COL-216 ASSIGNMENT-5

((MULTI-CORE PROCESSORS))

HARIKESH 2019CS10355

SOURAV 2019CS10404

APPROACH FOR THE CODE

- PREVIOUSLY, WE HAVE DESIGNED A BASIC INTERPRETER IN C++ FOR HANDLING A SUBSET OF THE INSTRUCTIONS AND A MIPS SIMULATOR TOO. WE HAD BEEN DEVELOPING THE MAIN MEMORY AND THEN ARE INTEGRATING IT INTO MORE ADVANCED INTERPRETER THAN THE PREVIOUS ONE.
- NOW WE ARE EXTENDING OUR MODEL OF A DRAM FOR MULTI-CORE PROCESSORS. FOR THIS, WE ARE GIVEN N NUMBER OF DIFFERENT PROGRAM FILES IN MIPS (T1.TXT, T2.TXT, ...) ALONG WITH N CPU CORES EACH FOR A FILE.
- WE HAVE ALSO BEEN GIVEN A PARAMETER M WHICH IS THE SIMULATION TIME I.E. NUMBER OF CYCLES.
- NOW OUR MAIN MEMORY HAS TO TAKE UP TO 2^20 BYTES. SO, WE ARE STORING ADDRESSES INTO AN 2-D ARRAY OF SIZE 1024*1024. WE START BY DIVIDING OUR MAIN MEMORY THAT IS DRAM INTO N EQUAL PARTS EACH FOR A CORE SO THAT THE DATA OF DIFFERENT PROGRAMS DON'T CORRUPT THE OTHERS. WE ARE USING THE CONCEPT OF RELATIVE ADDRESS FOR DIFFERENT FILES SO THAT THE DATA DOESN'T CORRUPT AND ALL OUR CORES ARE INDEPENDENT TOO.
- NOW WE START DECODING ALL THE FILES AND THEN ARE MAPPING THEM CORRESPONDINGLY WITH THE CYCLE NUMBERS OF THAT CORRESPONDING CORE. ONE THING TO TAKE CARE OF IS THAT BOTH DRAM AND MRM (MEMORY REQUEST MANAGER) ARE INDEPENDENT. SO, THERE CAN BE MORE THAN ONE REQUEST WAITING IN THE MRM BUT ONLY ONE REQUEST WILL BE ENTERTAINED AT A TIME IN DRAM. MRM IS NOTHING BUT KIND OF A FINITE BUFFER FOR STORING THE WAITING REQUESTS.

- Now our other instructions i.e. except lw and sw are not affected by this DRAM (Dynamic Random-Access Memory) implementation and still require only 1 cycle to complete that operation.
- In case of lw and sw commands we have to take this case of DRAM and hence we have to first check the row number in which the work of our importance is there and after that check the column offset and these 2 things are given as an input to us which is row_access_delay and column_access_delay and hence we are then calculating the total number of cycles required in this case. Not only this but this time we are also trying to reduce the total number of cycles taken in some cases of memory read and write operations. We are doing that by storing all such requests in a data structure and then re-ordering all those DRAM requests according to the concept that will take two such instructions of the same row together and then implement that.
- All the reductant instructions are removed also this time which is if some instruction is coming more than once in the file then that instruction is executed only once.
- Apart from that, we are mapping all those instructions and their registers with something to found those easily.
- The total cycles required in that case can be easily found by considering these things. Now suppose we are having a particular row in our row buffer and the next instruction details also lie in that then we don't need to get that row back and then do the computation. We can update without going back. Else we have to go back in if it is not the same row and bring the other one.
- So, we have to do this till our file is completely read by us or the simulation time is over and during each cycle we have to print some basic details and finally after completing that we have to print the number of row updations and total number of cycles as in the previous assignment. We do this by keeping track of row number and cycle number of all the files given to us.
- Our task is to maximize the throughput/ IPC or CPI in the given simulation time.

Assumptions and Basic Notes

- We have assumed that there is no comment in the text files given to us but we are dealing the case of empty lines.
- Programs running in the different CPU cores are independent of each other i.e. they all have their different set of registers.
- It is assumed that the instructions are not stored in the main memory and hence we access DRAM only for lw/sw instructions.
- Jump and other branch instructions are also taken care of this time.
- It is assumed that the number of cores is finite and we have assumed that number to be less than 1024 just to equally divide the DRAM.
- Printing "Wrong output" in case some register is out of scope or there is some kind of error in MIPS instructions. Also have taken the input format in the context that was given and asked for.
- The MIPS file may or may not end with "exit:".
- (Same architectural ana ISA assumptions as in assignment 3).

Testing Strategy

- We have tested our code for some easy text files which includes file with only 2 to 3 instructions and empty file (ignored in this case).
- Have checked the result intensively for a single-core processor also just by taking the value of N=1 and rechecked the result. We also tested it with as many as 6 files.
- We have tested by taking both limits of simulation time i.e. taking very large simulation time and very little simulation time also.

- Tested with some errors in one file and others correct (in this case the execution of that one file stop only). Tested by giving wrong commands or out-of-scope registers and print necessary error.
- Taken some files with only sw and lw commands and otherwise.
- The case of jump and other branch instructions is also handled here by taking labels at different places in the file.
- As which file will be executed first also affect our throughput efficiency we have taken two sets of files with the same files shuffled among themselves to see the clear difference between the throughput.
- Take a file with a good number of instructions and just shuffled the instructions and tried to reduce the number of cycles as much as we can to make our code efficient and then do this to every file to maximize the throughput.
- Checked the value of each and every register after running the code and confirmed it by doing some basic manual calculations.

Strengths and Weaknesses of our code

- This implementation is handling jump and branch statements too unlike the previous one. There is also a possibility of little much inefficiency in recording in some test cases. There is a high chance of branch hazards here in our program as loops can be of any size and we are not giving any specific memory for taking care of all such loops and not executing the instructions in the loop again and again.
- The case of starvation is ignored this time also (according to Piazza). This will overall increase our throughput efficiency but the result is compromised with the priority.
- We have tried to implement the concept of pipelining intensively and had done forwarding in instructions waiting in the MRM only as then only we can reorder the instructions of a file.
- As we are finding out how many other DRAM requests are there in the next lines of the given file (we are searching up to the extent of the cycles take in first load or store instruction). So if some test case has these instructions repeating correctly then our

code is efficient but our code is highly inefficient in those cases where these commands are not repeating simultaneously.

- Also, we are just going up to the delays to check any other instruction belonging to the same row to reorder those requests to make our implementation efficient but one can say that there are millions of lines in real-world codes so maybe our code is not so efficient. But strength of our code lies in the fact that any other code giving efficient output, in that case, will require advanced hardware and hence will have more cost than ours, and also it will require more space than ours.
- We are using non-blocking memory in our case so any independent instruction will not require an extra cycle and will be done alongside DRAM and thus make our code efficient.

Delay Estimation

In our MRM, we have taken the delay as one cycle only which is of course not the practical case as the MRM is doing all the computation and arrangements like which instruction will be implemented next in the sequence where all the instructions are waiting for their DRAM requests to be issued. Now doing so is fine as an assumption only because if we consider all the sequencing also to be done by our hardware only then it will take as many cycles as the number of sequences waiting there and our throughput will be affected badly in that case. On the other hand, if we want to do that in a single cycle we need many different flip flops and ports etc and thus the area will be increased. So, we are compromising the area of the hardware piece just to have a better throughput efficiency. Hence we have assumed all the copying of the data that the MRM does from the files to the DRAM in constant time only for better throughput results.

Thanks !!