

Understanding and Evaluating the Performance of DRAM Memory Controller Policies under Various Algorithms Using DRAMsim

Tanima Dey, Enamul Hoque, Sudhanva Gurumurthi

Department of Computer Science
School of Engineering and Applied Sciences
University of Virginia
Charlottesville, Virginia, 22904

td8h@virginia.edu, eh6p@virginia.edu, gurumurthi@cs.virginia.edu

December 4, 2008

Abstract

Memory system is an integral part of any processor as it is responsible to provide one of the major resources that a program needs to run. The performance of a program, benchmark or processor sometimes is much dependent on that of memory, regardless of underlying architectural design. Again the performance of the memory is dependent on how it accesses its content, i.e, the memory controller policies. In this paper, we have compared the performance of various memory controller policies, including conventional policies, such as, FCFS, priority-based scheduling etc. with a novel reinforcement learning algorithm. This comparison can help us to better understand how the performance of the memory access is affected by the underlying memory controller's scheduling policies.

Keywords: Memory controller, Memory controller policies, DRAM, Reinforcement learning algorithm, DRAMsim

1. Introduction

Memory is an important part of the CPU (Central Processing Unit) primarily because it contains the programs and the data used by the programs. By the term memory, it actually refers to the primary memory system of the CPU which is responsible for providing space for program execution. On the other hand, secondary memory systems refer to the hard disks, flash drives, optical drives etc. for storing the programs permanently unless it is explicitly deleted by the users. The performance of a processor is much dependent on that of the primary memory as it is directly related to the program execution. The more efficient the underlying memory system will be, the faster the processor will run and the better the program will perform.

Memory in microprocessors is also known as random access memory or in short RAM. RAMs are mainly of two types – static RAM or SRAM and dynamic RAM or DRAM. Now-

a-days almost every processor uses DRAM. The speciality of DRAM is that it stores each bit of data in a separate capacitor within an integrated circuit. Since real capacitors leak charge, the information eventually fades unless the capacitor charge is refreshed periodically. Because of this refresh requirement, it is a dynamic memory as opposed to SRAM and other static memory. The advantage of DRAM is its structural simplicity: only one transistor and a capacitor are required per bit, compared to six transistors in SRAM. This allows DRAM to reach very high density [1].

Memory controller in DRAM performs the task of read and write in the DRAM and also sends the necessary current signal to charge the capacitor in order to make sure that the corresponding memory address contains the voltage to represent the correct bit. In the purpose of controlling the read/write operation in the appropriate memory location, one of the major tasks of memory controller is to control the order of these operations to make sure that the performance during the memory access also remains good. There are a number of scheduling policies taken by memory controllers, such as, First-come first-serve(FCFS), priority based scheduling, First-ready FCFS(FR-FCFS) etc.

In this work, we have implemented a novel algorithm proposed in [2] which is a machine learning technique to optimize the operations of memory controller for accessing the DRAM. The algorithm is called the Reinforcement Learning (RL) algorithm for memory controllers. Machine learning is the field of computer science where computer programs and algorithms try to improve their performance automatically with the experience. Reinforcement learning is a sub-area of machine learning concerned with how an agent ought to take actions in an environment so as to maximize some notion of long-term reward. Reinforcement learning algorithms attempt to

find a policy that maps states of the world to the actions the agent ought to take in those states [3]. This type of learning is also known as “learning from interaction”.

We have used DRAMsim [4] as the simulator to get the result of our implementation. This is a special simulator designed and developed in University of Maryland, College Park. There are a number of memory controller scheduling policies available in the simulator detailed described in Section 3.1. We have incorporated the RL algorithm in the source code of the simulator and ran in on several trace files obtained from [5].

The paper is organised as follows. Section 2 contains the related research works in different memory controller and its scheduling policies. Section 3 contains detailed description of two major portions of our work, DRAMsim and the RL algorithm. Section 4 contains our some implementation details and research methodology. The results of the simulation and comparisons are made in Section 5. And Section 6 concludes.

