COL216 Assignment 4

Prakhar Jagwani (2019CS10382) and Rayyan Shahid (2019CS10392)

April 2021

1 Approach

1.1 Strategy for ordering DRAM Requests

The following sections describe the approach followed for handling the DRAM requests. The strategies followed for storing, processing and efficient ordering of DRAM requests are discussed.

1.1.1 Storing and processing DRAM requests

An object of class `ScheduleIns` has been used to schedule the various instructions and DRAM requests. The pending DRAM requests are stored in `dram_requests` which is a vector of deques. An object of class `Dram` acts as the DRAM for our architecture and has various methods to process the different DRAM requests. It also maintains a field `cycle` which is used by the `ScheduleIns` object for scheduling the requests.

1.1.2 Efficient ordering of DRAM requests

`dram_requests` maintains a `deque` for each row of the memory. If the current `lw/sw` instruction being processed is considered safe, a DRAM request is issued and the request is pushed in the corresponding `deque` in `dram_requests`. The `ScheduleIns` also maintains a field `currDRAMRequest` which keeps track of the DRAM request being processed currently.

The method `nextDRAMRequest` updates the next DRAM request to be processed. If the deque currently being processed is non-empty, the next dram request is the request at the front of the deque. Else, we find a DRAM request, which makes the current instruction unsafe, and start processing the deque corresponding to this request. If no such request exists (i.e instruction in safe), we choose the first non-empty deque.

At every instruction, we call the `cycleUpdate` method of `ScheduleIns` to update the `cycle` at which the instruction is to be processed. This is done by processing the stored DRAM requests until the current instruction is considered safe. The dependencies between the DRAM requests and the registers are maintained using `isBusyRegStore`, `isBusyRegWrite` and `RegValChanged`.

These are updated whenever a request is pushed in `dram_requests` or when it is completed. These dependencies are used to determine if an instruction is safe or unsafe.

1.2 Strengths and Weaknesses of the approach

1.2.1 Strengths

- In case of consecutive memory access, we are separating the instructions into queues for each row. So, we process all the instructions in a queue, before moving on to the next queue. This will reduce the number of row write-backs and improve efficiency. Test 1 shows the difference in efficiency between reordering and no reordering approaches.

- To select the next row to process on, we find the row with the memory operation using the dependent register of the next instruction. We start processing this row first, so that the next instruction becomes safe to execute as soon as possible.

```
#access delays are 1
addi $s0, $zero, 10
sw $s0, 1000
lw $s1, 2000
lw $s2, 3000
addi $s2, $zero, 10

#Optimal choice: 8
#Non-optimal choice: 11
```

  Here, processing the `lw $s2` after `sw $s0`, was the optimal choice.

- In `sw` and `lw`, there are registers being read from. We consider that the current value of the register gets loaded into the data bus and is free to be modified. But there may be subsequent `sw` and `lw` instructions using the same registers and because of reordering, these may get executed first. This may lead to problems if there were any instructions in between two such instructions which modified the same register. So, we have used `RegValChanged` to check if there were any modifications to the value of a register being read from, between two memory operations involving reading from that register. If there were none, it is safe to push the second DRAM request, else it is not. This reduces the number of cycles if there were no modifications.

- While a DRAM request is processing, subsequent `sw` and `lw` can be read and pushed into queue, this saves a cycle, even if there is no reordering.

1.2.2 Weaknesses

- The re-ordering of DRAM requests is done only among the requests belonging to different rows of the memory. The order of processing of DRAM requests corresponding to the particular row of memory remains unchanged. This re-ordering may result in smaller wait times for some unsafe instructions which is absent in our approach.

```
#access delays are 1
addi $s0, $zero, 1000
lw $s2, 1024
lw $s0, 1020        #2
lw $s1, 1000
addi $s0, $s0, 4
lw $s2, 1024
#Cycles : 12

addi $s0, $zero, 1000
lw $s2, 1024
lw $s1, 1000
lw $s0, 1020        #3
addi $s0, $s0, 4
lw $s2, 1024
#Cycles : 13
```

- While finding the next DRAM request to be processed, if the deque currently being processed is empty, we are searching for the next optimal request to be processed. Although this reduces the wait time for the current unsafe instruction, however, searching for the request may take additional clock cycles which might reduce the efficiency of the architecture. This has to taken into account during the realization of the above approach in a real life model.

- A separate deque is being maintained for storing DRAM requests corresponding to different rows of the memory. Although this allows for efficient grouping of dependent and independent DRAM requests, however, it also requires additional memory for storing the deque structures. Therefore, this approach is not suitable in cases where memory is the major constraint.

2 Testing

The following test cases have been considered

1. Consecutive memory access

```
sw $s0, 1000
```

```
lw $s1, 1024
sw $s0, 1000
#Reordering
    Cycles : 37
    Row write-backs : 1
#No Reordering
    Cycles : 59
    Row write-backs : 2
```

2. Modifications between memory access

```
addi $s0, $zero, 1000
addi $s1, $zero, 1024
sw $s2, ($s0)
add $s0, $s0, $s0
lw $s2, ($s0)
#Reordering: 38
#No Reordering: 38
```

3. No modifications between memory access

```
addi $s0, $zero, 1000
addi $s1, $zero, 1024
sw $s2, ($s0)
add $s1, $s1, $s1
lw $s2, ($s0)
#Reordering: 17
#No Reordering: 18
```

4. Empty request queue

```
#access delays are 1
addi $s3, $zero, 1
addi $s4, $zero, 2
sw $s3, 1000
add $s4, $s4, $s3
add $s4, $s4, $s3
add $s4, $s4, $s3
sw $s6, 1028
#Cycles: 10
```