USIMM: the Utah SImulated Memory Module

A Simulation Infrastructure for the JWAC Memory Scheduling Championship

Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H. Pugsley, Aniruddha N. Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, Zeshan Chishti [‡]

UUCS-12-002

University of Utah and [‡] Intel Corp.

February 20, 2012

Abstract

USIMM, the Utah SImulated Memory Module, is a DRAM main memory system simulator that is being released for use in the Memory Scheduling Championship (MSC), organized in conjunction with ISCA-39. MSC is part of the JILP Workshops on Computer Architecture Competitions (JWAC). This report describes the simulation infrastructure and how it will be used within the competition.

1 Introduction

The Journal of Instruction Level Parallelism (JILP) organizes an annual Workshop on Computer Architecture Competitions (JWAC). The 2012 competition is a Memory Scheduling Championship (MSC) that will be held in conjuction with ISCA-39 in Portland, Oregon.

The memory sub-system is an important component in all computer systems, accounting for a significant fraction of execution time and energy. When a processor core fails to find relevant data in processor caches, it sends a request to the memory controller on the processor chip. The memory controller is connected to DRAM chips on DIMMs, via a memory channel. The memory controller manages a queue of pending memory requests and schedules them, subject to various timing constraints. The memory latency is the sum of queuing delay and the time to fetch a cache line from the DRAM chip row buffer. Many studies have shown that the queuing delay is a significant component of memory access time and it is growing as more processor cores share a memory channel [6, 10, 14, 22, 23]. The queuing delay for a request is heavily impacted by the memory scheduling algorithm. The memory scheduling algorithm can determine the wait time for individual requests, the efficiency of the row buffer, the efficiency in handling writes, parallelism across ranks and banks, fairness across threads, parallelism within threads, etc. There are many considerations in designing memory schedulers and some recent algorithms have significantly advanced the field [10, 14, 21–23]. The Memory Scheduling Championship hopes to provide a forum to not only compare these many algorithms within a common evaluation framework, but also stimulate research in an important area.

This report describes the common simulation infrastructure in detail. It describes the modules within the simulator code, the traces used to represent workloads, and the evaluation metrics that will be used for the competition. It also provides some guidelines that will help contestants develop efficient scheduling algorithms. We expect that the simulation infrastructure will continue to be used beyond the competition, and will facilitate research and education in the area. The Utah Arch group will continue to release extensions of the simulator.

2 Memory System Overview

Memory Basics. In a conventional memory system, a memory controller on the processor is connected to dual in-line memory modules (DIMMs) via an off-chip electrical memory channel. Modern processors have as many as four memory controllers and four DDR3 memory channels [19, 31]. While other memory channel and memory controller topologies exist (such as Intel's SMB [25]), our infrastructure is restricted to schedulers for DDR3 channels and DIMMs.

A modern DDR3 channel typically has a 64-bit data bus and a 23-bit address/command bus that can support 1-2 DIMMs. Each DIMM is typically organized into 1-4 ranks. When the memory controller issues a request for a cache line, all DRAM chips in a rank work together to service the request, i.e., a cache line is striped across all chips in a rank. A rank and its constituent DRAM chips are also partitioned into multiple (4-16) banks. Each bank can process a different cache line request in parallel, but all banks in the active rank must sequentially share the data and command wires of the memory channel.

Row Buffer Management. Each bank has a row buffer storing the last row accessed within the bank. If a request can be serviced by data already present in an open row buffer, the row buffer hit takes less time and energy. If a row buffer is unlikely to yield future hits, it is beneficial to close the row and precharge the bitlines so that the bank can quickly access a new row when required. Many commercial memory controllers use proprietary algorithms to decide when a row must be closed; several patents and papers construct mechanisms for prediction-based row closure [9, 12, 20, 24, 26, 27, 29, 33].