2. Related Work

There has been numerous works in improving the performance of the memory controller in a number of ways. In [6], the authors have designed a new memory system architecture, called *Impulse* that adds two important features to a traditional memory controller. First, *Impulse* supports application-specific optimizations through configurable physical address remapping which improves applications’ cache and bus utilization. Second, *Impulse* supports prefetching at the memory controller, which can hide much of the latency of DRAM accesses.

In [7], the authors have suggested a memory controller design that provides a guaranteed minimum bandwidth and a maximum

latency bound to the intellectual property components (IP). This is accomplished using a novel two-step approach to predictable SDRAM sharing. The authors of [8] describes the development of an intelligent memory interface that can exploit compiler-provided information on streamed memory access patterns to improve memory bandwidth.

Modern DRAM systems consist of dual inline memory modules (DIMM), which are composed of multiple DRAM chips put together to obtain a wide data interface. Each of the DRAM chip is organized as multiple independent banks each of which is further organised as rows \times columns [2]. In [9], this organization is referred to as *3-D structure of DRAM chips*. This paper introduces memory access scheduling, a technique that improves the performance of a memory system by reordering memory references to exploit locality within the 3-D memory structure.

It is mentioned in [2], there is no works done previously in DRAM scheduling that provides a scheduler that can learn the long-term impact of its scheduling decisions. In [10], the authors proposes a history-based memory scheduler that adapts to the mix of read and write requests from the processor. The authors in [2], have adapted the history-based scheduler as their baseline controller organization.

3. Major Components

In this section, two major portion of our work is described – DRAMsim and RL algorithm. These are described as follows:

3.1 DRAMsim

DRAMsim [4] is a special simulator developed for DRAMs at ECE department of University of Maryland, College Park. The authors have described their developed simulator and its

interfaces with other simulators. DRAMsim also comes with additional tools. One of the tools is DRAM VisTool [11], a visual tool developed in Java that creates a graphical representation of activity on the various busses and in physical resources over the course of time. We used this tool for visualising the results of the simulation.

The system architecture of the DRAMsim is shown in Figure 1. The simulator is designed for multi-processor system. The life of a memory transaction begins when any functional unit requests for a DRAM access. The requests are placed in the BIU as long as there is a free spot available there. The simulation of a system controller begins when the memory access request is placed into the transaction queue from BIU. The transaction queue maps the physical address of the transaction into the memory address in terms of channel ID, rank ID, bank ID, row ID and column ID via an address mapping scheme. The selection of transaction from the transaction queue to the actual DRAM system is done by a number of memory controller policies. The policies that are supported by the DRAMsim are FCFS, Greedy, OBF (Open Bank First), RIFF (Read and Instruction Fetch First), Least pending and Most pending. We have incorporated our implemented RL algorithm as another scheduling policy of the memory controller.

3.2 Reinforcement Learning Algorithm

Reinforcement Learning studies how autonomous agent situated in a stochastic environment can learn to maximize the cumulative sum of a numerical reward signal received over their lifetime through the interaction with their environment. The agent basically maintains some states, actions and a reward assignment scheme. The agent senses the current state of its environment and exe-

