ESTRUCTURA DE COMPUTADORES (ETSINF)

Lab session 11

MIPS R2000 CACHE MEMORY
DATA CACHE

1. Introduction

This lab session is a continuation of the previous one, where we focused on the code cache. We continue to use the same assembly program as in the previous session, but our target is now to study the operation and performance of the data cache. We also continue to use the *PCSpim-Cache* simulator.

1.1. Goals

This session shares the goals of the previous one, but centred on the data cache this time.

- To observe how cache memory helps reducing access time to data.
- To determine the address structure from the cache point of view
- Analyze how cache organization affects hit rate.

1.2. Material

The material is available on Resource folder at PoliformaT:

- PCSpim-Cache Simulator MIPS R2000.
- Source file "program.s".

2. Test program: vector by constant product

```
                ######################################################
                # Data segment
                ######################################################

                .data 0x10000000
A:              .word 0,1,2,3,4,5,6,7   # Vector A
                .data 0x10001000
B:              .space 32               # Vector B (result)
                .data 0x1000A030
k:              .word 7                 # Scalar constant
dim:            .word 8                 # Vector dimension
```

```
                ######################################################
                # Code segment
                ######################################################

                .text 0x00400000
                .globl __start

__start:        la $a0, A               # $a0 = A address
                la $a1, B               # $a1 = B address
                la $a2, k               # $a1 = k address
                la $a3, dim             # $a2 = dimension address
                jal sax                 # Subroutine call

                ######################################################
                # Execution ending with a system call
                ######################################################

                addi $v0, $zero, 10     # Exit code
                syscall                 # Execution end

                ######################################################
                # Subroutine that computes Y <- k*X
                # $a0 = Starting address of vector X
                # $a1 = Starting address of vector Y
                # $a2 = Address of scalar constant k
                # $a3 = Address of dimension
                ######################################################

sax:            lw $a2, 0($a2)          # $a3 = constant k
                lw $a3, 0($a3)          # $a3 = dimension
loop:           lw $t0, 0($a0)          # Reading X[i] into $t0
                mult $a2, $t0           # Computes k*X[i]
                mflo $t0                # $t0 <- k*X[i] (HI value is 0)
                sw $t0, 0($a1)          # Writing Y[i]
                addi $a0, $a0, 4        # Address of X[i+1]
                addi $a1, $a1, 4        # Address of Y[i+1]
                addi $a3, $a3, -1       # Decrements the number of elements
                bgtz $a3, loop          # Jumps if elements remain
                jr $ra                  # Subroutine return


                .end
```

3. Data cache

We now consider a data cache with the features given in Table 1. We call this the *original* configuration because we will be changing it during this session.

Parameter	Value
Capacity	256 bytes
Mapping	Direct
Block or line size	16 bytes
Write policy	Write through, write allocate

Table 1. *Original* data cache parameters.

In this section we pay attention to data accesses from the program, as a result of *load* and *store* instructions, that is, *lw*, *sw* and their type variations *lb*, *lbu*, *sh*, etc. In the test program, there are two load instructions for constants *k* and *dim*, and then *dim* reads of the source vector A and the same number of writes to the result vector B.

Take the following questions **before you use the simulator**.

1. *Considering the given cache parameters, indicate how many lines are there in the cache and how many lines are needed to store each vector.*

2. *Give the address structure from the point of view of this data cache (bits for tag, line and offset).*

If you apply this structure to the addresses of variables, you should verify that the variables *dim* and *k* are both mapped to line 3 in the data cache.

3. *Give the cache lines where vectors A and B are mapped.*

Note that, if both vectors are mapped to conflicting cache lines, we are going to face a number of conflict misses, given the mapping scheme. We will check this **with the simulator**.

