Study Memory Behaviors using SIS Testbed

Xianwei Zhang
xianeizhang@cs.pitt.edu

Abstract

Main memory is the critical building block of computer systems, and it services the read/write requested from CPU. The request needs the coordination of CPU, memory controller and underlying memory devices, etc. While understanding memory behaviors is crucial for system researchers, it is ordinarily difficult for novices. This project uses SIS testbed to bridge the gap between high-level sense and low-level details.

With the finished project, we can easily visualize the general memory behaviors. In particular, we can observe the back propagation from memory to CPU, the detailed mapping policy and also the different memory latency effects (hit or miss), etc. The results positively show the powerfulness and generality of SIS testbed.

Video URL: https://youtu.be/4kUM0Tn_Drk

Germs of this project:

1. explored Computer Architecture stuff in multi-media scenario, with reasonable simplifications;
2. the work proved the powerfulness and generality of SIS Testbed, and services to bridge the gap between high-level sense and low-level details;
3. constructed FOUR new components, including one super component and three common components;
4. the positive results faithfully repeat real-life facts to large degree;
5. implemented classical algorithms, including Address Mapping, Scheduling and time-driven simulation;
6. well-formatted video, with clear contents and brief notes;
7. video has been uploaded to Youtube with URL provided in this report;
8. report is well written with LaTex;
9. codes were formally written with clear comments, partial snippets are pasted in later sections.

1. Introduction

Requests, filtered through all cache hierarchy, will finally be serviced by underlying memory. As illustrated by Figure 1, the requests leaving from CPU are first buffered in the queue of memory controller (MC), which is a on-chip logic unit to schedule memory accesses to off-chip physical memory devices. Memory controller follows predefined policy, like First-come-first-serve (*FCFS*), to issue the buffered requests to memory devices.

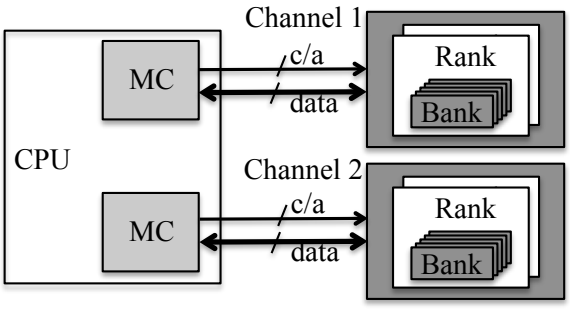


Figure 1: Memory logical hierarchy.

Generally, main memory is composed of multiple **channel**s, which are working independently with each other. Each channel has a on-chip MC to send requests to and receive data from off-chip devices via command/address bus and data bus. In terms of memory, each channel contains one or more **rank**s, and each rank typically consists of eight **bank**s. Further, each bank is made up of 2D arrays, with data being uniquely located with **row** and **column**.

When MC receives requests, it first inserts them into the queue, and then follows certain scheduling policy to pick up one to issue. Next, MC chops the numeric address into the tuple of **(channel, rank, bank, row, column)**, and then forwards the request to the target bank when the bank is available. As a performance optimization, each bank provides a row buffer to cache the last opened row, and thus if the target row matches that of last opened, then the access is a row buffer hit which has a much shorter latency than normal miss case.

Table 1: System Configuration

CPU	4 Ghz, IPC=1.0, MPKI=100 (i.e., one memory request per 2.5ns) reads:writes=2:1, address is randomly generated stalled when MC queue is full, resume when half full
Memory Controller	Queue capacity: 48 entries Low watermark: 24 entries Scheduling: FCFS Address mapping: rw:rk:bk:ch:cl:offset
Memory	capacity: 128MB (i.e., #address bits = 17) 1channel, 1rank/channel, 2banks/rank, 16K rows/bank, 4KB/row, 64B cache line Latency: hit=10ns, miss=50ns

Given the superior complexity in real-system, this project makes the following reasonable simplifications and assumptions:

1. main memory is composed of one single rank, and the rank contains two banks only;
2. memory controller has a single queue to hold both reads and writes to the two banks;
3. the queue has 48 entries, full queue stalled CPU, otherwise CPU periodically generate requests;
4. the scheduling policy is *FCFS*.

The detailed configurations are as shown in Table 1.

2. Design and Implementation

2.1. Design

To use SIS testbed to study memory behaviors, I adopt the design as shown in Figure 2. Generally, four components are involved, including `CPU`, `Memory Controller`, `Bank 0` and `Bank 1`. Among them, `Memory Controller` is the super components. Communications between components are achieved via messages, whose type can be alert, emergency, etc.

