Memory Centric Systems for AI Final Project Report: Play with DRAMsim3

2020314087 Sungmin Ryu
School of Electrical and Electronic Engineering
Yonsei University, Seoul, Korea

***Abstract*—*DRAMsim3* [2] is a cycle-accurate DRAM simulator, which faithfully models almost all aspects of modern DRAM, including the timings that we have covered, power consumption, etc. In order to understand how *DRAMsim3* works, I first made a trace-based memory controller. With the memory controller, I tested the ResNet-18 traces changing scheduling methods of the simulator. Next, I built a timing memory controller to simulate simple memory timing test. Finally, I connected *DRAMsim2* and *gem5* [1] and then simulated AlexNet first convolutional layer. With the simulation, I tested *loop-unrolling* method which can reduce loop overhead. This report describes how to use the trace-based and timing memory controller, difference between FCFS and FR-FCFS scheduling, and some results of simulation.**

I. Introduction

This report consists of three parts. First, I explain how to use DRAMsim3 as a trace-based simulator and timing-based simulator respectively and show the results of the ResNet-18 trace simulation, in *Section II*. Second, I compare FCFS and FR-FCFS with the trace-based memory controller, in *Section III*. In the last section, I describe the *loop-unrolling* method and test the method with gem5-DRAMsim2.

A. Overview of Project Sources

```
.
|-ext                       // ext includes DRAMsim3
  |--configuration.cc        // for my convenience
  |--DRAMsim3                // DRAMsim3
     |--libdramsim3.so       // DRAMsim3 shared library
     |--Makefile             // DRAMsim3 Makefile
     |--src                  // DRAMsim3 source
     |--...                  // DRAMsim3 other directories and files
|-Makefile                  // My project Makefile
|-src                       // My project source
  |--main.cc                 // My project main
  |--mem_ctrler.cc           // My project memory controller source
  |--mem_ctrler.h            // My project memory controller header
|-traces                    // ResNet-18 traces
  |--trace1.txt
  |--trace2.txt
  |--trace3.txt
  |--trace4.txt
  |--trace5.txt
  |--trace6.txt
  |--trace7.txt
  |--trace8.txt
```

- Source Tree -

Before we start, check the source tree above which contains main files and directories of this project. *DRAMsim3* is located in *'ext'* and there are three main source files in *'src'* directory: mem_ctrler.h, mem_ctrler.cc, and main.cc. The mem_ctrler.h and mem_ctrler.cc have some memory controller classes and functions respectively. In the main.cc, I can call the memory controllers and simulate trace-based simulation. All sources are available on my github(https://github.com/WheatBeer/play_with_dramsim3).

B. DRAMsim3 Build Process

During trace and timing simulation, I use DRAMsim3 as shared library(*libdramsim3.so*) which is in ext/DRAMsim3(after library building). One good thing about using the shared library is that we do not need to build every time the sources are changed. I cloned *DRAMsim3* in the *'ext'* directory and changed DRAMsim3/src/configuration.cc to ext/configuration.cc, for my convenience(to save simulation outputs with output prefix). After the change, I built *DRAMsim3* and then *libdramsim3.so* came out.

```
/* DRAMsim3 headers */
#include <common.h>
#include <dramsim3.h>
```

- DRAMsim3 Headers -

Now, we are able to use memory systems in *DRAMsim3* by adding some headers as above. The 'common.h' includes *Transaction* class which can take trace files, so the header is needed to make a trace-based memory controller. In addition, the 'dramsim3.h' is a required header to create *DRAMsim3*'s *MemorySystem*.

