Computer Architecture Cache Report

Nishi Baranwal, CS23BTECH11041
Paidala Vindhya, CS23BTECH11044

Contents

1 Coding Approach

1.1 Design Decisions

- We created a new .cpp file (cache_gen_func.cpp) for all the functions related to Cache. This provides better readablity of code and segregation of functions.
- **riscv_header.h** header file consists of all the function declarations and the global variables which are written in other .cpp files. The header file is then included in the .cpp files
- In addition to the functions, data structures, variables (from the Simulator and Assembler) we have added the following to simulate the Cache.
- In the **cache_gen_func.cpp** we have the following general functions.
  - isPower_of_2
  - inValidate_cache
  - print_Cache_Statistics
  - updateMemory
  - updateCache
  - parse_address
  - getDataFromMemory
  - find_min_rpOrder

  - getDataFromCache
  - storeDataInMemory

- We have defined the following global variables for our usage.
  - isCacheEnabled - type bool
  - configDetails - type struct cache_info
  - stats - type struct cacheStats
  - cache - map from int to vector of struct blockDetails
  - cache_output_filename - type string

- We have created the following structs (given with their data members) to simulate different parts of cache:
  - **cache_info**
    - int cacheSize;
    - int blockSize;
    - int associativity;
    - string replacementPolicy;
    - string writeBackPolicy;
    - int rows;
  - **cacheStats**
    - int access;
    - int hit;
    - int miss;
  - **blockDetails**
    - bool isValid;
    - bool isDirty;
    - int tag;
    - string data;
    - int rpOrder;

1.2 Implementation (Explanation of Code)

- In extension to the Assembler and Simulator implementations we have the following
- We have defined a number of global variables which are extern to access across multiple files. They are:
  - **isCacheEnabled**: It is a variable of type bool. It is assigned true if the cache is enabled and false otherwise. Since the cache is disabled by default, it is initially set to false.
  - **configDetails**: It is of type struct **cache_info** which stores all configuration details such as cache size, block size, associativity, replacement policy, write back policy and count of rows (i.e. the number of sets or entries in the cache) which are read from the config file provided.

  - **stats**: It is of type struct **cacheStats**, which stores the cache statistics: the number of hits, misses, and the total number of accesses performed.
  - **blockDetails**: It is a struct which stores information related to that particular block which includes isValid(bool), isDirty(bool), tag(int), data(string) and rpOrder(int). The variable rpOrder is used to keep track of the order in which the blocks are accessed in case of 'LRU' or to know which block was all as as ocated first in case of 'FIFO' replacement policies.
  - **cache** : It is a map from 'int' to 'vector<blockDetails>', where the key is the index and the value is the vector containing the blocks (of that index). The size of the vector is equal to the associativity i.e., number of blocks is equal to the associativity.
  - **cache_output_filename** : It stores the filename of the file which contains the cache simulation data.

- The new cache commands used in main function while loop is running infinitely are:

  - **cache_sim enable config_file**:
    * It assigns true to the global isCacheEnabled variable.
    * Reads the config_file and stores them in the configDetails variable. But while storing, if the associativity = 0 i.e., if fully associative, it will store associativity as capacity/block size.
    * It resizes the vectors of cache to associativity and initializes cache blocks with its default values by calling invalidate_cache function.
    * It initializes all variables(access, hit and miss counters) of stats to zero.
  - **cache_sim disable**: It assigns false to the global isCacheEnabled variable.
  - **cache_sim status**: If cache is enabled, prints the cache configuration values and if disabled, it prints "Cache is disabled".
  - **cache_sim invalidate**: It initializes cache to its default values by calling invalidate_cache() function.
  - **cache_sim dump file**: It writes all the current valid D-cache entries to the file.
  - **cache_sim stats**: It prints the cache statistics of the code in execution at that point of time by calling print_Cache_Statistics() function.