Address Mapping. A cache line is placed entirely in one bank. The next cache line could be placed in the same row, or the next row in the same bank, or the next bank in the same rank, or in the next rank in the same channel, or in the next channel. The data mapping policy determines the extent of parallelism that can be leveraged within the memory system. The MSC focuses on two different processor-memory configurations (more details in Section 4); each uses a different data mapping policy. The first configuration (*1channel*, with ADDRESS_MAPPING set to 1) tries to maximize row buffer hits and places consecutive cache lines in the same row, i.e., the lower-order bits pick different columns in a given row. The address bits are interpreted as follows, from left (MSB) to right (LSB):

$$1channel\ mapping\ policy :: row : rank : bank : channel : column : blockoffset$$

The second configuration (*4channel*, with ADDRESS_MAPPING set to 0) tries to maximize memory access parallelism by scattering consecutive blocks across channels, ranks, and banks. The address bits are interpreted as follows:

$$4channel\ mapping\ policy :: row : column : rank : bank : channel : blockoffset$$

Memory Commands. In every cycle, the memory controller can either issue a command that advances the execution of a pending read or write, or a command that manages the general DRAM state. The four commands corresponding to a pending read or write are:

- **PRE:** Precharge the bitlines of a bank so a new row can be read out.
- **ACT:** Activate a new row into the bank's row buffer.

- **COL-RD:** Bring a cache line from the row buffer back to the processor.
- **COL-WR:** Bring a cache line from the processor to the row buffer.

The six general "at-large" commands used to manage general DRAM state and not corresponding to an entry in the read or write queues are:

- **PWR-DN-FAST:** Power-Down-Fast puts a rank in a low-power mode with quick exit times. This command can put the rank into one of two states - active power down or precharge power down (fast). If all the banks in the DRAM chip are precharged when the PWR-DN-FAST command is applied, the chip goes into the precharge power down mode. However, if even a single bank has a row open, the chip transitions into the active power down mode. The power consumption of the active power down mode is higher than that of the precharge power down mode. In both these states, the on-chip DLL is active. This allows the chip to power-up with minimum latency. To ensure transition into the lower power state, it may be necessary to first precharge all banks in the rank (more on this below).
- **PWR-DN-SLOW:** Power-Down-Slow puts a rank in the precharge power down (slow) mode and can only be applied if all the banks are precharged. The DLL is turned off when the slow precharge power down mode is entered, which leads to higher power savings, but also requires more time to transition into the active state.
- **PWR-UP:** Power-Up brings a rank out of low-power mode. The latency of this command (i.e., the time it takes to transition into the active state) is dependent on the DRAM state when the command is applied (fast or slow exit modes). If the chip is in the active power down mode, it retains the contents of the open row-buffer when the chip is powered up. When the rank is powered down, all pending requests to that rank in the read and write queue note that their next command must be a PWR-UP. Thus, picking an instruction from the read or write queues will automatically take care of the power-up and an at-large power-up command (similar to a PWR-DN-FAST or PWR-DN-SLOW) is not required. Similarly, refresh operations will automatically handle the exit from the power-down mode.
- **Refresh:** Forces a refresh to multiple rows in all banks on the rank. If a chip is in a power-down mode before the refresh interval, the rank is woken up by refresh.
- **PRE:** Forces a precharge to a bank (so the bank is ready for future accesses to new rows).
- **PRE-ALL-BANKS:** Forces a precharge to all banks in a rank. This is most useful when preparing a chip for a power down transition.

