COL 216: Computer Architecture

Assignment 5

Preface:

In assignment 4 we had implemented a DRAM memory management for a single-core processor. Now in this assignment we extend our single core implementation to N cores where the N is provided by the user. Each of the N cores is independent in terms of their registers but share the same DRAM, and they run simultaneously N different MIPS program.

As there is one common memory (although disjoint) for the N cores we need a **Memory Request Manager (MRM)** to handle memory accesses. This MRM has a delay which needs to be accommodated in the clock timing. Our main aim is to maximize the throughput of the system.

Changes from Assignment 4:

- Basic input and output handling is the same. What we have done in this assignment is give a better structure to the codes using inheritance. We have added a **Waiter** class (DRAM requests), **DRAM** class (handling of DRAM), **MIPS** class (mips program handling), **Register** class and **Queue** class.
- As pointed out by Prof. Panda in the help session of assignment 4, we had to make our simulation as close to hardware as possible, so here are those changes:
    - To easy hardware implementation we have taken queues which are implemented through arrays. So, 1024 arrays each of 32 size only is considered.
    - Before **executeInst** executed the tokenized instruction of a single core. Now it feed an array of tokenized instruction to **doIns** of the Dram class for execution.
- Functions like **InnitTask** (intialise the task in DRAM), **doIns** (multicore execution of instructions), **performInst** (non lw-sw instruction), **handleBlock** (blocked core handling) etc. are put to manage multiple cores.

Our Approach:

So, basically to implement the multicore we have an array of N register set each for one core named **allReg**, we have an array of N size **stuck** which keeps track of whether core is stuck (value of the array for that index would be -1), completed (value would be -2) or which instruction to jump to (0 for the next instruction or line no to which we have top jump) , then an array of size N **program** which is the MIPS object for each core. We use an array indexed by core number which contains the instruction that is being executed in that particular core.

So, first all the text files are converted to MIPS object containing the tokenized instruction set which is done by **lineToken** function. After this now the **executeInst** provides the array of instruction which is to be executed. Now there are a few conditions on which the instructions are set in the array which are:

- If core is stuck or completed (stuck[i]==-1 || -2) then don't change the instruction which is already present.
- Else provide the next instruction if stuck[i] is 0 or the instruction to where it is jumped.

This array of instruction is provided to **doIns** which now further takes work. Now if there is any dependency it blocks the core through stuck[i] and further **handleBlock** will take care of the execution. If there isn't any dependency then **performInst** will take care of non lw/sw instruction and waiter is initialized for lw/sw. Now here we also take care of **redundancy** such as 2 lw into the same register or two sw from the same address.

This process is repeated clock cycle after clock cycle till clock cycle reaches M. Then we stop execution and print the relevant information.

Constraints:

1. Initially all the registered are set to 0.
2. Memory values used will be multiple of 4 only, no intermediate memory change available.
3. Maximum memory size is 2^20, any value more than that will cause "Overflow Error".
4. Since the size of a row buffer is 32 lw request in a particular row cannot exceed 32.
5. The number of cores should be a multiple of 2 for the better management of memory.

MRM Implementation:

- We have stored the **priority** that is given to cores. We have **FIFO** (first in first out) logic for priorities, i.e. the core that get stuck the first is given highest priority.
- To make sure we need to check for **dependency** every time for an stuck instruction, we store its dependency information at first computation and as instructions gets executed in DRAM, we modify that, saving some clocks.
- So, the core idea is once the core get stucked, add it in **priority** and once an DRAM execution is done modify the **dependency.** Once dependency becomes empty, remove the first element of priority and resume the corresponding core by making stuck to -1.

Design Choices:

- We have stored dependency of only the highest priority core, and for lower priority cores we have to again find it. We choose this because storing for all cores can be very expensive memory wise.
- We choose not to destroy the redundant lw/sw, just add bubbles in its place, i.e. they cant be called but are still there in memory. This is because to destroy them we had to shift every instruction in Queue after it, that is a complex ask and we had to devote clocks to it.

Strengths:

- This queue implementation using array is very easy to implement in hardware (basic memory element).
- Once any redundant lw/sw occurs we remove it (by introducing bubbles in queue) saving many clocks.
- Dependency check and DRAM execution are done parallelly (as if they are different blocks in pipeline) that too saving clocks.
- For dependency check we have first check if queue is empty or not (that can be saved as 1 bit not taking any clock to check) to save unnecessary clock delay.
- A disjoint memory for cores avoids data corruption.

Weakness:

- Few more clock cycles could be saved by forwarding and other optimizations like saturation.
- We have fixed size buffer for each row , so more than 32 requests per row can't be accommodated here.
- This implementation would mean that some cores maybe stuck for long time if other cores are stuck or they have higher priority in the DRAM request.
- Can't have cores not in power of 2 becuase creating disjoint memory sets for this would be difficult.

Input Handling:

From the command line we take the following values in the format:

$ /a.out N M <N textfiles > row_access_delay col_access_delay

Here N is the total no of cores (it has to be a multiple of 2 for easy hardware implementation) and M is the no of clock cycles for which it is to run.

We take these N files and create their input streams and while checking the syntactical errors and rest we store them in N MIPS object which contains a vector of these instructions which are tokenized (by the lineToken function).

After the tokenization we call the **executeInst** function. This will furnish the required N instructions which will be executed in that particular clock cycle in the form of an array of N tokenized instruction. This is fed to the **doIns** of ram which executes the program.

Error Handling:

For every command we have type matched them the line tokens with the format that it is supposed to be and if there is some type mismatch then we ended the program with appropriate error line. Also if type is correct but the execution is not possible or meaningful like jumping to a line out of scope of program or reading or writing the value in memory that is more than the range, then also we have thrown appropriate error.

We have tried our code to be as complete as possible, that can handle every possible combination of syntactically correct or incorrect commands.

Moreover, we have also taken into account the **ignoring of comments**.

Testing Strategy:

We have extensively tested this assignement to gurantee that it doesn't have any errors. We used a python function checker.py for checking whether the multicore evaluation gives the same register values as the single core in assignemnt 4. There is an has been 2 strategy for testing which are as follow:

1. **Randomized testing:** In this all the testcases which we had taken for previous assignment was used for the multicore case . With different files given to different cores. And then we used the checker.py to compare the values from the single core case to this to confirm the values. All the testcases considered has been provided in the files.
2. **Tragetted Testing:** This is the more important one. In this we have custumized the testcases according to specific weakness which the code might have . We have identified such threats which are:
    I. Dependency and Redundency together.
    II. All cores stuck .
    III. Priority functioning among all cores.
    IV. Dependecy of lw-lw kind,lw-sw kind and lw-add-lw kind.
    V. lw/sw and non lw/sw occuring in same clock.

Now for all these cases we have test cases and we ran then in order to identify if these is bugs but it wasn't the caase.