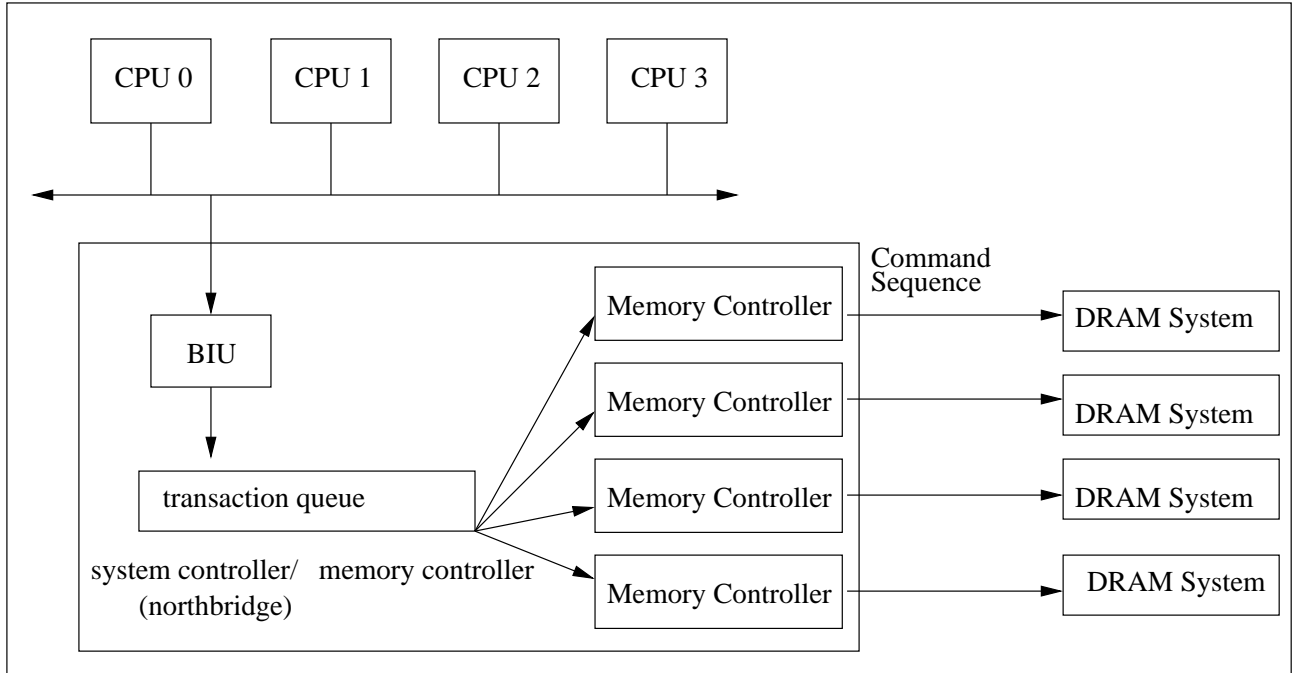


Figure 1: Transaction Queue and Memory Controller System Architecture [4]

cutes an action. This results in a change in the state of the environment, and produces an immediate reward. The agent’s goal is to maximize its long term cumulative reward by learning an optimal policy that maps state to an action [2].

In our case, the agent is the memory controller which maintains some state and takes an action to maximize the reward under the policy of maximizing the long term data bus utilization. Each state maintains some attributes which are calculated from the memory access instructions stored in the transaction queue (TQ) of the memory controller. The scheduler for memory controller is given an immediate reward of 1 for each time it schedules a read or write command and a reward of 0 at all other times. The detailed RL algorithm can be found in [2].

The RL algorithm maintains a table of Q-values for all possible state-action pair which

initially contains the highest possible Q-value. In each cycle, the scheduler observes the TQ and finds all DRAM commands that can be legally issued without violating timing constraints. Normally it picks a command to explore with the highest Q-values. The selected command is issued at the clock edge and records the reward for that action. In a given cycle, the scheduler also updates Q-values according to the immediate reward and Q-values of current and previous state-action pairs.

4. Our implementation

In this section, the major steps of our implementation of the RL algorithm are described along with portion of the source code. These are as follows:

4.1 Understanding Existing DRAMsim Code

Before integrating the Reinforcement Algorithm, we had to understand the code of the DRAMsim simulator completely. Here we will describe a high level description of how the incoming memory instructions are scheduled by the memory controller. Each incoming memory request is stored in a bus interface unit slot. These slots are freed when the requests are moved to the transaction queue. When a request is moved to a transaction queue, it needs a set of commands to be generated for successful completion of the request. So each entry in the transaction queue is essentially a set of commands such as RAS, CAS, PRECHARGE, REFRESH, etc. The job of the memory controller is to schedule these commands from multiple transactions of the transaction queue. A transaction can be moved from the transaction queue only after all of its commands are scheduled and executed properly. During each cycle, the memory controller searches through the transaction queue for finding a command to schedule according the specified scheduling algorithm.