II. Memory Controller

A. Memory Controller Base Class

```
/* Memory Controller Base */
class mem_ctrler_base_t {
public:
    mem_ctrler_base_t(const std::string& m_config_file,
                      const std::string& m_output_dir)
    : clk(0) {
        mem = dramsim3::GetMemorySystem(m_config_file, m_output_dir,
                                std::bind(&mem_ctrler_base_t::read_callback,
                                          this, std::placeholders::_1),
                                std::bind(&mem_ctrler_base_t::write_callback,
                                          this, std::placeholders::_1));
    }
    virtual ~mem_ctrler_base_t() {};
    virtual void tick() = 0;
    virtual void read_callback(uint64_t m_addr) = 0;
    virtual void write_callback(uint64_t m_addr) = 0;
    void print_stats() { mem->PrintStats(); }

protected:
    dramsim3::MemorySystem *mem;
    uint64_t clk;
};
```

- Memory Controller Base Class -

The *mem_ctrler_base_t* class is the same as *CPU* class in DRAMsim3/src/cpu.h(only the class's function and valable names are changed). The class has *MemorySystem* pointer, *'mem'*, which can use DRAMsim3 memory APIs in DRAMsim3/src/dramsim3.h. In src/main.cc, I can initiate the class's child classes(trace-based and timing memory controller) and get some outputs.

B. Trace-based Memory Controller

```
/* Trace Memory Controller */
class trace_mem_ctrler_t : public mem_ctrler_base_t {
public:
    trace_mem_ctrler_t(const std::string& m_config_file,
                       const std::string& m_output_dir,
                       const std::string& m_trace_file)
    : mem_ctrler_base_t(m_config_file, m_output_dir),
      get_next(true), trace_cnt(0), callback_cnt(0) {
        trace_file.open(m_trace_file);
        if(trace_file.fail()) {
            std::cerr << "Trace file does not exist" << std::endl;
            exit(1);
        }
    }
    ~trace_mem_ctrler_t() { mem->GetQueueSize(); trace_file.close(); }
    void tick();
    void read_callback(uint64_t m_addr) { callback_cnt++; return; }
    void write_callback(uint64_t m_addr) { callback_cnt++; return; }
    bool is_end() { return trace_file.eof() && trace_cnt == callback_cnt; }

private:
    bool get_next;
    size_t trace_cnt;
    size_t callback_cnt;
    std::ifstream trace_file;
    dramsim3::Transaction trans;
};
```

- Trace-based Memory Controller Class -

In order to simulate trace memory requests, a trace-based memory controller is needed. The controller is already in DRAMsim3/src/cpu.h as *TraceBaseCPU* class which is inherited from *CPU* class, so users can use this by executing 'dramsim3main(executable file)' with a flag(e.g. -t trace.txt). However, to run this way, users have to put cycles(e.g. -c 100000) and the cycles do not guarantee the end of trace. For instance, the simulation is finished in the cycles, but traces may still remain. So, to track the last cycle, I made a new trace-based memory controller, *trace_mem_ctrler_t*, which is inherited from *mem_ctrler_base_t* class. The difference between *trace_mem_ctrler_t* class(new) and *TraceBaseCPU* class(original) is whether the last cycle can be tracked automatically. The new trace-based memory controller has some more functions and variables, such as *is_end()*, *trace_cnt*, and *callback_cnt*.

The way to track the last cycle is to count sending and receiving both and compare them. When sending transactions, *trace_cnt* is counted and when receiving callback(read and write), *callback_cnt* is counted. The *callback_cnt* is incremented in the callback functions inside the controller and the increase of *trace_cnt* occurs at the *tick()*, as you can see in the code below.

```
/* Trace Memory Controller */
void trace_mem_ctrler_t::tick() {
    mem->ClockTick();
    if(!trace_file.eof()) {
        if(get_next) {
            get_next = false;
            trace_file >> trans;
            trace_cnt++; // Here
        }
        if(trans.added_cycle <= clk) {
            get_next = mem->WillAcceptTransaction(trans.addr, trans.is_write);
            if(get_next) {
                mem->AddTransaction(trans.addr, trans.is_write);
            }
        }
    }
    clk++;
    return;
}
```

