
63   ulong currentCycle;
64
65   ulong num_of_cache_to_cache_transfer;
66   ulong num_of_flushes;
67   ulong num_of_interventions;
68   ulong num_of_invalidations;
69   ulong num_of_mem_trans;
70
71   Cache(int, int, int);
72   ~Cache() { delete cache; }
73
74   cacheLine *findLineToReplace(ulong addr);
75   cacheLine *fillLine(ulong addr);
76   cacheLine *findLine(ulong addr);
77   cacheLine *getLRU(ulong);
78
79   ulong getRM() { return readMisses; }
80   ulong getWM() { return writeMisses; }
81   ulong getReads() { return reads; }
82   ulong getWrites() { return writes; }
83   ulong getWB() { return writeBacks; }
84
85   void writeBack(ulong) { writeBacks++; }
86   void Access(ulong, uchar);
87   void printStats();
88   void updateLRU(cacheLine *);
89 };
90
91 #endif
```

3. Analisis de Resultados

3.1. Preguntas

1. ¿Cuáles son los beneficios de tener un protocolo de coherencia?
   El protocolo de coherencia es sumamente necesario para poder tener múltiples procesadores con caches internos ya que se necesita saber que la variable leída tiene el valor correcto y no fue modificada por otro procesador. Como se puede observar en la figura 1 habría un fallo en la lectura del dato x.

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Figura 1: Coherencia de Cache

2. ¿El protocolo MESI provee estos beneficios? En caso afirmativo explique cómo. Si, ya que este mantiene un estado de todos los datos ingresados a la cache e invalida los datos que fueron modificados por otros procesadores, por lo que cada procesador sabe que no puede utilizarlo y debe hacer una lectura a memoria.

3. ¿Qué debilidades o problemas podría presentar el protocolo de coherencia MESI? Para computadores de pocos cores, se puede volver un método con mucha latencia y lo volvería menos eficiente que un protocolo de snooping. El cual solo se fija si algún otro core utiliza este dato. Además este protocolo puede ser modificado incluyendo un estado extra que es el de Owned (O). Este protocolo se llama MOESI e incluye los estados de MESI más el owned. Este estado previene una lectura a la memoria provocando una ejecución más rápida del cache.

3.2. Resultados

Luego de ejecutar el programa con el archivo de instrucciones dado por la profesora, se puede ver en la figura 2 el miss rate que cada procesador tuvo. Ademas se pueden ver la cantidad de lecturas y escrituras que hubieron.

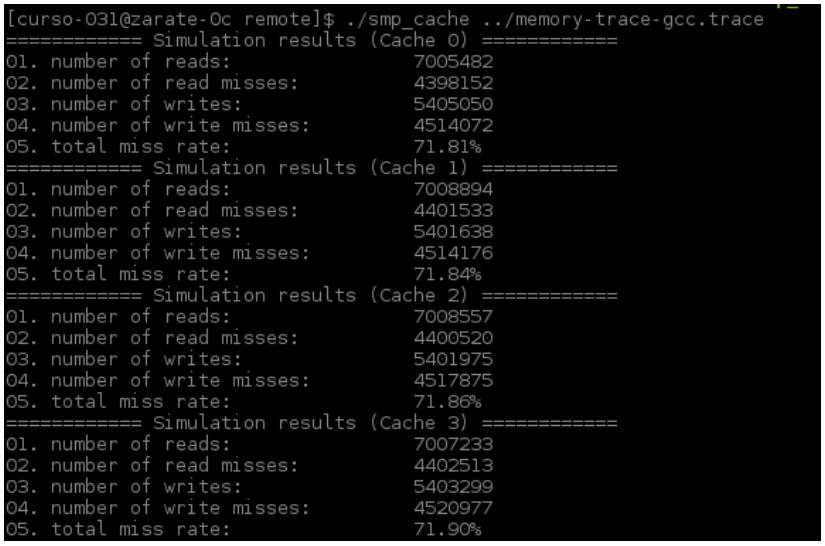

Figura 2: Miss rate de los procesadores

4. Conclusiones

Para tener varios núcleos funcionando a la vez y que estos tengan su propio cache, es necesario que haya coherencia entre ellos. Para esto el protocolo MESI funciona bastante bien ya que logra controlar la lectura de datos validos e impide lecturas de datos inválidos.
También como se pudo observar en la figura 2 tenemos un hit rate de más del 28 % lo cual es una mejora de la velocidad de lectura y escritura de múltiples cores al no tener un cache propio.