COL216 A-5 Write-up

Sayam Sethi 2019CS10399
Mallika Prabhakar 2019CS50440

April 2021

1 Assumptions

The following hardware assumptions were made in the design of the **MIPS Processor**:

1. There are **three ports** for communication with the processor:
   (a) Two **output ports** to send the DRAM request to the **DRAM queue** (explained in detail in the next point) and signify if there is a "priority load" required by the processor if it is being stalled, respectively.
   (b) An **input port** to receive, and accept or reject the result of the DRAM request after delay.
   (c) A **pair of {address, value}** for every register which stores the value of the latest DRAM address from which the value is to be loaded from.
   (d) The remaining hardware is the same as for a single pipeline MIPS processor.
2. The **Memory Request Manager (MRM)** has the following hardware components and ports:
   (a) N **buffer queues** (which is a fixed sized data segment, $size = 32$) for every core which stores the instructions send to the **MRM**. Detailed hardware structure is explained in the next section. A **counter** is also used to keep a track of the number of elements present in the segment.
   (b) A sw-lw forwarding **pipeline register** which stores the latest value of **{issue clock cycle, value}** for the address(es) which are currently present in the **buffer queue**.
   (c) Two **cyclic priority encoders** which helps decide the core whose instruction is to be processed next.

(d) $2 \times N$ **input ports** which receive the DRAM requests from the cores and the priority signals from the cores.

(e) N **output ports** which return the DRAM load results to the cores.

(f) Two more **counters** which keep a track of the number of instructions processed in the current row and cycles remaining for the current request to be processed.

(g) Two **registers** which store the current core and row whose request is being serviced.

3. The **DRAM architecture** is as follows:

(a) It contains a **2D square array** of 1024 rows with each row having 1024 bytes (256 words). This memory segment is divided into N blocks to separate the memory locations accessible by the cores.

(b) It contains another storage location called the **row buffer** which is used to store the current memory retrieval row.

Note: N is taken to be 16 in our implementation, however the value can be changed to any number that is a factor of 1024.

2 Implementation

2.1 Algorithm

Note: The hardware specifications and decision delays are discussed in the next subsection

The idea behind our implementation is to ensure that all cores are serviced as soon as possible and no core is left waiting for too long. The **MRM** has separate buffers for every core, which has a maximum size of 32 requests each. The buffer is implemented as an *unordered map* of *queue* which helps service the requests sequentially for every row.

When the requests are sent to the **MRM** for the first time (or after the **MRM** was emptied), it selects the first core which has a request that has been sent to the buffer. In the subsequent selections, it selects requests from the same row (and hence the same core), if requests are available.

Now, if all the requests of the current row are processed or $maxToProcess$ ($maxToProcess = \frac{rowAccessDelay}{colAccessDelay}$) number of instructions have been serviced from the current row, the **MRM** decides which core, row to service next. This is determined by cyclically selecting the next core which has a priority load, if any. If no core has a priority load, then the next core with a pending request is cyclically selected. If the core was selected using priority, $maxToProcess$ ensures that no core is left **starving** and cyclic selection ensures that no core gets more priority when deciding which request to serve.

Other Optimisations:

- **Forwarding:** sw to lw forwarding is implemented using a register file (pipelining registers) present in the **MRM**
- **Skipping Redundant Instructions:** If there are multiple lw instructions for the same register, only the most recent request is sent to the DRAM and the others are skipped. Similar implementation is done for multiple sw instructions.
- **Dirty-only Write-back:** Write-back only when the buffer is dirty, else no write-back.

2.2 Hardware Perspective

The delays have been evaluated under the following assumptions:

1. The processor sends requests to the **MRM** before the falling edge of the current clock cycle (so that the information is available during the next clock edge).
2. The clock cycle is large enough that about 5 sequential operations can be performed in a single clock cycle. This is necessary since the processor performs IF, ID, Registers, ALU control, ALU.