4.2 Converting Sample Trace Files to DRAMsim Input File

The DRAMsim input trace file takes three parameters in each line. Each line contains the address to be accessed, the type of operation to be performed on the address (read/write) and the timestamp when the operation needs to be performed. The input trace-files from [5] have similar input parameters. Those trace files have six parameters in each line. Each line contains the address to be accessed, type of operation to be performed, timestamp, attribute, processor field etc. The conversion of the input trace-file is done by writing a C program which will convert the input trace file

from [5] to a DRAMsim compatible input file. In both of the input files, the size of the address field is 32 bits. The trace-files include also the I/O operations and several other operations not related to the memory access. We have carefully omitted them and made the timestamps such that it runs on DRAMsim smoothly without deadlock.

4.3 Implementing the Required Data Structures of the RL Algorithm

First of all we needed to define a state which depends on a number of attributes like number of reads, writes, etc in the transaction queue. We implemented each state by a two dimensional array whose dimensions are number of reads and writes in the transaction queue. For simplicity of implementation, we constructed each state based on number of reads and writes in the transaction queue only. Then we implemented the table that maps each state action pair to a Q-value. Here this table's rows are indexed by integers which we get from the two dimensional array representing states. And columns are indexed by all possible commands.

4.4 Selecting the Highest Q-Value Command from the List of All Valid Commands

At each step of the RL algorithm, we need to generate the list of all the valid commands and choose the one containing the highest Q-value from the list. The existing code has a method which can determine whether a command is valid at a particular time. We iterate through the transaction queue and check the commands under each transaction and add each valid command determined by that method to our list of all valid commands. After enumerating the list of valid commands completely, we select the command having the highest Q-

value with the help of the Q-value table indexed by the current state and each candidate command. After selecting a command, we determine the immediate reward and update the Q-value of the selected command.

4.5 Integration with the Existing Code

After writing the code separately and testing it under various inputs, we integrated it with the existing code of DRAMsim. We added option for selecting RL algorithm for the memory controller policy from the input and if RL is selected by the user then we execute our portion of the code each time when scheduling of the existing commands in the transaction queue is necessary. The key portion of the code is given in Appendix A.

| Attributes | Values |
|-------------------|-----------------------|
| type | s dram |
| datarate | 133 |
| channel_count | 1 |
| channel_width | 8 |
| PA_mapping_policy | s dram_close_page_map |
| row_buffer_policy | close_page |
| rank_count | 8 |
| bank_count | 4 |
| row_count | 8192 |
| col_count | 1024 |
| t_ras | 10 |
| t_rp | 3 |
| t_rcd | 3 |
| t_cas | 3 |
| t_cac | 2 |
| t_cwd | 0 |
| t_rtr | 0 |
| t_dqs | 0 |
| t_rc | 8 |

Table 1: DRAM configuration for simulation

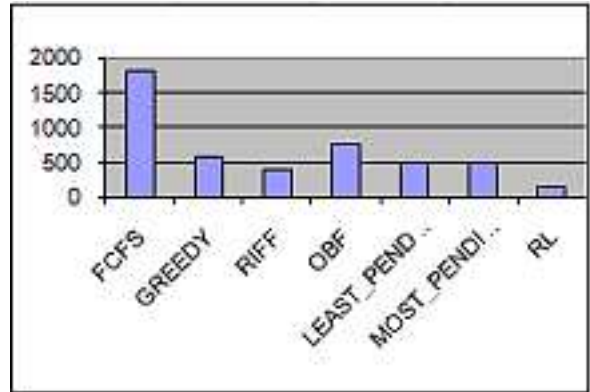


Figure 2: Average latency of read instructions

5. Results and Evaluation

After implementing and integrating the Reinforcement Learning Algorithm with the DRAMsim simulator, we evaluated its performance in comparison with the other scheduling algorithms supported by DRAMsim. We simulated the same trace file under various scheduling algorithms and compared their performances. In all cases we use the DRAM with the configuration given in Table 1:

First we compared the average latency of all the read instructions under various scheduling algorithms, FCFS, Greedy, RIFF, OBF, least-pending, most-pending, and RL. For each algorithm we generated the latency of each read instruction from DRAMsim and then calculated the average value. The comparison is shown in Figure 2. From the figure we can see that the RL algorithm outperforms all other scheduling algorithms in the case of average latency of read instructions for the given trace file. Here the latencies are taken from the outputs of the DRAMsim, which does not provide any unit of the latency, rather the latency is a count of how many times a loop runs before a read request is completed. So the unit along the y axis of Figure 2 should be interpreted in the same