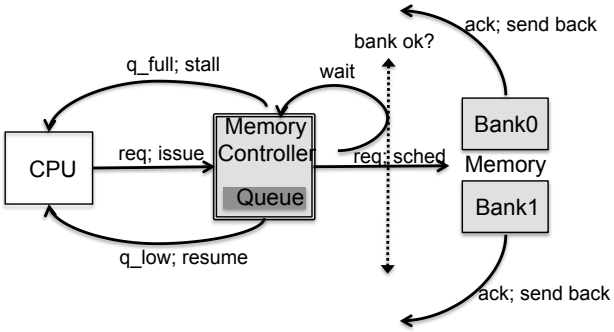


Figure 2: Overall design. Four components are constructed. `Memory Controller` **is the super component, and the others are common components.**

Normally, CPU periodically generate requests, which are then issued to MC. Upon receiving the requests, MC puts them into the queue, and then check whether the queue is fully occupied. If queue is full, then MC

immediately sends an alert back to CPU to make it stall, and thereafter MC drains the existing requests staying in the queue; when the queue occupancy reaches the low watermark (i.e. half full), MC alerts the CPU to resume.

To schedule requests downward to memory, MC picks up one request from the queue, and extracts the address IDs in terms of **bank** and **row**, etc. Now, MC is aware of the target bank and row, and it can decide the proper time to send out the request, and also the duration to service the request. Note that request cannot be issued until the target bank is idle, and the access latency varies depending on row hit or miss. When receiving requests from MC, banks start to work on data fetch or store. Upon finish, the banks broadcast an alert.

2.2. Implementation details

2.2.1. primary data structures Memory controller queue is implemented using LinkedList to enable FCFS scheduling. Besides, an separate variable current_q_size is used to denote the queue occupancy. Queue capacity, low watermark and full flag are also provided to decide when to communicate with CPU. MC decides everything for the request scheduling and service, and memory is completely dumb. Array variables NEXT_TIMES and OPENED_ROWS are used to update bank available time, and denote the last opened row of each bank. With these two arrays, MC can safely schedule the requests and also decide the access latency of each request. In addition, a private class is utilized to hold the address IDs.

```
static LinkedList<String> reqQueue =  new LinkedList<String>(); //MC request queue
static int current_q_size = 0;  //current occupancy

static int REQ_SIZE = 48;       //q capacity: #entries
static int REQ_LOW = 24;        //low watermark to resume CPU
static int full_flag = 0;       //flag to indicate q is full

//total memory capacity = 1*1*2*16384*4KB = 128MB
static int NUM_CHANNELS = 1;    //#channels
static int NUM_RANKS = 1;       //#ranks
static int NUM_BANKS = 2;       //#banks
static int NUM_ROWS = 16384;    //#rows
static int NUM_COLUMNS = 64;    //#columns, i.e., 4KB/64B
static int CACHE_LINE_SIZE = 64;//cache line size

static int[] OPENED_ROWS = new int[NUM_BANKS];//last opened row of each bank
static long HIT_LAT = 10*100;   //hit latency
static long MISS_LAT = 50*100;  //miss latency
static long[] NEXT_TIMES = new long[NUM_BANKS];//next available time of each bank

//class to store addr IDs
private static class Addr_IDs{
    int chID = -1;//channel
    int rkID = -1;//rank
    int bkID = -1;//bank
    int rwID = -1;//row
    int clID = -1;//col
}

static Addr_IDs my_addr_ids = new Addr_IDs(); //obj to store addr IDs
```

Figure 3: Data structures.

2.2.2. Scheduling Algorithm With the aforementioned data structures, we then move on to scheduling algorithm. For simplicity, FCFS is used and thus the queue head is always serviced. Each time, the algorithm gets the first entry of the queue, and then chops the address into a series of IDs. MC then checks the target bank ID to decide whether issue is allowed. If the target bank is still busy, then just wait. If the bank is ready, then compare the target row against last opened row to determine a row hit or miss. Finally, OPENED_ROWS and NEXT_TIMES are updated for future scheduling.

```
if(reqQueue.size() == 0) //empty q
    return;

String type_addr =  reqQueue.get(0); //FCFS: fetches first request

if(full_flag==1 && current_q_size <= REQ_LOW){//just now FULL, now LOW
    full_flag = 0;
    //Notify CPU to resume
    emergency.putPair("Receiver", "CPU");
    emergency.putPair("MainComponent", NAME);

    emergency.putPair("Purpose", "MC_queue_low");
    encoder.sendMsg(emergency);
    System.out.println("Warning: queue is low !!!");
}

/*PARSE THE REQUEST TO GET ADDRESS*/

calc_dram_addr(addr);//extract addr IDs

int bankID = my_addr_ids.bkID;//bank ID
int rowID = my_addr_ids.rwID;//row ID

if(System.currentTimeMillis() < NEXT_TIMES[bankID]){//target bank is still busy, wait
    return;
}

//target bank is ready now
reqQueue.removeFirst();
current_q_size --;

//latency: hit or miss
boolean rowbuffer_hit = (rowID == OPENED_ROWS[bankID]) ? true : false;
long latency = rowbuffer_hit ? HIT_LAT : MISS_LAT;

/*SENDS MESSAGE TO THE TARGET BANK*/

OPENED_ROWS[bankID] = rowID;//record the opened row
NEXT_TIMES[bankID] = System.currentTimeMillis() + latency;//next bank available time
```

