INDIAN INSTITUTE OF TECHNOLOGY DELHI

COL216

Report

Assignment – 5

DRAM Request Manager for Multicore Processors

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 Overview

This assignment extends the earlier DRAM Driver to the multicore CPU case. The architecture now consists of N CPU cores, each running a different MIPS program, and sending DRAM requests to a DRAM Driver which interfaces with the DRAM

2 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

2.1 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.

2.2 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.

2.3 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.

2.4 DramDriver

- A `DramDriver` module is implemented to capture all `DRAM` requests from all cores and reorder them according to some pre determined heuristics.
- It is equipped with all optimizations to give the best performance. It includes forwarding, eliminating redundant `LW` and `SW` instructions.
- It also protects against starvation by appropriately context switching out DRAM requests of a core.
- It automaticaly converts the Virtual address space to the physical address space before queuing in a memory request.

3 Approach

The approach used to implement the `DramDriver` for multicore processors is described in this section.

3.1 Data

We propose to implement the `DramDriver` in hardware using a few lookup tables and some queues for storing the pending requests. The data modules required by our implementation are described in detail below.

3.1.1 Queues

8 queues, each with a size of 8 requests. This gives us a space for exactly 64 requests at a time. Queues need to be of finite size since we would have hardware supporting only a finite size and any implementation with an infinite size assumption is infeasible to implement.

Keeping in mind the design principle *Simplicity favours regularity*, we have supported only push and pop operations on the queue, which can be easily implemented using a shift register. Each queue can be indexed using a multiplexor as done in a cache. We have also assumed random access of any of the 8 elements of the queue which is easy to implement. A queue holds all the requests of the same row. Hence every queue deals with a different row of the DRAM and this makes it easier to schedule requests.

The decision for the size of each queue and the number of queues was taken keeping in mind the hardware constraints and design principle *Smaller is faster* However, our implementation is completely modular, and by changing just one value we can simulate the Driver with some other values to test which one seems to give the best throughput.

3.1.2 Look-up Tables

The state of the queues is maintained using several lookup tables which can be easily implemented in the hardware using multiplexors and is also used in many practical hardware implementations. The description of each look up table along with its corresponding name in our simulator code is given below. Assume N is the number of cores in the processor.

LUT name	Size	Index	Stored value
`__core2freq_LUT`	N	core number	Frequency of DRAM accesses (`lw`/ `sw`)
`__core2instr_LUT`	N	core number	Number of instructions executed.
`__core2PA_offsets_LUT`	N	core number	Virtual to physical address offset.
`__core2blocked_reg_LUT`	$N \times 3$	core number	The set of blocked registers.
`__queue2row_LUT`	8	queue number	row of requests in the queue.
`__queue2offset_LUT`	8	queue number	Used internally to store the change in size of queue since it was emptied.
`__core_reg2offsets_LUT`	$N \times 32$	(core num, register num)	Returns the (queue number, index in queue) pair of a `lw` request involving the given register of the core.

Table 1: Description of LUTs

3.2 Control

This section specifies how the data modules are used to implement the queuing model for DRAM requests. The following points briefly illustrate the working of our model. Assume that the relevant lookup tables are updated at each operation, the exact details of which are skipped here for simplicity.

- When a request is received, the row number is checked and a queue is searched with the given row, this can be done parallely in hardware using 8 comparators attached to the `__queue2row_LUT`. If a queue is found with the row, a push operation is attempted in the queue, if the queue is full then an exception is thrown which signals the processor that it should retry the request once the queue has some space.

- When a request is completed it is removed from the queue (pop), updating the relevant lookup up tables to ensure correctness.

- The queue switching logic takes into account the blocked registers of each core (`__core2blocked_reg_LUT`), the number of instructions executed by a core (`__core2instr_LUT`), the frequency of DRAM accesses by a core (`__core2freq_LUT`) and creates a combined metric to decide the next queue. All this logic can be handled by a combinational circuit and is feasible to implement in hardware.

- The switching logic first tries to schedule the queue that has a blocking request. However, if multiple queues with such requests exist then the core with the lowest frequency of DRAM accesses and largest number of instructions is chosen (to increase the throughput).

- The circuits also incur a delay of some clock cycles and a busy bit is maintained inside the controller. When the controller is busy, it does not accept any more requests, and generates an exception.

3.3 Optimizations

We have also implemented several optimizations to eliminate redundant work for increasing the throughput.

- `Forwarding:` Whenever we recieve a `lw` for an address such that a `sw` for it already exists in our queue, we do not enqueue the `lw` request in the queue and instead forward the value directly to the corresponding register from the value stored in the `sw` request in the queue.
- `Redundant sw:` Before inserting a `sw` request in the queue we lookup for a matching `sw` request in the queue and nullify the searched request.
- `Redundant lw:` Before inserting a `lw` request in a queue, a matching `lw` request is obtained using the `__core_reg2offsets_LUT` lookup table. If the lookup table contains valid entries the request corresponding to the matching request is nullified.
- `Starvation:` The driver executes all requests in the current queue. However to prevent starvation a counter is maintained each time a request is completed, if this counter exceeds the queue size then the queue is switched out and some other queue is chosen to execute requests.

Note: Since the queue size is finite (8) looking up a request in a queue can be implemented using 8 comparators which would operate parallely to check for the required conditions.

3.4 Advantages

- The approach ensures that the overall throughput is maximized.
- The optimizations to replace requests pertaining to the same addresses or registers eliminate redundant requests thus further reducing the runtime without affecting the correctness.
- Forwarding enables us to entirely skip going to the DRAM and fetching the value directly in a short period of time.
- Eliminating starvation ensures that a single core does not eat up all the resources on a DRAM driver.

- The sophisticated scheduling logic incorporates all parameters (instruction access count, frequency of DRAM accesses, presence of blocking requests) and combines them to generate a measure that is compared to decide the next queue in schedule. This metric plays an essential role in determining the throughput.

3.5 Disadvantages

- Our approach will be useful for programs with large number of instructions and comparing the throughput on small programs could give incorrect interpretations.
- Since the queue supports only push and pop, we can execute requests starting from the front only. However when a queue is scheduled it may be possible that a blocking request is present at the end of the queue, and it would have to wait for all the requests to its front.

4 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.

- `test1.txt`: description here.
- `test2.txt`: description here.
- `test3.txt`: description here.

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

5 Assumptions

- We have assumed private physical spaces for each core, and this is achieved by dividing the total available memory into equal parts between each core. Although, in real implementations this would be done using a page table and the memory allocation is handled by the OS, but for simulation purposes our assumption is quite reasonable and shouldn't affect simlulation of ordinary programs.

- 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.