- Trace-based Memory Controller's tick() -

After all trace transactions are sent and *DRAMsim3*'s *MemorySystem* process the last transaction, *callback_cnt* becomes equal to *trace_cnt*. The *is_end()* function checks the end of the trace file and if *trace_cnt* and *callback_cnt* are the same.

```
        /* Trace Memory Controller Test */
        trace_mem_ctrler_t *trace_mem_ctrler = new trace_mem_ctrler_t(dram_cfg_path
       , output_path, trace_path);
        while(!trace_mem_ctrler->is_end()) {
          trace_mem_ctrler->tick();
        }
        trace_mem_ctrler->print_stats();
        delete trace_mem_ctrler;
```

- main.cc -

In the main.cc, we can send all transactions in the trace file to the simulator and check whether the transaction processing is finished, using the *is_end()*.

```
$ wc -l ./traces/*
  110272 traces/trace1.txt
   33920 traces/trace2.txt
   77568 traces/trace3.txt
   30976 traces/trace4.txt
   80128 traces/trace5.txt
   29184 traces/trace6.txt
   55104 traces/trace7.txt
   96000 traces/trace8.txt
  513152 total

$ grep -o READ trace1.txt | wc -w
9920
$ grep -o WRITE trace1.txt | wc -w
100352
```

- The number of trace transations -

There are 8 ResNet-18 traces in the *traces* directory. The *trace1* has 110272 transactions, of which READ transactions are 9920 and WRITE transactions are 100352. In the output of *trace1* simulation below, *num_reads_done* and *num_write_done* are the number of READ and WRITE transactions respectively. (There is 1 count difference at *num_reads_done*, but I don't have enough time to debug.) With the output, we can see that the number of cycles required to process all transactions is 559529(*num_cycles*).

```
########################################
## Statistics of Channel 0
########################################
hbm_dual_cmds                 =            0   # Number of cycles dual cmds issued
num_read_row_hits             =         9703   # Number of read row buffer hits
num_write_buf_hits            =            0   # Number of write buffer hits
num_reads_done                =         9921   # Number of read requests issued
num_writes_done               =       100352   # Number of read requests issued
num_cycles                    =       559529   # Number of DRAM cycles
```

- Simulation Output(trace1) -

To validate my trace-based memory controller, I use the original memory controller which name is *TraceBaseCPU* in DRAMsim3/src/cpu.h. I put 559529 in cycle(-c 559529) and simulate *trace1*, you can test as command below. After simulation, dramsim3.txt which is simulation output come out and the results are perfectly consistent with my results (even the *1 count difference* is the same).

```
$ ./dramsim3main DDR4_8Gb_x8_2400.ini -c 559529 -t trace1.txt
```

C. Timing Memory Controller

A downside of trace-based simulation is it cannot simulate the real slowdown of an application, since the requests will arrive at the designated cycle no matter what. We can create a timing memory controller which has a fixed number of load buffer. In this report, I only describe how to make the timing memory controller and how the controller works with infinite buffer.

```
/* Timing Memory Controller */
class timing_mem_ctrler_t : public mem_ctrler_base_t {
public:
    timing_mem_ctrler_t(const std::string& m_config_file,
                        const std::string& m_output_dir)
    : mem_ctrler_base_t(m_config_file, m_output_dir),
      send_cnt(0), callback_cnt(0) {}
    ~timing_mem_ctrler_t() {}
    void tick() { mem->ClockTick(); clk++; return; }
    bool will_accept_transaction(uint64_t addr_, bool is_write_) const;
    void add_transaction(uint64_t addr_, bool is_write_);
    void read_callback(uint64_t m_addr);
    void write_callback(uint64_t m_addr);
    bool is_end() { return send_cnt == callback_cnt && queue.size() == 0; }

private:
    size_t send_cnt;
    size_t callback_cnt;
    std::list<mem_request_t> queue;
};
```