Figure 4: Scheduling algorithm.

2.2.3. Address Mapping Algorithm Original address from CPU is a numeric value in the range of $[0, 128M)$. Memory controller needs to translate the value into address IDs in terms of rank, bank and row, etc. Multiple candidate mapping policies can be used to chop the address, and this project adopts the row-hit-friendly one `row:rank:bank:channel:column:offset`. To extract the corresponding bits for each part, bit manipulation is used, with detailed algorithm shown in Figure 5.

```
int chBitWidth = log_base2(NUM_CHANNELS);   //channel bit width
int rkBitWidth = log_base2(NUM_RANKS);      //rank bit width
int bkBitWidth = log_base2(NUM_BANKS);      //bank bit width
int rwBitWidth = log_base2(NUM_ROWS);       //row bit width
int clBitWidth = log_base2(NUM_COLUMNS);    //column bit width
int offsetWidth = log_base2(CACHE_LINE_SIZE);//offset bit width

int input_a, temp_a, temp_b;
input_a = addr;                     //rw:rk:bk:ch:cl:of
input_a = input_a >> offsetWidth;   //shift out offset --> rw:rk:bk:ch:cl

temp_a = input_a;                   //rw:rk:bk:ch:cl
input_a = input_a >> clBitWidth;    //rw:rk:bk:ch
temp_b = input_a << clBitWidth;     //rw:rk:bk:ch:cl:00
my_addr_ids.clID = temp_a ^ temp_b; //XOR: 00:00:00:00:cl

temp_a = input_a;                   //rw:rk:bk:ch
input_a = input_a >> chBitWidth;    //rw:rk:bk
temp_b = input_a << chBitWidth;     //rw:rk:bk:ch:00
my_addr_ids.chID = temp_a ^ temp_b; //XOR: 00:00:00:ch

temp_a = input_a;                   //rw:rk:bk
input_a = input_a >> bkBitWidth;    //rw:rk
temp_b = input_a << bkBitWidth;     //rw:rk:00
my_addr_ids.bkID = temp_a ^ temp_b; //XOR: 00:00:bk

temp_a = input_a;                   //rw:rk
input_a = input_a >> rkBitWidth;    //rw
temp_b = input_a << rkBitWidth;     //rw:00
my_addr_ids.rkID = temp_a ^ temp_b; //XOR: 00:rk

temp_a = input_a;                   //rw
input_a = input_a >> rwBitWidth;    //
temp_b = input_a << rwBitWidth;     //00
my_addr_ids.rwID = temp_a ^ temp_b; //XOR: rw
```

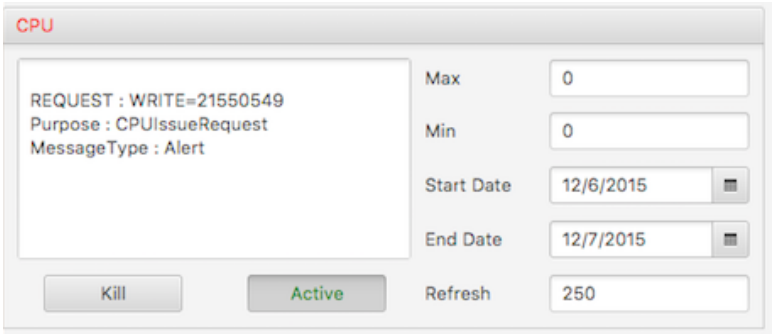
Figure 5: Address mapping.