4. *Load the original program in PCSpim-Cache. Configure the data cache with the parameters given in Table 1. Run the program step by step (F10) and observe the effects on the data cache. Verify that vectors are stored in the intended data cache lines and then fill the following table:*

Nr. of data accesses	
Nr. of hits	
Nr. of misses	
Hit rate achieved (H)	

5. *Which data access(es) have caused cache hits?*

6. *Explain whether the low hit rate is due to absence of locality in the program or due to one or more of the cache parameters used.*

In the following experiments, we will explore the effect of three different approaches to improve this low rate:

a) Change the write miss policy to *no-allocate*.
b) Modify the vector addresses in the data segment using `.data` directives.
c) Increase associativity using 2-way set associative mapping.

NOTE:

Each of the proposed changes must be applied **in isolation** with respect to the original configuration given in Table 1. Be careful not to combine them when you move to the next exercise.

3.1. First alternative: change the write policy

In the original configuration, write misses to vector B cause allocation of the missed block. Since vector B is only accessed for writing, we could avoid collision with vector A by simply not allocating B, thus changing the write miss policy to write no-allocate. Let's see if that works. See Table 2 for the cache configuration to use (change highlighted *wrt*. Table 1).

Parameter	**Value**
Capacity	256 bytes
Mapping	Direct
Block or line size	16 bytes
Write policy	Write through, write no-allocate

Table 2. Data cache parameters for first alternative.

7. Configure the data cache in PCSpim-Cache as indicated in Table 2. Now run the original program step by step (F10) to observe its behavior. Then fill the table:

Nr. of data accesses	
Nr. of hits	
Nr. of misses	
Hit rate achieved (H)	

Double check that you have counted one miss for every access to vector *B*, two misses accessing vector *A* (read elements A [0] and A [4]) and a miss reading variable *k*. Note that we have avoided collision between *A* and *B*, since *B* is only accessed for writing and there is no allocation in case of a write miss. However, all accesses to *B* are misses. Let's try another approach.

3.2. Second alternative: change the vectors allocation

A different way to avoid collisions is to modify the allocation of vectors in the data segment. We can do that using the `.data` directive. In the original program, the addresses used just prior to the declarations of A and B are causing collision in the original configuration. We just need to change the starting address of either A or B so that they are mapped to non-conflicting lines in the cache.

> **8.** *Change the directive* `.data 0x10001000` *so that vector B maps to lines 4 and 5 of the cache. This would avoid collision with lines 0 and 1 (vector A) and line 3 (k and dim). Write the new address for vector B here.*

> ***9.*** *Assemble and run the program with this change. Remember to keep the original write policy (write allocate) and fill the following table:*

Nr. of data accesses	
Nr. of hits	
Nr. of misses	
Hit rate achieved (H)	

Compare the hit rate achieved after taking this second alternative.

3.3. Third alternative: increase the cache associativity

The third alternative is to use a more flexible mapping scheme. Restore the original location of vector *B* (`.data 0x10001000` before its declaration) and configure the data cache as shown in Table 3.

Parameter	Value
Capacity	256 bytes
Mapping	2-way set associative
Block or line size	16 bytes
Write policy	Write through, write allocate
Replacement policy	LRU

Table 3. Data cache configuration for third alternative

Using 2-way set associative mapping, a block from main memory may now be brought to any of the two lines of the *set* it maps to.

Before you use the simulator again, take the following questions

10. *Give the address structure from the point of view of this data cache (bits for tag, line and offset).*

11. *Give the cache sets where vectors A and B are mapped.*

Note that the blocks of vectors *A* and *B* map to coincident cache sets. With direct mapping, this caused continuous replacements at run time. Are we in the same situation now? If not, what has changed?

Time to **use the simulator** again.

12. *Load the original program and run it with the new cache configuration (Table 3). Then fill the following table:*

Nr. of data accesses	
Nr. of hits	
Nr. of misses	
Hit rate achieved (H)	

Note that there are no replacements in this case. All misses are compulsory, *i.e.*, caused by the first access to the block. You should only find this type of misses.

Compare alternatives 2 and 3. Which one do you think is preferrable in general?