- Timing Memory Controller Class -

```
/* Timing Memory Controller */
bool timing_mem_ctrler_t::will_accept_transaction(uint64_t addr_, bool is_write_)
const {
    return mem->WillAcceptTransaction(addr_, is_write_);
}

void timing_mem_ctrler_t::add_transaction(uint64_t addr_, bool is_write_) {
    mem->AddTransaction(addr_, is_write_);
    mem_request_t mem_request(addr_, is_write_);
    queue.push_back(mem_request);
    send_cnt++;
    return;
}

void timing_mem_ctrler_t::read_callback(uint64_t m_addr) {
    for(auto it = queue.cbegin(); it != queue.cend(); ++it) {
        if(it->addr == m_addr) {
            callback_cnt++;
            queue.erase(it);
            return;
        }
    }
}

void timing_mem_ctrler_t::write_callback(uint64_t m_addr) {
    for(auto it = queue.cbegin(); it != queue.cend(); ++it) {
        if(it->addr == m_addr) {
            callback_cnt++;
            queue.erase(it);
            return;
        }
    }
}
```

- Timing Memory Controller's Functions -

The above memory controller, *timing_mem_ctrler_t*, has wrapper functions which include memory system functions in ext/DRAMsim3/src/dramsim3.h. In the trace-based memory controller, *WillAcceptTransaction()* and *AddTransaction()*, which are memory system functions, are used inside the *tick()*. You can take this method in timing simulation. Another way is to wrap these functions with *will_accept_transaction* and *add_transaction* respectively and use them outside the controller. For example, in your timing simulator, *will_accept_transaction* should be called first to check whether DRAMsim3 can receive transactions. If DRAMsim3 can accept the transactions, you can send memory requests through *add_transaction*. Next, you have to run *tick()* for processing the transactions in *DRAMsim3*. The difference between these two methods is to schedule memory requests inside or outside the controller(I am currently looking for a suitable method in my research).

```
/* Memory Request Structure */
struct mem_request_t {
  mem_request_t(uint64_t m_addr, bool m_is_write)
    : is_write(m_is_write), addr(m_addr) {}
    bool is_write;
    uint64_t addr;
};
```

- Memory Request Structure -

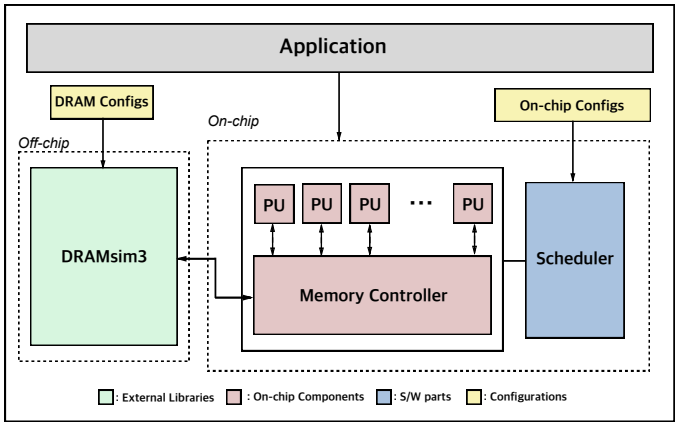

Fig. 1: Timing Simulation Example

The *timing_mem_ctrler_t* has a queue which can contain memory requests(*mem_request_t*). The queue size can be fixed as needed.

To reduce description in the last section(*gem5 with DRAMsim3*) and explain how the *timing_mem_ctrler_t* works, I use Fig.1 as an example. When application is launched, PUs(Processing Units) send memory requests to the memory controller. If the memory controller queue has enough space, the memory requests are accepted and send to DRAMsim3. If not, the PU is stalled until queue can take the requests. After DRAMsim3 processing, the memory controller receive processed address through callback functions. With the callback functions, memory controller should send data or message to PUs. In *gem5*, there can be various combinations of CPU and caches by configuration scripts.