Timing parameter	Default value (cycles at 800MHz)	Description
tRCD	11	Row to Column command Delay. The time interval between row access and data ready at sense amplifiers.
tRP	11	Row Precharge. The time interval that it takes for a DRAM array to be precharged for another row access.
tCAS	11	Column Access Strobe latency. The time interval between column access command and the start of data return by the DRAM device(s). Also known as tCL.
tRC	39	Row Cycle. The time interval between accesses to different rows in a bank. tRC =tRAS +tRP.
tRAS	28	Row Access Strobe. The time interval between row access command and data restoration in a DRAM array. A DRAM bank cannot be precharged until at least tRAS time after the previous bank activation.
tRRD	5	Row activation to Row activation Delay. The minimum time interval between two row activation commands to the same DRAM device. Limits peak current profile.
tFAW	32	Four (row) bank Activation Window. A rolling time-frame in which a maximum of four-bank activations can be engaged. Limits peak current profile in DDR2 and DDR3 devices with more than 4 banks.
tWR	12	Write Recovery time. The minimum time interval between the end of a write data burst and the start of a precharge command. Allows sense amplifiers to restore data to cells.
tWTR	6	Write To Read delay time. The minimum time interval between the end of a write data burst and the start of a column-read command. Allows I/O gating to overdrive sense amplifiers before read command starts.
tRTP	6	Read to Precharge. The time interval between a read and a precharge command.
tCCD	4	Column-to-Column Delay. The minimum column command timing, determined by internal burst (prefetch) length. Multiple internal bursts are used to form longer burst for column reads. tCCD is 2 beats (1 cycle) for DDR SDRAM, and 4 beats (2 cycles) for DDR2 SDRAM.
tRFC	128	Refresh Cycle time. The time interval between Refresh and Activation commands.
tREFI	6240	Refresh interval period.
tCWD	5	Column Write Delay. The time interval between issuance of the column-write command and placement of data on the data bus by the DRAM controller.
tRTRS	2	Rank-to-rank switching time. Used in DDR and DDR2 SDRAM memory systems; not used in SDRAM or Direct RDRAM memory systems. One full cycle in DDR SDRAM.
tPDMIN	4	Minimum power down duration.
tXP	5	Time to exit fast power down
tXPDLL	20	Time to exit slow power down
tDATATRANS	4	Data transfer time from CPU to memory or vice versa.

Table 1: DRAM timing parameters for default memory system configuration [11].

Timing Parameters. Only some of the above commands can be issued in a given cycle, depending on the current state of the ranks, banks, channel, and several timing parameters. The timing parameters listed in Table 1 are honored by USIMM. The values in Table 1 are typical of many Micron DDR3 chips, with only the tRFC parameter varying as a function of chip capacity. Consider tWTR as an example timing parameter. The direction of the memory channel data bus must be reversed every time the memory system toggles between reads and writes. This introduces timing delays, most notably, the delay between a write and read to the same rank (tWTR). To reduce the effect of this delay, multiple writes are typically handled in succession before handling multiple reads in succession. Note that commands are not required to turn the bus direction; if sufficient time has elapsed after a write, a read becomes a candidate for issue.

DRAM Refresh. Every DRAM row must be refreshed within a 64 ms window while at a temperature under 85 degrees Celsius. The refresh process generally works as follows. The memory controller issues a Refresh command every $7.8\mu s$ (the $tREFI$ parameter). This command initiates a refresh to multiple rows in all banks on the channel. For a few hundred nano-seconds (the $tRFC$ parameter) after the refresh command, the DRAM chips are unavailable to service other commands. The JEDEC standard allows a refresh command to be delayed by up to 8 $tREFI$s, as long as the average rate of refresh commands is one per $tREFI$ [30]. USIMM models this by confirming that 8 refreshes are issued in every $8 \times tREFI$ time window, with full flexibility for refresh issue within that time window.

3 Simulator Design

3.1 Code Organization and Features

High-Level Overview. This section provides a detailed description of the USIMM code. USIMM has the following high-level flow. A front-end consumes traces of workloads and models a reorder buffer (ROB) for each core on the processor. Memory accesses within each ROB window are placed in read and write queues at the memory controller at each channel. Every cycle, the simulator examines each entry in the read and write queues to determine the list of operations that can issue in the next cycle. A scheduler function is then invoked to pick a command for each channel from among this list of candidate commands. This scheduler function is the heart of the MSC and is the code that must be written by MSC competitors. The underlying USIMM code is responsible for modeling all the correctness features: DRAM states, DRAM timing parameters, models for performance and power. The scheduler must only worry about performance/power features: heuristics to select commands every cycle such that performance, power, and fairness metrics are optimized. Once the scheduler selects these commands, USIMM updates DRAM state and marks instruction completion times so they can be eventually retired from the ROB.

