DRAM Request Manager for Multicore Processors

Aditya Goel — 2019CS10671

May 17, 2021

Program Design Specifications: Input & Output Format

- The C++ Program can be compiled using the following command line instruction
  g++ -o main main.cpp
- The executable can then be run by the following command line instruction
  ./main ROW_ACCESS_DELAY COL_ACCESS_DELAY NO_OF_CYCLES NO_OF_CORES INPUT_FILE_NAME$_1$ . . . INPUT_FILE_NAME$_N$
- ROW_ACCESS_DELAY and COL_ACCESS_DELAY are the delays in number of cycles for accessing row and column from the DRAM respectively.
  - An "Error: ROW_ACCESS_DELAY not specified correctly" or "Error: COL_ACCESS_DELAY not specified correctly" is printed if the ROW_ACCESS_DELAY or COL_ACCESS_DELAY are not non-negative integers respectively.
- INPUT_FILE_NAME is the name of the MIPS assembly language program (as text file, NOT machine instructions).
  - An "Error: File does not exist or could not be opened" is printed if the file doesn't exists or is not a text file.
- At every clock cycle the program prints the cycle number, the address of completed instruction and any modified registers, any modified memory locations or any activity on DRAM.
- After execution completes, the program prints relevant statistics such as the Total execution time in clock cycles, Number of requests sent to the Memory Manager, Number of requests pending in the Memory Manager and the Number of row buffer updates. These statistics are printed for each core as well as a cumulative account of all the cores.

Strategy & Approach Used

- Each instruction in the input file is stored in a vector line by line, for each core. Any instance of a comment is ignored.
- As we iterate through the instructions one by one in a sequential manner, each itertion of an instruction involves processes of parsing the instruction and performing relevant operations on concerned registers of the core or memory locations.
- Appropriate errors are raised during parsing each line, if any instruction other than add, addi, sub, mul, slt, j, beq, bne, lw or sw is encountered or if these instructions are not found in the format accepted as MIPS assembly language.
- The DRAM is implemented as a 2-Dimensional array of size 1024 x 1024 bytes. A single Read/Write in a lw/sw instruction corresponds to accessing a 32-bit (4 byte) value from the DRAM. Within a row, a given piece of data is located at a column offset, so a memory address could be thought of as consisting of a ROW ADDRESS and COLUMN ADDRESS.
- ROW_BUFFER is implemented as a 1-Dimensional array of size 1024 bytes which helps in both accessing the value stored at a particular memory address or store and update any memory address by copying the necessary row from the DRAM and writing it back to DRAM whenever required.

- The Memory Request Manager as an ordered queue has been implemented using an 1-Dimensional array of finite size, such that it can accomodate a maximimum of 64 requests at any time. It supports for re-ordering of DRAM reuests, without causing any change to the program's semantics. Though adding Requests to the Memory Request Manager in an ordered manner asks for a certain delay but saves a lot on cycles corresponding to delays involved in accessing the DRAM. More about reordering of DRAM requests and benefits of reordering has been mentioned in the strengths & weakness section.

Strengths, Weaknesses & Optimisation Methods

- The Memory Request Manager implements reordering of DRAM requests, wherein DRAM requests associated to lw/sw instructions can be reordered in non-blocking memory access fashion unless we find another instruction involving the register as associated with previous lw/sw instructions.
- The reordering of DRAM requests is done in a stable manner, such that DRAM requests associated to any given memory address are executed in the same order as given in the MIPS assembly language code.
- Reordering of DRAM requests helps improve the Total Execution Time in clock cycles of the program as we cut on significant clock cycles otherwise required for row access delay and column access delay.
- Our implementation of Non-Blocking Memory Access, allows for execution of instructions even if a lw/sw instruction has not completed, only until we do not encounter any instruction that involves any register that was mentioned in the corresponding lw/sw instruction or a bne/beq/j instruction, or until we execute instructions one less than the delay in number of cycles that the corresponding lw/sw instruction had to face. All such instructions which cannot be executed when the execution of corresponding lw/sw instruction has not completed are said to be unsafe for execution.
- The Non-Blocking Memory Access further considerably increase in the Throughput Efficiency, as it allows for execution of instructions in cycles being otherwise utilised only for ROW_ACCESS_DELAY and COL_ACCESS_DELAY. The effect increases in case we have a large value of ROW_ACCESS_DELAY and COL_ACCESS_DELAY.
- A further increase in the Throughput Efficiency is brought by eliminating execution of any lw /sw instruction that aims to change the value of a register or memory, which is being changed by any other following lw/sw instruction in the program, provided no instruction in between these two needs for value in that register or memory.
- To further increase the Throughput Efficiency, forwarding has been implemented. For example instructions like sw $t0, 0($s0) may take several cycles to write on the DRAM Memory, while the value of register $t0, is forwarded for future use without having to wait for cycles that involve a row access delay and column access delay.

