Lab 5 – Direct and Associative Caches

1 Access Time

Using the C language, the consecutive elements of the rows of multidimensional arrays are stored next to each other in the memory space. Moreover, rows themselves are stored next to each other in the memory space. This storage policy is called *row major*.

Exercice 1 – To check this statement, write a C program that declares an array of size 2x3 and displays the addresses (with indices) of each element of this array.

Regardless of the order in which the array elements are stored in memory, depending on the way array elements are accessed (row by row or column by column), the access time may differ significantly because of cache effects.

Exercice 2 – Write two C programs that declare an array of 1024x1024 integers. The first one initializes the array row by row (to any value), while the second one initializes the array column by column. To check the previous statement, measure the execution time of each program using the command `/usr/bin/time programName` and compare them.

2 Direct Cache

The aim of this lab session is to simulate and to compare a direct cache and a n-way set associative cache. During this session we assume that memory addresses are 32-bit long.

Download the archive `LAB5_resources.tar.gz`. It contains files that you will have to complete and a `makefile` to compile them. In particular, files `bits.h` and `bits.c` contain :

- the macro `#define SWORD(A,N,T) ((A & (((0x1 « T) - 1) « N)) » N)` providing the integer value of the sub-word of size `T` bits starting at bit `N` of word `A`,
- the function `void print_bits(int n)` printing bits of `n`, from left (most significant bit) to rigth (least significant bit).

We want to simulate a cache memory whose blocks are 32-bit long.

Exercice 3 – For a given number of blocks `nb_blocks` (`nb_blocks` being a power of 2), detail the decomposition of a memory address in :

- tag,
- set index,
- offset.

Files `direct_cache.h` and `direct_cache.c` implement a direct cache. Structure `struct block` implements a cache block and stores (using integers, to keep it simple) its tag, its validity bit and its content as a 32-bit memory word. Structure `struct direct_cache` implements a direct cache memory. Its field `nb_blocks` is its number of blocks (it must be a power of 2) and its field `table` is an array of `nb_blocks` elements of type `struct block`. Function `void direct_cache_print(struct direct_cache* c);` (already implemented) prints a direct cache.

Exercice 4 – Complete the following functions :

- `struct direct_cache* direct_cache_create(int nb_blocks)`,
- `void direct_cache_delete(struct direct_cache* c)`,

which allow to initialize a direct cache memory with `nb_blocks` blocks and to free it, respectively. At initialization all tags, validity bits, and block contents are set to 0. Test your functions in `test_direct_cache.c`.

Exercice 5 – Complete the following functions :

- `int get_offset(int address, struct direct_cache* c)`,

— `int get_index(int address, struct direct_cache* c)`,
— `int get_tag(int address, struct direct_cache* c)`,

which achieve the decomposition of an address into a tag, a set index, and an offset. Test your functions in `test_direct_cache.c`.

Function `int set_direct_cache(int address, int word, struct direct_cache* c)` loads in cache `c` the word `word` located at address `address` in memory. It simulates a copy of a memory block into the cache. Function `int lw_direct_cache(int* reg, int address, struct direct_cache* c)` is equivalent to the MIPS instruction `lw` : its goal is to load the content at address `address` in memory to the register `reg`. We first look for that word in the cache. If it is in the cache (hit), `reg` is set and the function returns 1. Otherwise (miss), we look for the word in main memory and it is copied into the cache (this can be simulated with function `set_direct_cache(...)`, with any value for `word`). In this case the function returns 0.

Implement functions `set_direct_cache(...)` and `lw_direct_cache(...)`. Because our goal is only to study performance, we may use any value for the memory words. Moreover, `reg` may be `NULL`. In this case it will not be set. Test your functions in `test_direct_cache.c`.

We want now to simulate a direct cache. The parameters of the simulation are :
— the number of blocks in the cache,
— the range of available memory addresses,
— the number of write requests.

File `direct_cache_simulation.c` contains function `int rand_address(int inf, int sup)` which returns a random address in the address range [`inf`, `sup`].

Exercice 6 – Complete function `int main(int argc, char* argv[])` to achieve a program simulating the use of a direct cache,. Simulation parameters will be passed to the program though command line options. The final number of hits and misses will be print on the terminal.

3 n-Way Set Associative Cache

We now want to simulate a n-way set associative cache memory and to compare it with a direct cache memory. Structure `struct associative_cache` from file `associative_cache.h` implements a n-way set associative cache. Its field `nb_tables` specifies its number of sets (each of them is similar to a direct cache memory), and its field `direct_table` is an array of pointers to structures `struct direct_cache`.

Exercice 7 – In `associative_cache.c`, complete the following functions :
— `struct associative_cache* associative_create(int nb_blocks, int nb_tables)` which creates an associative cache with `nb_tables` sets that contain `nb_blocks` blocks (ways) each,
— `void associative_cache_delete(struct associative_cache* c)` which frees a `struct associative_cache` structure.

Test your functions in `test_associative_cache.c`

Exercice 8 – Complete the following functions :
— `int set_associative_cache(int address, int word, struct associative_cache *c)`,
— `int lw_associative_cache(int* reg, int address, struct associative_cache *c)`.

These functions work in a similar way than their direct cache counterparts, but according to set associative cache behavior for the `set_associative_cache(...)` function, where a replacement policy has to be implemented (which simple policy may you implement ?). Function `lw_associative_cache(...)` has a similar behavior than `lw_direct_cache(...)`.

Exercice 9 – In file `associative_cache_simulation.c`, write the function `int main(int argc, char* argv[])` to build a program that reads the parameters of a n-way set associative cache simulation, then runs it and finally prints the number of hits and misses.