- Functions used in cache implementation:

  - **invalidate_cache()**: It assigns all its cache block entries to default values i.e., isValid and isDirty to false, tag to -1, data as empty string, rpOrder to 0. If any block is dirty, then writes its value to the memory.
  - **print_Cache_Statistics()**: Prints cache statistics of the the program in execution at that instance of time which includes access, hit and miss counts and hit rate.
  - **updateMemory(address, size, data)**: It takes address, data and size of the data as arguments. This data is written into memory at that address.
  - **updateCache(index, blockNo, byteOffset, size, data)**: It updates cache block with the data provided in argument, corresponding to that 'index' and 'blockNo' from that specified byte offset onwards.
  - **parse_address(address)**: It parses the address using the cache cofiguration details provided, and returns a vector<int> containing the tag, index number and the byte offset values.

- **getDataFromMemory (address)**: It returns data from the memory (from the address given as argument) as a string of hexadecimal characters whose size is the block size of cache.
- **find_min_rpOrder(index)**: This function is used in the 'LRU' and 'FIFO' replacement policies to get the block number with the minimum rpOrder value. In the case of the 'LRU' policy, a minimum rpOrder value indicates that the block has not been used recently. Since, in the 'LRU' policy, the rpOrder variable is updated whenever that block is accessed, rpOrder is set to the access counter value. In the 'FIFO' policy, a minimum rpOrder value means that the block has been present the longest. Since, in the 'FIFO' policy, the rpOrder variable is updated only on the block's initial allocation, it is set to the access counter value only during its first allocation.
- **getDataFromCache(address, size)**:
  - This function is called in 'load' instructions when the cache is enable. It takes the address from where it wants the data and size which is the number of bytes of data it wants as arguments. It returns the data as a string which has the hexadecimal characters.
  - It calls the parse_address function to get the tag, index and the byte offset values.
  - Checks if it is a cache hit or miss:
    - If hit, we find the block no.
    - If it's a miss, we find the block number of any invalid block or the block which is to be replaced according to the replacement policy. While replacing, if the block is dirty, we will update its data in memory and make it clean. Now, after evicting the victim block, it gets the data from the memory and stores it in that block.
    - Once we get the block no., we will find the data using the byte offset, index and block no. This data will be returned.
    - Access, hit and miss variables are updated accordingly.
    - These cache access details are written to the output file.
- **storeDataInMemory(address, size, data)**:
  - This function is called in 'store' instructions when the cache is enabled. It modifies the data in the cache in the address specified. It also takes data and size of the data to be written as arguments.
  - It calls the parse_address function to get the tag, index and the byte offset values.
  - Finds if it's a cache hit or miss:
    - if it is a Hit::
      - in WT: update both memory and cache
      - in WB : update only cache and make that block's dirty bit to 1.
    - if it is a Miss::
      - in WT: update only memory(Since it is without allocate)
      - in WB : we find the block number of any invalid block or the block which is to be replaced according to the replacement policy. While replacing, if the block is dirty, we will update its data in memory and make it clean. Now, at this location it first gets data from memory and then updates data in that block with the argument's data. So, the block is turned dirty.
  - Access, hit and miss variables are updated accordingly
  - These cache access details are written to the output file.

2 Testing

- We checked the different combinations of the replacement and the write policies.
- We checked for different values for set associativity.
- We checked for varying values of the cache size, block size.
- We checked if the cache and memory synchronization is occurring correctly in both the write-back with allocate and the write-through without allocate write policies.
- We checked the cache outputs, enable, disable working. Whether the enable, disable commands is working mid execution or not.
- We checked the cache invalidation process when the invalidate command is called separately or when we load another file mid-way or after the execution of the previous input file.
- We checked with load, store instructions of different data sizes (b,h,w,d) and also with different byte offsets.
- We cross checked our answers for all (except FIFO replacement policy) with RIPES.
- To check the replacement policies - LRU, FIFO; we printed the ordering variable in the output file to track and verify whether it is working properly.
- We ran the Homework-4 assignment with the different configurations given on our simulator and cross checked the answers.
- We created test cases to clearly check and see the cache behavior with different configurations (including FIFO replacement policy). We have included our test case input file in the Test Cases 3 file.