Estimation Of Delay in Clock Cycles of Memory Request Manager

- The Memory Request Manager as an ordered queue has been implemented using an 1-Dimensional array of finite size, such that it can accommodate a maximum of 64 requests at any time.
- We insert requests in the Memory Request Manager in an order, such that minimum cycles are involved in delay for accessing the DRAM. If there are no requests pending in the Memory request manager it takes us 1 cycle to add the request to the memory manager. While, if there are certain number of requests pending in the Memory Request Manager, we utilise 1 cycle to copy the request in a auxiliary block of memory , available as a part of the Memory Request Manager, and then of these pending requests, we need to make comparisons one greater than the number of requests which have a priority lower than the request being added, starting from the one with the lowest priority. Each request with a priority lower than the request being added is moved down, and utilising an additional cycle the new request is added to the ordered queue.
- Hence adding any new request to the Memory Request Manager involves a delay equal to three more than the the number of pending requests with priority lower than the request being added. Further issuing the request for the DRAM access takes only 1 cycle, since it is always the case that we need to make a DRAM request corresponding to instruction at the first place of the Ordered Queue generated by the Memory Request Manager. Hence, This accounts for all the additional delay that the Memory Request Manager

introduces in the execution of the program over a single core program execution without re-ordering that involves no Memory Request Manager.

Test Cases & Corner Cases Handled

1. In case of a comment both, as a separate line or along with those Instructions are completely ignored by the program.

   – A test on input
   #My New Increment Program
   addi $t0, $t0, 1 #Incrementing $t0

   resulted in a successful execution of the program and incremented the value of $t0 by 1 , ignoring any comments that came in between.

   – A test on input
   #My New Increment Program
   #addi $t0, $t0, 1 Incrementing $t0

   resulted in a successful execution of the program with no Instructions to be evaluated.

2. Entering any invalid instruction, other than those specified in the program design specifications prints "Error: Invalid operation" error.

   – A test on input
   move $t0, $t1

   resulted in an "Error: Invalid operation" error.

3. If results of instructions such as add, sub or addi results in an arithmetic overflow, corresponding errors are raised.

   – A test on input
   addi $t0, $t0, 1
   addi $t0, $t0, 2147483647

   resulted in an "Addition Overflow" error.

4. If at any stage of program, the number of instructions executed is found to exceed 131,072, the execution stops, and print "Number of Instructions exceeded the maximum allowed values" error, for that particular core.

5. If the input to commands such as j, bne, beq is a label which is not present in the Source Code, a "Label Not Found" error is raised.

   – A test on input
   label1:
   addi $t0, $t0, 1
   bne $t0, $zero, label2

   resulted in an "Label Not Found" error.

6. Entering a memory address beyond the allowed bounds of (0-1048576) or a memory address that is not a multiple of 4, prints "Invalid Memory address" error.

   – A test on input
   addi $s0, $zero, 1048580
   sw $t0, 0($s0)

   resulted in an "Invalid Memory address" error.

7. Entering an input with register names other than the 32 registers specified, prints an "Error: Invalid register" error. Also entering non-32 bit integer values in instructions such as addi, prints an "Error: Given value is not a valid number" or "Error: Out of Range Integer" error respectively.

   – A test on input
   addi $m0, $m0, 1

   resulted in an "Error: Invalid register" error.

- A test on input
  addi $t0, $t0, abc

  resulted in an "Error: Given value is not a valid number" error.
- A test on input
  addi $t0, $t0, 2147483648

  resulted in an "Error: Out of Range Integer" error.

8. Our Implementation can parallely execute programs equal to the number of cores available, and any mismatch in the number results in an "No. of cores not equal to no. of files" error.

9. The Program Only allows for execution of programs on a finite number of cores, i.e. 64 in number. An input that demands for a greater than 64 cores raises an error "No. Of Cores Exceed Maximum Allowed Value Of 64."

10. No two writes are performed on a particular register file, in one cycle.

    - A test on input
      main:
      addi $s0, $zero, 1000
      lw $t0, 0($s0)
      add $t1, $t1, $t0
      exit:

      The value of register $s0, $t0 & $t1 are written in distinct cycles, and not the same cycle.

11. Since the Memory Request Manger is of a finite size and can accomodate a maximum of 64 requests at a time, the program gracefuly adds a new request to the memory manager only when there is space created on the Memory Request Manger, i.e. only when a request is completely processed.