III. DRAM Scheduler

A. FCFS vs FR-FCFS

DRAMsim3 follows FR-FCFS(first-ready-first-come-first-serve) policy according to its github(https://github.com/umd-memsys/DRAMsim3). I change the policy to FCFS as follows.

```
Command CommandQueue::GetFirstReadyInQueue(CMDQueue& queue) const {
    for (auto cmd_it = queue.begin(); cmd_it != queue.end(); cmd_it++) {
        Command cmd = channel_state_.GetReadyCommand(*cmd_it, clk_);
        if (!cmd.IsValid()) {
            continue;
        }
        /*if (cmd.cmd_type == CommandType::PRECHARGE) {
            if (!ArbitratePrecharge(cmd_it, queue)) {
                continue;
            }
        } else */ if (cmd.IsWrite()) {
            if (HasRWDependency(cmd_it, queue)) {
                continue;
            }
        }
        return cmd;
    }
    return Command();
}
```

- FR-FCFS to FCFS -

DRAMsim3 memory system has two types of queue: transaction queue and bank command queue. The default value of transaction queue size is 32 and each bank has 8 command queue size. From transaction queue to bank command queue, FCFS policy is applied. After commands sent to command queue, row-hit commands are served first following FR-FCFS.

	trace1	trace2	trace3	trace4
READ Transactions	9920	1152	52480	18432
WRITE Transactions	100352	32768	25088	12544
Total Transactions	110272	33920	77568	30976
(1) FR-FCFS				
a. num_read_row_hits	9703	1142	51910	18156
b. num_write_row_hits	95949	30667	24860	12430
c. num_act_cmds	4636	2096	798	392
d. num_pre_cmds	4607	2080	790	381
(2) FCFS				
a. num_read_row_hits	9697	1142	51910	18088
b. num_write_row_hits	95944	30666	24858	12429
c. num_act_cmds	4769	2103	805	521
d. num_pre_cmds	4740	2087	799	510
(1) - (2)				
a. num_read_row_hits	6	0	0	68
b. num_write_row_hits	5	1	2	1
c. num_act_cmds	-133	-7	-7	-129
d. num_pre_cmds	-133	-7	-9	-129

Fig. 2: Simulation Results(FR-FCFS vs FCFS)

B. Simulation Results and Analysis

I simulate ResNet-18 trace1-4 changing scheduling policy. Fig. 2 is results of the simulation. The yellow box shows the subtracted values from FR-FCFS to FCFS. In most cases, row-hit ratio is higher when FR-FCFS scheduler is used than FCFS. Since the traces have their own charateristics, the outputs are different for each trace.

IV. GEM5 WITH DRAMSIM2

A. Application Description

```
int main()
{
  cout << "Loop Unrolling Test(Original)!" << endl;
  layer_t alex_conv_1(227, 227, 3, 11, 11, 96, 4, 1);
  layer_t &layer = alex_conv_1;

  cout << "Convolutional Layer Execution Start!" << endl;
  for(unsigned k = 0; k < layer.K; k++) {
    for(unsigned oh = 0; oh < layer.OH; oh++) {
      for(unsigned ow = 0; ow < layer.OW; ow++) {
        for(unsigned c = 0; c < layer.C; c++) {
          for(unsigned r = 0; r < layer.R; r++) {
            for(unsigned s = 0; s < layer.S; s++) {
              unsigned h = oh * layer.stride + r;
              unsigned w = ow * layer.stride + s;
              *layer.output(ow,oh,k)+=layer.input(w,h,c)*layer.filter(s,r,c,k);
            }
          }
        }
      }
    }
  }
  cout << "Convolutional Layer Execution Finish!" << endl;
  return 0;
}
```

- AlexNet Conv1 Layer(Original) -

```
  cout << "Convolutional Layer Execution Start!" << endl;
  for(unsigned k = 0; k < layer.K; k++) {
    for(unsigned oh = 0; oh < layer.OH; oh++) {
      for(unsigned ow = 0; ow < layer.OW; ow++) {
        for(unsigned c = 0; c < layer.C; c++) {
          for(unsigned r = 0; r < layer.R; r ++) {
            for(unsigned s = 0; s < layer.S; s += 4) {
              unsigned h = oh * layer.stride + r;
              unsigned w = ow * layer.stride + s;
              *layer.output(ow,oh,k)+=layer.input(w,h,c)*layer.filter(s+0,r,c,k);
              *layer.output(ow,oh,k)+=layer.input(w,h,c)*layer.filter(s+1,r,c,k);
              *layer.output(ow,oh,k)+=layer.input(w,h,c)*layer.filter(s+2,r,c,k);
              *layer.output(ow,oh,k)+=layer.input(w,h,c)*layer.filter(s+3,r,c,k);
            }
          }
        }
      }
    }
  }
```