3 Facilities

- We have supported all parts of the project:
  - **Part-1**: Only support read access modeling for a direct-mapped cache
  - **Part-2**: Part-1 + caches with associativity + FIFO, LRU and RANDOM replacement policies
  - **Part-3**: Part-2 + write access modeling
- We support all types - R-Type, I-Type, S-Type, U-Type, B-Type, J-Type of instructions. It supports read access, write access in cache with associativity following any of the policies ‘LRU’, ‘FIFO’ or ‘RANDOM’ specified.
- We support all functionalities cache enable, disable, status, invalidate, dumping cache entries to a file, displaying cache statistics and also writing cache simulation data to the output file.
- We support all the functionalities from the Simulator and Assembler part - load, run, regs, exit, mem <addr> <count>, step, show-stack, break <line>, del break <line>.

- If the cache config file has errors, we just print error and proceed with the Cache disabled. This allows user to fix their config file without the program crashing.
- For the cache config file we support case insensitive values for replacement and write policies. We support associativity range till 16.
- Whenever there is an erroneous line in the input file, the error with the line number is printed on the terminal. The error is also printed on the corresponding line in the output.hex file (except for the case of .data section errors)
- The code runs through the entire input file while loading it even when it encounters an error. So the errors are given all at once.
- For the .text section (the instructions) we facilitate both hexadecimal immediate values and decimal immediate values as input.
- For the .data section facilitate both decimal values as well as non-negative hexadecimal values. The hexadecimal number can have both uppercase and lowercase for the non - digit characters.
- We facilitate both labels and immediate values (decimal or hexadecimal) as input for branch or jump instructions.
- We facilitate storing of hexcode in the text section of memory and also the facility to view it.
- The Error message (both on the terminal and the output file) clearly states why that error is coming. So, it becomes easier for the user to fix their code.

4 Limitations

- It only allows the cache size, block size and the associativity to be powers of 2, with a maximum associativity of 16.
- It only allows LRU, FIFO, and RANDOM replacement policies, and Write-back with allocate and Write-through without allocate write policies.
- It does not facilitate negative hexadecimal values for the .data section.
- There is strict format rules regarding spacing, commas, colons, blank lines. The program is not flexible enough to accept them as correct input and give out the machine code or to execute then. If the user does not follow the input format, an error message is printed and the program goes to the next instruction (if any).
- It facilitates only RISC-V (RV64I) variant assembly code.
- It does not support comments and pseudo instructions.

5 Challenges Faced

- Made a few changes in the simulator part where the test cases failed - the bgeu, bltu, srl, srli instructions and breakpoints indicator variable.

- For choosing the appropriate data type for the cache, we followed a similar approach to data_mem. As the data is stored as hexadecimal characters in data_mem, the data in each cache block is also stored as a hexadecimal string. Since we are handling addresses as int types, we have also used the int data type for the tag and index.
- In assigning value to the counter variable which stores order in which blocks are accessed or allotted in cache when using replacement policies such as 'LRU' and 'FIFO'. Basically, tracking the order of accesses or time of entry with these replacement policies.
- To prevent the functioning of the cache enable or disable commands after loading until the execution of that file is finished. It gives an error message saying *"Cannot enable/change cache configs after loading the input file."* if the commands are used.
- To prevent users from entering incorrect or invalid cache parameters in the config file. If incorrect values are entered, the cache is simply disabled.
- While updating cache, address wise addressing the data in cache block was mistakenly interpreted as index wise addressing, which resulted in storing numbers in reverse. Also since we are storing data as hexadecimal characters, size of cache data in each block is 2*block_size (Since, each byte is equivalent to 2 hex characters) which initially we had mistakenly taken as block_size.
- For testing, we needed to consider different aspects and combinations of cache configurations for the test cases. We came up with various test cases to observe how the memory and cache synchronize etc.