Code Files. The code is organized into the following files:

- **main.c :** Handles the main program loop that retires instructions, fetches new instructions from the input traces, and calls update_memory( ). Also calls functions to print various statistics.
- **memory_controller.c :** Implements update_memory( ), a function that checks DRAM timing parameters to determine which commands can issue in this cycle. Also has functions to calculate power.
- **scheduler.c :** Function provided by the user to select a command for each channel in every memory cycle.
- **configfile.h memory_controller.h params.h processor.h scheduler.h utils.h utlist.h :** various header files.

Inputs. The main( ) function in file main.c interprets the input arguments and initializes various data structures. The memory system and processor parameters are derived from a configuration file, specified as the first argument to the program. Each subsequent argument represents an input trace file. Each such trace is assumed to run on its own processor core.

Simulation Cycle. The simulator then begins a long while loop that executes until all the input traces have been processed. Each iteration of the while loop represents a new *processor cycle*, possibly advancing the ROB. The default configuration files assume 3.2 GHz processor cores and 800 MHz DRAM channels, so four processor cycles are equivalent to a single memory bus cycle. Memory functions are invoked in processor cycles that are multiples of four.

Commit. The first operation in the while loop is the commit of oldest instructions in the pipelines of each core. Each core maintains a reorder buffer (ROB) of fixed size that houses every in-flight instruction in that core. For each core, the commit operation in a cycle attempts to sequentially retire all completed instructions. Commit is halted in a cycle when the commit width is reached or when an incomplete instruction is encountered. A commit width of 2 per processor cycle corresponds to a core IPC of 2 if the trace was devoid of memory operations. The simulated IPC for most traces will be much less than 2.

Checking for Readiness. The next operation in the while loop is a scan of every memory instruction in the read and write queues of the memory controller to determine what operation can issue in this cycle. A single memory instruction translates into multiple memory system commands (e.g., PRE, ACT, Column-Read). Our scan first computes what the next

command should be. Note that this changes from cycle to cycle based on the current row buffer contents, the low-power state, and whether a refresh is being performed. We also examine a number of DRAM timing parameters to determine if the command can issue in this cycle. In addition to examining the read and write queues, we also consider the list of general commands (refresh, power down/up, precharge) and determine if they can be issued.

Scheduling. Once a list of candidate memory commands for this cycle is determined by our scan, a schedule( ) function (in file schedule.c) is invoked. This is the heart of the simulator and the function that must be provided by contestants in the JWAC MSC. In each memory cycle, each memory channel is capable of issuing one command. Out of the candidate memory commands, the schedule function must pick at most one command for each channel. Once a command has been issued, other commands that were deemed "ready for issue in this cycle" to the same channel will be rejected in case the scheduler tries to issue them. While each channel is independently scheduled, some co-ordination among schedulers may be beneficial [13].

Scheduling Algorithm Constraints. To qualify for the MSC, the user's scheduling algorithm should be implementable within a few processor cycles and with a hardware storage budget not exceeding 68 KB. Details must be provided in a 6-page paper that is submitted with the code. A program committee will evaluate the implementability of the algorithm, among other things.

Instruction Completion Times. Once the scheduler selects and issues commands, the simulator updates the state of the banks and appropriately sets the completion time for the selected memory instructions. This eventually influences when the instruction can be retired from the ROB, possibly allowing new instructions to enter the processor pipeline.

Advancing the Trace and Trace Format. Next, new instructions are fetched from the trace file and placed in the ROB. Memory instructions are also placed in the read and write queues. This process continues until either the ROB or write queues are full, or the fetch width for the core is exhausted. The trace simply specifies if the next instruction is a memory read (R), memory write (W), or a non-memory instruction (N). In case of memory reads and writes, a hexadecimal address is also provided in the trace. For the MSC, we assume that a trace can only address a 4 GB physical address space, so the trace is limited to 32-bit addresses. Memory writes do not usually correspond to actual program instructions; they refer to evictions of dirty data from cache. As a simplification, we assume that each line in the trace corresponds to a different program instruction. Note that this is an approximation not just because of cache evictions, but because some x86 instructions

correspond to multiple memory operations and the traces will occasionally include memory accesses to fetch instructions (and not just data).