- AlexNet Conv1 Layer(Unrolled) -

	Original		Unrolled	
sim_seconds	40.49874	>	39.35633	# Number of seconds simulated
sim_ticks	40,498,700,000,000	>	39,356,330,693,500	# Number of ticks simulated
system.mem_ctrls.bw_read::cpu0.inst	1,381	<	1,431	# Total read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_read::cpu0.data	1,542,780	<	1,587,772	# Total read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_read::total	1,544,161	<	1,589,203	# Total read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_inst_read::cpu0.inst	1,381	<	1,431	# Instruction read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_inst_read::total	1,381	<	1,431	# Instruction read bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_write::writebacks	76,387	<	78,604	# Write bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_write::total	76,387	<	78,604	# Write bandwidth from this memory (bytes/s)
system.mem_ctrls.bw_total::writebacks	76,387	<	78,604	# Total bandwidth to/from this memory (bytes/s)
system.mem_ctrls.bw_total::cpu0.inst	1,381	<	1,431	# Total bandwidth to/from this memory (bytes/s)
system.mem_ctrls.bw_total::cpu0.data	1,542,780	<	1,587,772	# Total bandwidth to/from this memory (bytes/s)
system.mem_ctrls.bw_total::total	1,620,548	<	1,667,808	# Total bandwidth to/from this memory (bytes/s)
system.cpu0.dtb.rdAccesses	11,461,623,462	>	11,337,041,862	# TLB accesses on read requests
system.cpu0.dtb.wrAccesses	2,973,804,875	<	2,983,388,075	# TLB accesses on write requests
system.cpu0.dtb.rdMisses	14,866	=	14,867	# TLB misses on read requests
system.cpu0.dtb.wrMisses	486	=	486	# TLB misses on write requests

Fig. 3: Simulation Results(Original vs Unrolled)

I use the first convolutional layer in AlexNet and compare this with an unrolled version. The original has 6 for-loop and each loop is increased by 1. On the other hand, the unrolled version's inner most loop is counted by 4(S: filters' width). Because of this, the unrolled version have 4 multiplication operations. The applications run on gem5 8 TimingSimpleCPU(X86) cores with command below.

```
build/X86/gem5.opt configs/example/se.py \
  --cmd=${EXE_FILE_PATH} \
  --cpu-type=TimingSimpleCPU --num-cpu=8 \
  --caches --l2cache \
  --num-l2caches=8 --l1d_size=32kB --l1i_size=32kB --l2_size=512kB \
  --mem-type=DRAMSim2 \
```

- gem5 Command Line -

B. Simulation Results and Analysis

Fig. 3 is results of the simulation. As expected, the unrolled test is finished faster than the original. The reason of the difference of cycles is that the unrolled loop encounters branch instructions less than original. At the *system.mem_ctrls.bw* parts in the results, all the unrolled outputs are larger than the original's. Since the requests are almost same for both and the the unrolled test's taken time is shorter than original's, the unrolled version memory bandwidth numbers are larger than original's.

V. CONCLUSION

I tried to attach *DRAMsim3* to *gem5*, however, some problems came out. At *dramsim3::BaseDRAMSystem::ResetStats()*, which resets output stats, segmentation fault signal is detected. After fixing this, *dramsim3::MemorySystem::ClockTick()* problem occured. Since I don't have enough time to debug this, I decided to use *DRAMsim2* which is more stable than *DRAMsim3*. While working on this project, I was able to learn how DRAM works and how hardware simulators work. In my future work, what I have learned from this project will be helpful a lot.

REFERENCES

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[2] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "Dramsim3: a cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, 2020.