Design for Assignment 4

Shrey J. Patel
2019CS10400

April 2021

1 Aim

To extend and enhance the DRAM memory model of the MIPS simulator(written in C++) prepared in assignment 3 and minor by introducing **Memory Request Ordering**, which reorders the DRAM operations so that there is minimum time loss in row/column delays.

2 I/O Specifications

- **Input:**
  1. MIPS assembly language program in a text file.
  2. DRAM specifications: ROW_ACCESS_DELAY and COL_ACCESS_DELAY.
- **Output:**
  1. After every instruction:
     - Current instruction and its number, instr address is: (Instruction no. printed -1)*4
     - Clock cycle(s) required for that instruction.
     - Modified registers, if any.
     - Modified memory locations, if any.
     - Activity on DRAM.
  2. After completion of execution:
     - Total number of clock cycles used, including those required for write back on last DRAM operation.
     - Total number of row buffer updates.
     - Number of executions for each instruction.
     - Final list of non-zero memory locations.
     - Final values of non-zero registers.

- **Execution Instructions:**
  - The code is written in the file assignment4.cpp. It can be compiled with the command '*make*' in the command line.
  - The executable generated is assignment4.exe which can be run by typing the following in command line:
    ./assignment4.exe input_file_name row_delay col_delay
  - Note that row and column delays will use their default values of 10 and 2, if not explicitly provided.
  - The executable files can be removed using the command '*make clean*'.

3 Approach

Our simulator at the end of minor could handle DRAM memory model as well as non-blocking memory, i.e. it was able to separate the functioning of the processor and memory by allowing the processor to execute normal instructions while the memory was busy in lw/sw operations. This resulted in reduction of overall clock cycles because of this parallel functioning. But the problem that still remains is that the row access and column access delays still cause the clock cycles to increase and in worst case, it can result in very large number of clock cycles. For instance, in case our MIPS program consists only of lw operations, such that all adjacent lw instructions access different rows of DRAM memory, every lw instruction will cause maximum delay as each of them will request a row buffer update.

Thus, to improve the number of clock cycles even further, we have implemented a re-ordering mechanism to our simulator, which allows the lw/sw instructions to re-arrange themselves so as to cause minimum number of row buffer updates and therefore reduce the total delay time, without causing any change in the sequence of other instructions and also taking care of potential conflicts in registers i.e. it also handles unsafe instructions.

We identified that, because we implemented a queue in our earlier code, all the memory related operations were executed in a fixed order without much flexibility. So, we replaced the queue with a map which stores all the pending memory operations along with the row and column of the memory to be accessed, as well as the the register it uses and the corresponding instruction number(global counter). We have also stored the currently engaged(unsafe) registers in an unordered map called *registerUpdate*. Additionally, we have maintained an alternative clock called req_time which pre-computes the time required from the current clock cycle to process the completion of a register which is required in another instruction.

All this combined information helps us to decide which memory instruction to execute next out of the ones stored in the waiting list(map).

4 Code Design and Implementation in C++

The code of this assignment is present in the file assignment4.cpp, and the additional data structures and algorithms that we have added to the simulator prepared before, are described below:

4.1 Data Structures

1. **waitingList:** This map is the main data structure which stores pending lw/sw instuctions, in a flexible order(unlike queue). For a particular memory operation, it stores the row and

column indices of the requested memory address, the register which it uses, and the counter which indicates how many operations have been executed before this.

This waiting list contains maps, all of which map an integer to a queue of (string,integer) pairs. And so, the outer list is indexed by row number while the inner map is indexed by the column number of the memory address required.

Note that both the inner and outer maps are ordered, so as to maintain a sequence of lw/sw operations.

2. **registerUpdate:** An unordered map which stores the engaged registers along with the counter and the last address where they were updated. So, if a register is present in this map, it means this register is queued for some memory operation, and before using it anywhere else, we need to process its completion.

3. **store**: A tuple which stores the information to be printed once the next DRAM instruction/operation has finished execution. In particular, the tuple is of the format (string,string,string,string,int) which stores two different kinds of tuples depending on whether the next operation is a load or store operation:

   (a) (sw, clockCycles, address range, value, counter)

   (b) (lw, clockCycles, register name, value, counter)

4. **insCount**: A vector of integers which stores which operation corresponds to which instruction in the input file. It is used for printing the corresponding instruction to every cycle description in the output.

5. **forRefusing**: One more feature we have added in this assignment is forwarding, in which we keep track of recently updated memory addresses so that any future load requests on the same address may not require us to execute the complete DRAM operation which causes delays, potentially saving time, as we can fetch the already acquired values. So, this unordered map contains the pair (string,int) which stores the register(which stores the recently acquired value) and the last count where it was updated.

4.2 Algorithm

The main objective of our algorithm is to minimize the number of clock cycles even further which is achieved by re-ordering these operations to minimize the row/col access delays and by forwarding, which helps us to skip redundant operations. And all of this is achieved while also handling conflicts in register accesses. The algorithm is divided into several procedures:

- When we encounter any 'lw' instruction, we check whether the required memory address has already been handled by forwarding, and if it is so, we don't require to issue any DRAM request as we already have the value. Otherwise, we issue a new DRAM request and add the details of the instruction to the waiting list. Additionally, lw instructions can also be unsafe if the address itself uses an engaged register, in which case we first call completeRegister to complete the execution of that register. Same is true for an 'sw' command.