3. Results

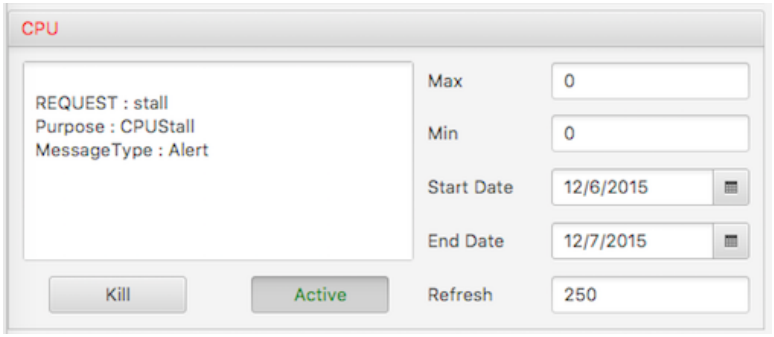
Normally, CPU periodically generate request to access memory. Following real system, reads are generate roughly two times of writes, and I assume $IPC = 1$ and $MPKI = 100$, meaning that CPU executes one instruction per clock cycle and one hundred out of 1000 instructions access memory. Thus, one request will be generated every 10 cycles (CPU frequency is 4G Hz), translating into 2.5ns. For ease of observation in experiments, treat 250ms as 2.5ns, and the same amplification applies to memory latency values.

3.1. CPU behaviors

As Figure 6 illustrates, CPU experiences two states, **active** and **stall**. The state transition is driven by the queue occupancy of MC queue. When queue becomes full, an alert is sent to CPU to stall; and then MC drains the requests to decrease queue occupancy; when queue is half full, another alert will be sent to CPU to resume.



(a) CPU is active



(b) CPU is stalled

Figure 6: Periodical behaviors of CPU.

3.2. Memory controller behaviors

Memory controller has a queue of 48 entries to store requests from CPU. And it follows FCFS scheduling policy to forward the requests to underlying memory banks. MC is the master of all banks, and it maintains the bank states, including last opened row and the bank next available time. When queue becomes fully occupied, MC alerts the CPU to temporally stall request generation. While CPU is being stalled, MC drains the queue to free space. Later, when queue reaches the predefined low watermark (half full), another alert is sent to CPU to let it resume.

```
Need to wait 0 for [224]ns...
READ 14779303 to Bank0
Warning: queue is low !!!
READ 68505667 to Bank1
Need to wait 1 for [4999]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [4750]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [4497]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [4242]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [3988]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [3736]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [3481]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [3228]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [2974]ns...
CPUtoMemoryController received, start processing...
Need to wait 1 for [2720]ns...
```

(a) Process requests following FCFS scheduling policy.

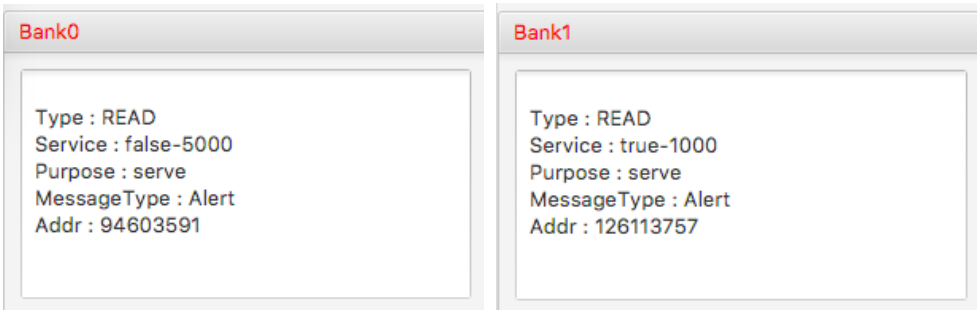
```
from MemoryController: MC_queue_full
from MemoryController: MC_queue_low
from MemoryController: MC_queue_full
from MemoryController: MC_queue_low
from MemoryController: MC_queue_full
from MemoryController: MC_queue_low
from MemoryController: MC_queue_full
from MemoryController: MC_queue_low
from MemoryController: MC_queue_full
```

(b) Queue periodically full and low.

Figure 7: Memory controller behaviors.

3.3. Bank behaviors

Memory banks are dumb components. They just receive the requests from memory controller, and returns data or acknowledges within the timing goals set by MC. Upon finish the request, banks broadcast the information as an alert message.



(a) Request has a row miss. (b) Request has a row hit.

Figure 8: Banks service request as a hit or miss.

4. Summary

This project visualizes the memory behaviors using SIS testbed. With the finished project, we can observe the complete travel of a memory request, from leaving CPU to being buffered into MC queue, and next being scheduled by MC and finally serviced in underlying memory banks. Given the fact that CPU quickly feeds request to MC, and the queue is very limited (48 entries) and memory is generally much slower, the queue can be quickly fully occupied. Further, a back pressure is set on CPU, which stalls as a response. Afterwards, MC drains the queue to free some space and let the CPU resume. Consequently, we can observe periodical behaviors of both CPU and memory.

The demo video can be found at https://youtu.be/4kUM0Tn_Drk.