COL216: Assignment 5

Divyanka Chaudhari (2019CS50429)
Pranay Gupta (2019CS10383)

Idea

We implement multi-core functionality over the previous single-core version where we implemented re-ordering of DRAM instructions so that there are the least amount of row/buffer updates. This is done after taking care of the dependencies that arise during looking ahead in the code. Since we have only one DRAM we optimize the use of sw/lw by minimizing the total number of cycles while also not letting any file program starve. The dependencies are seen if there are instructions that make use of registers in which value is being loaded or if we try to access a register that has a dependency from an instruction. Memory locations accessed are also taken care of such that if different programs try to access same memory location virtually they're allocated different physical memories to avoid any clash.

Implementation

We use our main helper function called `process` in which we send every command to process. This time, we process the main loop in the program in a cycle-wise manner instead of going line by line earlier. This is because we need to run parallel files.

`saveCycles` looks for the `sw` and `lw` instructions to keep track of their number of cycles while non-blocking other commands. We also incorporate re-ordering between commands.

We use a `memoryManager` to re-order the DRAM requests which has a maximum size of 16.

When we encounter `sw` or `lw` in our main while loop, it jumps into the functions above. If we encounter any non-DRAM instruction, we keep a track of the memory and register used by it in different vectors. We then implement a check in the main while loop to execute them safely after our safe DRAM instructions are encountered. At any point while moving through instructions if a non-safe/busy register/memory is encountered in some file, we break and wait for that particular file.

Delay Estimation

For the memory manager, we compare the incoming request with all the other elements already present in the buffer. Under the assumption that one comparison takes one cycle, the extra time/delay the memory manager would have would be the amortized time it'd take for the total number of comparisons. Since our buffer size is 16, this comes out to be around 4 cycles. It'll not take this time every time, but that's the average wait time it'd take. We have incorporated this delay into our program.

Advantages & Disadvantages

1. Since we're comparing the current element with all elements, then every time there's a delay that's inevitable.
2. The advantage of the above is that if there are consecutive accesses from different memory loads then we do not have to load it multiple times which is an advantage.
3. We're forwarding/removing redundant lw/sw so it makes the program faster.
4. We have a check-in place to avoid any file's program starvation.
5. If we see that any command is going to block that file, we execute that command first even if it's not on the top priority according to the default memory manager so that we free it up while other commands in other files run.

Test cases

All the input test cases are present in the folder `testcases` and their corresponding outputs are present in the folder `testcaseoutputs`.

1. Testcase 1: Basic implementation of the code that contains a few `sw` and `lw` commands along with a few commands in between.
2. Testcase 2: Accessing same memory location from different files.
3. Testcase 3: Maximising the wait buffer and looking if it starves.
4. Testcase 4: Checking if it executes blocking program first so that it again doesn't starve.