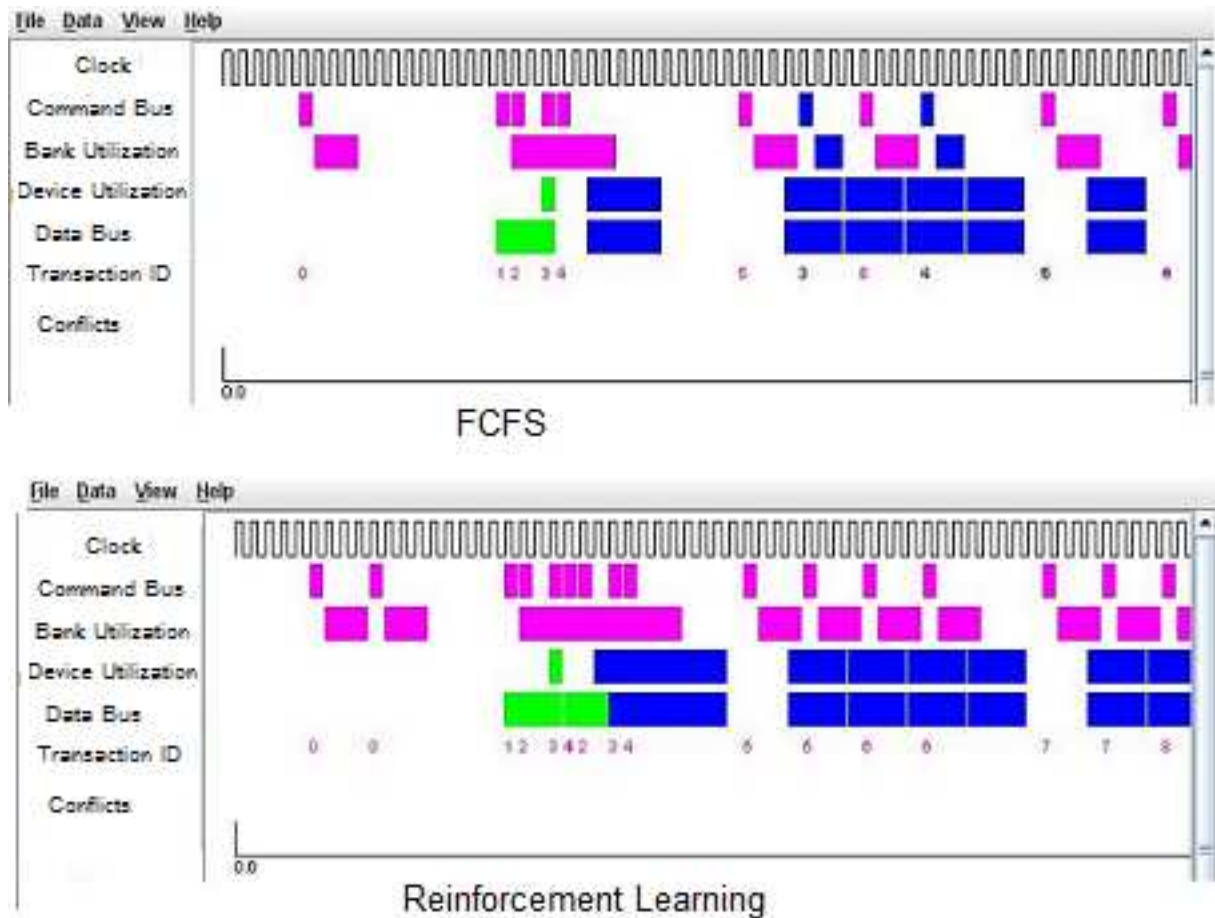


Figure 3: Output timing diagram using VisTool for FCFS and RL algorithm

way.

