INDIAN INSTITUTE OF TECHNOLOGY DELHI

COL216

Report

Assignment – 4

Memory Request Ordering

Abstract

A simulator is a software that emulates the actions of an entity without actually utilising the entity. Here we attempt to create a cross platform MIPS simulator that emulates all the hardware instructions supported by MIPS. This simulator takes as input a MIPS assembly program that translates it into instructions executed by MIPS.

Harsh Agrawal	2019CS10431
Saptarshi Dasgupta	2019CS50447

Contents

1 Modules **2**
- 1.1 Compiler . . . 2
- 1.2 Hardware . . . 2
- 1.3 DRAM . . . 2
- 1.4 DramDriver . . . 3

2 Approach **3**
- 2.1 Advantages . . . 4
- 2.2 Disadvantages . . . 4

3 Testing **5**
- 3.1 Result . . . 6

4 Assumptions **6**

1 Modules

We implemented a simulator in `C++` for executing MIPS instructions from assembly code. The program was divided into `compiler` ,`hardware`, `DRAM` and a `DramDriver` module. These modules perform different functions as outlined below

1.1 Compiler

- This module takes as input the assembly program and parses the tokens according to the syntax specifications of MIPS assembly programs.
- This module assembles the program into a sequence of instructions that can be executed according to the hardware specifications.
- It also generates appropriate syntax errors if an erreneous expression or an unidentified token is encountered.
- The compiler checks the encodibility of instructions into 4 bytes (without actually encoding it) using the size of operands involved.

1.2 Hardware

- This module emulates a subset of instructions provided by MIPS. They are `add, mul, sub, slt, addi, bne, beq, j, lw, sw`.
- The module maintains a record of all the register and memory values and modifies them according to the instruction specifications.
- It generates appropriate exceptions on performing prohibited actions like out of bounds memory access and reserved register access.

1.3 DRAM

- This module implements the DRAM model in the form of a `2D` array.
- It contains a row buffer to simulate an actual `DRAM`. A row is first copied into this row buffer and the module uses the buffer as a write-back cache for further read and write operations.

1.4 DramDriver

- A `DramDriver` module is implemented to capture all `DRAM` requests and order them in a queue according to some pre determined heuristics.
- The driver recieves the requests from hardware and issues them to `DRAM` in a particular order to maximize clock cycle utilization.
- The hardware can also send a blocking call to the driver instructing it to execute all requests until a certain condition is met.

2 Approach

The approach used to simulate the DRAM memory reordering is described in this section

- The `DramDriver` orders the memory requests to minimize the number of row buffer reloads required. Our approach orders the requests of the same row together and another row is chosen only when there is no more request of a given row.
- To ensure correctness we have not changed the order of requests of the same address. This ensures that requests of a particular address are always issued in the same order as they were executed by the processor.
- The driver also supports blocking on a particular register value. This blocks the processor and instructs the driver to complete all requests pending for the blocking register.
- The DRAM requests are issued asynchronously and the processor always executes in non blocking mode, since reordering of requests is not possible in blocking mode.
- An optimization was introduced that allowed the driver to overwrite multiple requests for the same address. For example if there are two consecutive write requests to a particular memory address then the first request is deleted provided the same memory address was not used to load another register in between.
- Also, when two consecutive read requests are issued for the same register consecutively, the first request is always deleted.

- Another optimization is introduced which selects the next batch of requests after all requests on a row are exhausted depending on the blocking registers involved in each batch. The batch with a blocking register is preferred and executed first. This is illustrated in the testcase `-tests/input/choosing_row.txt`

- In addition to choosing optimal batches of rows to process, we also choose the requests to execute with the same row but with different addresses.The requests with a blocking register is preferred This is illustrated in the testcase - `tests/input/choosing_address_in_row.txt`

2.1 Advantages

- The approach ensures that the number of row buffer reloads required is minimized, to reduce the overhead of writing back and loading the row buffer.

- The approach always ensures it is not slower than no reordering. We have also maintained correctness by preventing reordering in cases where reordering can yield incorrect results.

- The asynchronous nature of DRAM requests allows us to exploit the non blocking functionality of the processor and DRAM thus reducing the clock cycle times required for execution of a program.

- The optimizations to replace requests pertaining to the same addresses or registers eliminate redundant requests thus further reducing the runtime without affecting the correctness.

- The optimization to select the next batch of request based on the blocking registers helps to reduce the delay of processor clock cycles when a blocking register is encountered.

2.2 Disadvantages

- The reordering can cause processor to block for certain requests that are irrelevant to the blocking instruction. This results in a tradeoff between the DRAM row buffer access time and processor blocked time.

- While selecting the appropriate batch of requests for a given row we select it based on the blocking registers involved in each batch. However this may/may not result in an efficient execution workflow for every program as illustrated in one of our test cases.

3 Testing

The testing method was completely manual. The test cases written manually are included in the `./tests/input` directory. The test cases were run with different parameters of the program and the outcomes were compared.

- `continuous_lw.txt` : This test case loads multiple values from the memory in the same register. This tests our optimization of deleting previous requests when same register is involved.
- `continuous_sw.txt` : This test case stores multiple values into the memory in the same address. This tests our optimization of deleting previous requests when same address is involved.
- `alternate_load_store.txt` : This test case includes alternate load and store instructions with different offsets to test additional deletion of SW requests once the LW request between them is deleted.
- `multiple_sw.txt` : This file tests elimination of redundant instructions that involve the same address and the same register.
- `choosing_row.txt` : This file tests our second optimization of choosing the batch of requests based on the blocking registers involved.
- `choosing_address_in_row.txt` : This file tests our third optimization of choosing the requests within the same row based on the blocking registers involved.
- `memory_skips.txt`: This test case issues multiple read/write requests to the driver with each successive request differing in the row of the address involved. This tests the reordering of the requests in the appropriate manner to reduce row buffer reloads.
- `disadvantage.txt`: This test case demonstrates a situation in which our method takes more cycles as compared with an non-blocking implementation with a queue.

- `random.txt`: This test case is randomly generated.

3.1 Result

The program output was observed and all the register, memory and row buffer values were inspected for correctness manually. In each of the test case the expected outputs matched. The program blocked when necessary and executed parallely when blocking was not required.

The output logs of the test cases are available in the `./tests/output` directory. The test cases were run with different configuration options.

4 Assumptions

- We have assumed that the maximum memory available to any user program is 2^{20} bytes. This does not include the memory required for storing instructions.
- We have assumed 32 registers for MIPS each of them storing 32 bits. We have hardwired the register `$zero` to the value 0 as is the case in MIPS hardware. We have also restricted use of kernel reserved registers `$26, $27`.
- We have assumed that exactly 1 clock cycle is required for each of the instructions except `lw, sw` to display the execution statistics.
- We have enforced tight syntax rules to enforce good coding practice. We disallow certain instructions like (a) `lw $t1, ($sp)` and (b) `lw $t1, 40231`. The first instruction skips supplying the offset to the register and the second one involves raw memory access.
- We have enforced the use of a whitespace between the operand and its arguments. For example `add$t1,$t2,$t3` is disallowed as there is no whitespace between `add` and `$t1`
- We have implemented branching instructions like `beq, bne, j` to take only labels and instruction numbers as the last argument.