1. Objectives

The objective of this project is to learn how to use the SMPCache simulator and to carry out a study of the main cache coherence protocols based on bus.

Thanks to the simulator, important execution parameters can be observed, like the bus transactions, the cache misses, etc. From the number of bus transactions, and making some suppositions about the time cost of the operations to perform, it is possible to compute the approximate execution time of a particular problem.

SMPCache simulator implements three snooping protocols for cache coherence: MSI, MESI and DRAGON. MSI and MESI are invalidation protocols, while DRAGON is an update protocol. This will allow us to estimate which strategy (invalidation or update) is more suitable for certain cases.

2. Requirements

SMPCache uses as input the memory traces obtained from the execution of a program. A memory trace is a list with the memory accesses that are performed. An easy way to obtain memory traces consists in including code for registering the memory accesses that can be done during the execution of a program. The following code is an example of a C++ program that copies one vector to another. Since we have not considered the instruction accesses (captures of instructions), the trace would simply consist in one reading access and one writing access for each of the vector elements:

```
/*
   Example of program for generating multiprocessor memory traces for
   SMPCache simulator.
   This program generates traces simulating the copy of one vector to other,
   without considering the instruction accesses (captures of instructions).
   It is easy to change this operation (copy of one vector to other) for
   other operations with vectors.
   This program generates the trace files for configurations with 1, 2, 4
   and 8 processors.
*/

#include <fstream.h>
#include <stdio.h>

const int LVEC  = 1000; // Vector length
const int NPROC =    8; // Maximum number of processors
const int RD    =    2; // It indicates Read
const int WR    =    3; // It indicates Write

// This function writes a memory access with the simulator format.
// It needs the trace file, the memory access type, and the address
void trace(ofstream &file,int type,void *addr)
{
 file << type << " ";
 file.width(8);
 file.fill('0');
 file << hex << (int)addr << endl;
}
```

```
int main()
{
 int a[LVEC],b[LVEC];  // Test vectors
 ofstream file[NPROC]; // Files for each processor
 int n,i,proc;
 char fname[33];

 for (n=1;n <= NPROC;n*=2)
 {
   for (proc=0;proc < n;proc++)
   {
     sprintf(fname,"c:\\temp\\trace%d_%d.prg",n,proc+1);
     file[proc].open(fname);
   }
   for (i=0;i < LVEC;i++)
   {
     a[i]=b[i]; // Example of operation
       //proc=n*i/LVEC;    // Consecutive distribution
       proc=i % n;      // Interleaved distribution
       trace(file[proc],RD,&b[i]); // Read from b[i]
       trace(file[proc],WR,&a[i]); // Write to a[i]
   }
   for (proc=0;proc < n;proc++) file[proc].close();
 }
 return 0;
}
```

The source file of this program can be found in the web site of the SMPCache simulator. SMPCache uses a trace file for each of the processors that are used. Therefore, if we simulate with only one processor, a file (trace1_1.prg) will be necessary; if we simulate with two processors, two files will be necessary (trace2_1.prg and trace2_2.prg); and so forth. This program generates all the necessary files to be able to simulate a multiprocessor with 1, 2, 4 and 8 processors. There are two numbers in the name of the trace files, the first one is the number of processors for which the trace has been generated, and the second is the specific processor in which the file should be loaded.

In the previous program, there are two distribution types, among the processors, of the tasks to carry out: the consecutive and the interleaved distribution. In consecutive distribution, each processor manages a complete set of consecutive/successive elements of the vector. In interleaved distribution, the elements are distributed among the processors so that successive elements of the vector are in different processors. This allows us to show the false sharing problem.

Starting from this example program we can develop more elaborated programs including more complex traces, for example, studying the access to semaphores and barriers. We can also change the vector size in order to show the direct-mapping problem.

3. Development

3.1. False sharing

Compile and execute the program given in the previous section in order to generate the necessary trace files. Then, execute the SMPCache simulator, and load the traces, doing the simulations for 1, 2, 4 and 8 processors with the different coherence protocols, that is, MSI,

MESI and DRAGON.

The important parameters to highlight in the simulation are the number of bus accesses, and the miss and hit rates for the caches. These parameters can be obtained easily using the view “multiprocessor evolution” in SMPCache.

Since we want to analyse the false sharing, we have to do the whole study with the two distribution kinds for the vector elements, that is, interleaved and consecutive distribution.

The multiprocessor system has to be configured with the following parameters:

- Processors: 8
- Scheme for bus arbitration: LRU
- Word wide: 8 bits
- Words by block: 128
- Blocks in main memory: 8192
- Blocks in cache: 512
- Mapping: Set-Associative
- Cache sets = 256 (two-way set associative caches)
- Replacement policy: LRU

It is not necessary to change the number of processors in the system to simulate with a number smaller than 8 processors, since it is enough with configuring the system with 8 processors and then only loading the traces in those processors we will really simulate.

It is necessary to take notes of the results from each performed simulation, in order to be able to reach conclusions about the behaviour and performance of the multiprocessor.

3.2. Associativity

One of the cache problems is the mapping of the lines from main memory to cache. Direct mapping can cause problems when we try to replace time after time the same line in cache, although this corresponds to different locations in the main memory.

To display this problem we can modify the program of section 2 looking for an appropriate size for the vectors we are using, so that both vectors occupy the same cache line in spite of being in different main memory areas.

After finding the critical size of these vectors, do a simulation using direct mapping, and compare it with the two-way and four-way set associative mapping.

In order to test that the set associative mapping can also fail, introduce a third vector in the program that also shares the cache lines of the other two vectors. Simulate the new program with direct mapping, two-way and four-way set associative mapping.

Since this problem is independent of the number of processors, do the simulations with only one processor.

4. Reports

The conclusions of this project about multiprocessors must be typed. Basically, it is necessary to answer to the following questions:

- How much is the interleaved distribution worse than the consecutive distribution for tasks in a multiprocessor? Why?
- How does the number of processors influence in the false sharing problem? Why?
- What coherence protocol behaves better for the false sharing problem? Why?
- How have you modified the original program in order to show the direct mapping problem? Justify your answer numerically.
- How have you modified the original program in order to display the problem of the two-way set associative mapping? Justify your answer numerically.

All your answers should be justified with the help of graphs.