Buffer Queue

The buffer queue is a data segment which has a maximum size of 32 for every core. The requests sent by the processor is "pushed" to this queue. It has a similar to linked list implementation, where the position of the next element of the queue is stored in the current element. When creating a new queue, an algorithm similar to that used for hashing is used. This on an average is $O(1)$ operation and will complete in one clock cycle. Thus when the **MRM** selects the next instruction, it happens towards the end of the clock cycle and hence takes one complete cycle to select the next instruction to execute (when in the same row).

When priority encoder is involved to select the next core, another clock cycle is used to obtain the result from the encoders and then the optimal row is selected from the core of the output of the encoder.

Forwarding Pipeline Register

It is similar to the pipeline register taught in the lectures and stores the {issue clock cycle, value} for the addresses that have pending sw requests to be processed by the DRAM. The mapping is similar to that which is used by a cache. The result is returned one cycle after the request is sent to the **MRM** in which the **MRM** determines the forwarding value.

Cyclic Priority Encoders

If the core whose request to process has to be changed, the values from the cyclic priority encoders is read. It is different from a priority encoder in the sense that the priority changes cyclically with the core just after the current core having the largest priority. This uses a combinational circuit and hence returns the result with a delay comparable to only delays of logic gates.

3 Strengths and Weaknesses

3.1 Strengths

- Forwarding helps save a lot of cycles especially when there more pending instructions, this helps in ensuring that stalling does not cascade (if any) and the pending instructions do not keep on increasing.
- Every redundant instruction is not processed and this increased the instructions per cycle greatly. In the case when a redundant instruction was being processed by the DRAM, it is rejected by the processor (this saves cycles too by not stalling for the request to finish processing).
- Writeback happens only when the buffer was dirty, this saves clock cycles in a lot of cases and decreases the time taken to switch rows by 50%
- Starvation handling was done which helped improved the performance of the **MRM** for large number of cores and a large number of cycles. As the total number of instructions per core are increased, the eficiency of the manager improves.
- The operations of the decision making of the **MRM** happen independently of the core execution, which avoids stalling of processor instructions. This also ensures that on an asymptotic scale, the $rowAccessDelay$ is increased by a factor of 1.5, along with the number of instructions processed per row switch becomes $5\times$.
- Every core is allocated an individual **buffer queue** which ensures that the requests can be pushed independently of other cores and the pending requests in other cores doesn't limit the number of request that can be sent by a particular core.

3.2 Weaknesses

- To ensure that all cores have equal priorities and minimal effects of other cores, a lot of additional memory is needed in the **MRM**. Thus, it has a slightly larger component cost, proportional to N.

- Since the data structure for the DRAM request is somewhat complicated, the pushing of instructions is relatively slow as compared to other processes being executed in the **MRM**. This is the limiting process for the **MRM** delays.
- Our implementation provides sufficient "bubble time" to ensure that the **MRM** instructions execute within a single clock cycle. However, more instructions could have been squeezed in the MRM in a single clock cycle by decreasing the "bubble time".

4 Test Cases:

1. **Test1 -** Initial test case to test multi-core functionality
2. **Test2 -** Test case to test execution when row changes are needed (no dependent instructions)
3. **Test3 -** Tests execution when instructions are skipped + stopping a single core on error
4. **Test4 -** Contains primarily store expressions in the files
5. **Test5 -** Each core respectively: dependent loads, no DRAM, forwarding (reduction of cycles from about 450 cycles to 271 cycles compared to Assignment 4 implementation)
6. **Test6 -** Unsafe instructions in all cores (forwarding happens in some of the cores)
7. **Test7 -** Collection of files provided in Assignment 4 demonstration to be run in a parallel manner
8. **Test8 -** Random test case 1 (random)
9. **Test9 -** Random test case 2 (negative address)
10. **Test10 -** Random test case 3 (normal files and erroneous files run in parallel)