- When we encounter an 'sw', we first check if the register whose value we have to store is already engaged or queued in any other lw instruction. If it is so, we first process the completion of that register, and then add this sw instruction to the execution queue. Otherwise, we directly add it to the waiting list.

Note that we also save the address and its corresponding value where we are storing so that we can potentially avoid further load operations through forwarding.

- One of our weaknesses till minor was the inefficient usage of space, which causes a problem if the input program contains a large number of memory related operations. To resolve this we have limited the size of our waiting list to 64, which means that the re-ordering can be done only for 64 instructions at a time, and beyond that, it waits for the next instruction in the waiting list to complete its execution before queuing the incoming operation.

- **getCommand()**: This function returns the next instruction to be executed after re-ordering the current waiting list such that there is minimum amount of delay. This function prioritizes those instructions which use the current row buffer, i.e. those which don't cause any additional row access delay. If they are multiple such instructions, all of them are executed in order of their counter values, i.e. in the same order in which they appear in the original program. This return format of this function is a tuple of the form (int,int,int,string), which is the same format stored in the waiting list.

  An important feature of this function is that it ignores redundant operations and removes them from the waiting list without executing them. One example of a redundant instruction is:

  lw $t0, 1000
  lw $t0, 1004

  The final value in the register t0 will depend only on the second instruction, so the first lw instruction is redundant and it would save us time if we don't explicitly process it.

- **processCommand(tuple)**: This procedure handles the execution of the next scheduled command as a tuple obtained from getCOmmand().

  This function handles the calculation of the clock cycles elapsed till now as well as the alternate clock (time_req).

  It is also involved in updating the tuple 'store', which stores the details of the recently executed operation.

- **processCompletion()**: This causes the information stored in 'store' updated in the above function to be printed on the console at the appropriate clock cycle along with the corresponding instruction and the counter.

  This procedure is responsible for refreshing the values of the store data structure and the alternate cycle. This alternate clock tracks the current state of the processor. So, when its value is (-1), the processor is idle and ready to process. But note that the processor and the memory being separate, only the issuing of requests and the execution of normal instructions are handled by the processor, while the actual execution of DRAM requests are handled by the DRAM memory in a real simulator.

- **completeRegister(reg_name)**: This function is called in case of an unsafe instruction(a normal instruction or an 'sw' instruction which uses a register which is already present registerUpdate). This procedure tries to force an immediate execution of the operation(s) which use this register, and this function returns as soon as all the commands which use this register have all completed their execution.

4.3 Testing & Exception Handling

1. Testing:- We have manually created our own test cases and provided them as input to the assignment4.cpp . In case of correct input, we have printed the detailed output after every instruction and in the end we printed how many times each of the instructions were executed. While for incorrect expressions, we terminated the program with a appropriate error message which would be helpful to the user for debugging. The example test cases are present in the submission folder.

   We have also tested out input codes with unsafe lw/sw instructions, forwarding and redundant memory instructions, rejection, etc.

2. Error handling:- We have thrown error at required places to the best of our MIPS syntax knowledge. Almost the entire exception handling is same as that in assignment 3. In addition to all the previous error messages, we have added another error message which is raised when the waiting list has no space.

   Hence all the test cases have helped us to ensure that our assignment4.cpp is correct. Getting correct outputs on some test cases can never be the right criteria to judge if an algorithm is correct or not but extensive checking can still ensure us that we have done the things correctly.

4.4 Features of the simulator

4.4.1 Strengths

(a) This model of the simulator is successful in the primary objective which is to reduce the number of clock cycles by re-ordering the DRAM operations without altering the sequence of other instructions.

(b) It also correctly handles the unsafe instructions by first completing the dependant operations from the waiting list.

(c) Another way to reduce the number of clock cycles is to skip the execution of redundant instructions by forwarding, which is also supported by our simulator. In forwarding what we have done is if we have updated stored some register value in the Memory and next time we want to load value from that memory, then we will not issue another DRAM request, we will directly execute the lw operation.

(d) We have tried to keep our implementation close to the actual hardware implementation by separating the execution of the processor and the DRAM memory, which allows for parallel execution of normal and memory-related operations. This is the reason why we keep two different clock cycle counters to synchronise both these parts.

(e) We have also made our code more space-efficient by constraining the maximum size of waitingList to 64 instructions.

4.4.2 Weaknesses

(a) While implementing this algorithm, we have taken some assumptions such as:

   i. Even if the execution seems to be parallel, the actual order of execution is sequential, i.e. an lw instruction and an independent add/addi operation are not actually taking place in parallel, but one after the another. This is why we are actually pre-computing the values of ending cycle of every lw/sw operation. So, we are not executing the

operations in parallel but we are outputting the clock cycle values as if there was parallel execution.

ii. We have assumed that the processor has a mechanism to know that it has already updated a register in the processor or not. If it has already been updated, then if the DRAM tries to modify that register, the processor will reject it.

iii. We have not ignored any sw instruction, because there might be some cases where an lw instruction needs to access a value from that address. In this case it will be an unsafe instruction. So instead we have considered it an safe in the processor and have implemented all the READ and WRITE operations from the same address in an order in the DRAM.