Then we feed the schedule generated by DRAMsim for each scheduling algorithm to the VisTool [11] for generating timing diagrams. VisTool produces the output timing diagram for supported memory controller policies. For example the timing diagrams for FCFS and RL algorithm are shown in Figure 3 for a particular time period. From these two figures we can infer that by using RL algorithm we can issue more commands within the particular time period. Next time frames show similar results as well. The data bus, banks and devices are more properly utilized

for RL as well. For all these experiments we used $\gamma = 0.95$ and $\alpha = 0.1$, where α is the learning rate and γ is the discount rate parameters of the RL algorithm [2].

6. Conclusion

The effect of different memory controller policies are well understood from the result section of our simulation and we can infer that RL algorithm provides the best performance in terms of latency. The comparison shows that very basic scheduling algorithms are easier to implement but cannot provide us bet-

ter performance over the complex ones. As memory can be the bottle-neck of processor performance, efficient memory controller policies can be one of the ways to overcome that. And machine learning algorithms are preferable option to use as they provide greater insights in the performance utilization. Reinforcement learning algorithm is one of the different ways of learning. We are hopeful that better memory controller policies will be designed to maximise the performance of various parallel and sequential applications.

7. Acknowledgements

We would like to express our gratitude toward Professor Dr. Bruce Jacob, University of Maryland, College Park, for answering our queries about DRAMsim over e-mail.

References

- [1] Dynamic Random Access Memory, http://en.wikipedia.org/wiki/Dynamic_random_access_memory
- [2] Engin Ipek, Onur Mutlu, Jose Matinez, Rich Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach", International Symposium on Computer Architecture (ISCA), June 2008.
- [3] Reinforcement Learning, http://en.wikipedia.org/wiki/Reinforcement_learning
- [4] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, Bruce Jacob, "DRAMsim: A Memory System Simulator", SIGARCH Computer Architecture News, September 2005.
- [5] Trace Files, <http://tds.cs.byu.edu/tds/>
- [6] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixin Zhang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael Parker, Lambert Schaelicke, Terry Tateyama, "Impulse: Building a Smarter Memory Controller", Fifth Annual Symposium on High Performance Computer Architecture(HPCA), January 1999.
- [7] Benny Akesson, Kees Goossens, Markus Ringhofer, "Predator: A Predictable SDRAM Memory Controller", International Conference on Hardware-Software Codesign and System Synthesis (Code+ISSS), September 2007.
- [8] Sally McKee, William Wulf, "A Memory Controller for Improved Performance of Streamed Computations on Symmetric Multiprocessors", International Parallel Processing Symposium (IPPS), April 1996.
- [9] Scott Rixner, William Dally, Ujval Kapasi, Peter Mattson, John Owens, "Memory Access Scheduling", International Symposium on Computer Architecture (ISCA), June 2000.
- [10] Ibrahim Hur, Calvin Lin, "Adaptive History-Based Memory Schedulers", International Symposium on Microarchitecture (MICRO), December 2004.
- [11] Vincenet Chan, Austin Lanham, "DRAM VisTool User Manual", <http://www.ece.umd.edu/DRAMsim/download/VisTool-manual.pdf>

Appendix A: Key portion of source code of the implemented RL algorithm (from Section 4.5)

```
if(transaction_selection_policy == RL)
{
    if (col_command_bus_idle(chan_id) || row_command_bus_idle(chan_id))
    {
        get_current_state(chan_id);
        all_valid_commands = NULL;
        total_commands = 0;

        get_all_valid_commands(chan_id);
        int max_Qval_index = get_command_with_highest_Qvalue();

        for(ee = 1; ee < command_indices[max_Qval_index]; ee++)
        {
            issued_command = issued_command->next_c;
        }

        double observed_reward = get_observed_reward(issued_command);

        issue_cmd(now, issued_command, this_t, chan_id);

        if(first_time==1)
        {
            Qprev = observed_reward;
            index1 = getCurrentStateIndex();
            index2 = issued_command->command;
        }

        else
        {
            Qprev = calculateUpdatedValue(Qprev, Qcurrent);
            updateQvalue(index1, index2, Qprev);
            index1 = getCurrentStateIndex();
            index2 = issued_command->command;
            Qprev = Qcurrent;
        }

        first_time++;
    }
}
```