12. Reordering of DRAM request corresponding to lw/sw instruction resulted in a correct output, i.e. similar to the case of not reordering the DRAM requests. It reduced the Total Execution Time Considerably as opposed to the case of DRAM requests not being reordered.

    - A test on input
      main:
      addi $s0, $zero, 1000
      addi $s1, $zero, 2500
      addi $t0, $zero, 1
      addi $t1, $zero, 2
      addi $t2, $zero, 3
      addi $t3, $zero, 4
      sw $t0, 0($s0) store 1 at location 1000
      sw $t1, 0($s1) store 2 at location 2500
      sw $t2, 4($s0) store 3 at location 1004
      sw $t3, 4($s1) store 4 at location 2504
      lw $t5, 0($s0)
      lw $t6, 0($s1)
      lw $t7, 4($s0)
      lw $t8, 4($s1)
      exit:

      The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, with reordering of DRAM requests took 61 cycles as the Total Execution Time, while an execution without reordering of DRAM requests took 180 cycles as the Total Execution Time.

13. Reordering of DRAM request corresponding to lw/sw instruction was done in a stable manner, i.e., instructions corresponding to a given memory address is executed in the same relative order as written in MIPS assembly language program.

    - A test on input
      main:
      addi $s0, $zero, 1000

addi $s1, $zero, 2500
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
addi $t4, $zero, 5
sw $t0, 0($s0) store 1 at location 1000
sw $t1, 0($s1) store 2 at location 2500
sw $t2, 4($s0) store 3 at location 1004
sw $t3, 4($s1) store 4 at location 2504
sw $t4, 0($s0) store 5 at location 1000
lw $t5, 0($s0)
lw $t6, 0($s1)
lw $t7, 4($s0)
lw $t8, 4($s1)
exit:

The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, with reordering of DRAM requests executed instructions corresponding to a memory address in order of their appearance in the MIPS assembly language program. For example: Instructions sw $t0, 0($s0), sw $t4, 0($s0) & lw $t5, 0($s0) all corresponding to memory address 1000, are executed only one after the another, respectively, such that a register $t5 contains a value of 5 and not 1.

14. Multicore implementation with a single Memory Request Manager resulted in correct results, and increased the throughput efficiency as compared to what a single core implementation would have, had the programs been executed sequentially one after the other.

- A test on input

main1:
addi $s0, $zero, 1000
addi $s1, $zero, 2500
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
addi $t4, $zero, 5
sw $t0, 0($s0) store 1 at location 1000
sw $t1, 0($s1) store 2 at location 2500
sw $t2, 4($s0) store 3 at location 1004
sw $t3, 4($s1) store 4 at location 2504
lw $t5, 0($s0)
lw $t6, 0($s1)
lw $t7, 4($s0)
lw $t8, 4($s1)
exit1:

main2:
addi $s0, $zero, 4000
addi $s1, $zero, 5500
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
add $t0, $t2, $t2
add $t1, $t3, $t3
sw $t0, 0($s0) store 1 at location 4000
sw $t1, 0($s1) store 2 at location 5500
lw $t4, 0($s0)
lw $t5, 0($s1)
exit2:
main3:

addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
addi $t4, $zero, 5
exit3:

Since we have a separate register file for each core instructions in Program 1 result in changes in register file corresponding to the same core, and has no effect on the register file of the other cores. Memory Requests from all the cores (here, core 1 and 2) are added to the Memory Request Manager which helps in accessing the DRAM one request at a time.

15. The Non-blocking Memory Access Allowed for execution of safe instructions in cycles where the ROW_ACCESS_DELAY & COL_ACCESS_DELAY number of cycles are required for complete execution of the lw/sw instructions.

    – A test on input
    main:
    addi $s0, $zero, 1000
    addi $s1, $zero, 2500
    addi $t0, $zero, 1
    addi $t1, $zero, 2
    addi $t2, $zero, 3
    addi $t3, $zero, 4
    sw $t0, 0($s0) store 1 at location 1000
    sw $t1, 0($s1) store 2 at location 2500
    sw $t2, 4($s0) store 3 at location 1004
    sw $t3, 4($s1) store 4 at location 2504
    addi $t0, $zero, 10
    addi $t1, $zero, 20
    addi $t2, $zero, 30
    addi $t3, $zero, 40
    lw $t5, 0($s0)
    lw $t6, 0($s1)
    lw $t7, 4($s0)
    lw $t8, 4($s1)
    exit:

    The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, reorders the DRAM requests along with non-blocking memory access which allows for execution of the multiple addi instructions in a safe manner. For example: The value stored at an address of 2500 is 2 and not 20, i.e. the instruction addi $t1, $zero, 20 which is after the instruction sw $t1, 0($s1) has no effect on the value being stored at the memory address of 2500.

16. In case of lw instructions that aim to write a value to a particular register, is such that the particular value of the register is not being used by any other lw instruction before its value is changed to something else through some second instruction, the complete execution of the first instruction doesn' t take place and only the later is executed.

    – A test on input
    main:
    lw $t0, 0($s0)
    lw $t0, 4($s0)
    exit:

    The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, only the second instruction is completely executed as the first instruction is identified as being reduntant.

17. In case of lw instructions that aim to write a value to a particular register, is such that the particular value of the register is being used by any other lw instruction before its value is changed to something else through some second instruction, the execution of the first instruction also becomes necessary for a correct result.

- A test on input

```
main:
lw $t0, 0($s0)
add $t2, $t0, $t1
sw $t2, 4($s0)
lw $t0, 4($s0)
add $t4, $t0, $t3
exit:
```

  The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, executed both the instructions lw and addi as the value of register $t0 is being utilised in instructions between them. Such that the value stored in register $t2 = 0, and in $t4 = 1.

18. In case of sw instructions that aim to store a value at a particular location in the memory, is such that the value stored at this memory address is not being used by any other instruction before the value stored at this address is changed to something else through some second sw instruction, the complete execution of the first instruction doesn' t take place and only the later is executed.

- A test on input

```
main:
sw $t0, 0($s0)
sw $t1, 0($s0)
exit:
```

  The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, only the second instruction is completely executed as the first instruction is identified as being reduntant.

19. In case of sw instructions that aim to store a value at a particular location in the memory,is such that the value stored at this memory address is being used by any other instruction before the value stored at this address is changed to something else through some second sw instruction, the complete execution of the first instruction doesn' t take place and only the later is executed.

- A test on input

```
main:
sw $t0, 0($s0)
lw $t2, 0($s0)
addi $t1, $t1, 1 sw $t1, 0($s0)
lw $t3, 0($s0)
exit:
```

  The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, executed both the sw instructions as the value stored at the memory location 0-3 is being utilised in instructions between them. Such that the value stored in register $t2 = 0, and in $t3 = 1.

20. In case of we encounter errors in the program being executed by a particular core, only that core stops the execution while all other cores keeps on the execution of their respective programs. That is, error in program corresponding to 1 core doesn' t stop the execution of programs on the other core.

- A test on input

```
main1:
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
exit1:

main2:
addi $t0,$t0, 1
addi $t1,$t0, 2147483647
exit2:

main3:
addi $t0, $zero, 1
addi $t1, $zero, 2
```

addi $t2, $zero, 3
addi $t3, $zero, 4
exit3:

The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, and each program fed to a different core executed all the instructions corresponding to core 1 and core 3, while raised an appropriate "Addition Overflow" error corresponding to core 2.

21. If the Program is allowed with a lesser number of cycles, than it would require for complete execution of all the instructions, the Program finally prints the number of requests that were yet pending in the Memory Manager corresponding to requests generated by each core.

- A test on input

main1:
addi $s0, $zero, 1000
addi $s1, $zero, 2500
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
addi $t4, $zero, 5
sw $t0, 0($s0) store 1 at location 1000
sw $t1, 0($s1) store 2 at location 2500
sw $t2, 4($s0) store 3 at location 1004
sw $t3, 4($s1) store 4 at location 2504
lw $t5, 0($s0)
lw $t6, 0($s1)
lw $t7, 4($s0)
lw $t8, 4($s1)
exit2:

main2:
addi $s0, $zero, 4000
addi $s1, $zero, 5500
addi $t0, $zero, 1
addi $t1, $zero, 2
addi $t2, $zero, 3
addi $t3, $zero, 4
add $t0, $t2, $t2
add $t1, $t3, $t3
sw $t0, 0($s0) store 1 at location 4000
sw $t1, 0($s1) store 2 at location 5500
lw $t4, 0($s0)
lw $t5, 0($s1)
exit2:

The Program with ROW_ACCESS_DELAY = 10 and COL_ACCESS_DELAY = 2, which took 113 cycles for complete execution of both sets of instructions, was when allowed only 58 cycles, the Program 1 could queue 8 requests to the Memory Requests Manager, of which complete execution of 1 of the requests was pending, while the Program 2 could make 4 requests to the Memory Requests Manager, of which complete execution of 4 requests were pending at the end, as is reported by the Program.