Fetch Constraints and Write Drains. We assume that non-memory (N) and memory-write (W) instructions finish instantaneously, i.e., they are never bottlenecks in the commit process. Memory-writes will hold up the trace only when the write queue is full. To prevent this, it is the responsibility of the scheduler to periodically drain writes. Memory-reads are initially set to complete in the very distant future. The schedule function will later determine the exact completion time and update it in the ROB data structure. We do not model an explicit read queue size. The typical length of the read queue is determined by the number of cores, the size of the ROB, and the percentage of memory reads in a ROB. In other words, we assume that the read queue is not under-provisioned, relative to other processor parameters. The write queue on the other hand does need a capacity limit in our simulator since a write queue entry need not correspond to a ROB entry.

Refresh Handling. The simulator ensures that in every $8 \times tREFI$ window, all DRAM chips on a channel are unavailable for time $8 \times tRFC$, corresponding to eight Refresh operations. If the user neglects to issue eight refreshes during the $8 \times tREFI$ time window, USIMM will forcibly issue any remaining refreshes at the end of the time window. During this refresh period, the memory channel is unavailable to issue other commands. Each cycle, the simulator calculates a refresh deadline based on how many refreshes are pending for that window and eventually issues the required number of refreshes at the deadline. In order to ensure that the refresh deadline is not missed, the simulator marks a command ready only if issuing it does not interfere with the refresh deadline. So, when the refresh deadline arrives, the DRAM chip will be inactive (i.e., the banks will be precharged and in steady state or some rows will be open but with no on-going data transfer). The rank may also be in any of the power-down modes, in which case, it will be powered up by the auto refresh mechanism - the user does not need to issue the power-up command explicitly. At the end of the refresh period, all banks are in a precharged, powered-up state.

Implicit Scheduling Constraints. It is worth noting that the simulator design steers the user towards a greedy scheduling algorithm, i.e., the user is informed about what can be done in any given cycle and the user is prompted to pick one of these options. However, as we show in the example below, the user must occasionally not be tempted by the options presented by the simulator. Assume that we are currently servicing writes. A read can only be issued if time $tWTR$ has elapsed since the last write. Hence, following a write, only writes are presented as options to the memory scheduler. If the user schedules one of these writes, the read processing is delayed further. Hence, at some point, the scheduler must refrain from issuing writes so that time $tWTR$ elapses and reads show up in the list of candidate commands in a cycle.

3.2 Example Schedulers

As part of the USIMM distribution, we are releasing a few sample baseline scheduler functions. All of these functions were written in the matter of hours by graduate students in the Utah Arch group. These functions will make it easier for users to get started with their scheduling algorithms. We next describe these example memory scheduling algorithms. Note that these algorithms are not yet optimized and not suitable as "baselines" in papers. However, such baseline scheduling algorithms will eventually be released on the USIMM webpages. As users explore their own scheduling algorithms, they are advised to consider the features suggested in several recent papers on memory scheduling (for example: [10, 14, 21–23]).

FCFS, scheduler-fcfs.c : True FCFS, i.e., servicing reads in the exact order that they arrive and stalling all later reads until the first is done, leads to very poor bank-level parallelism and poor bandwidth utilization. We therefore implement the following variant of FCFS. Assuming that the read queue is ordered by request arrival time, our FCFS algorithm simply scans the read queue sequentially until it finds an instruction that can issue in the current cycle. A separate write queue is maintained. When the write queue size exceeds a high water mark, writes are drained similarly until a low water mark is reached. The scheduler switches back to handling reads at that time. Writes are also drained if there are no pending reads.

Credit-Fair, scheduler-creditfair.c : For every channel, this algorithm maintains a set of counters for credits for each thread, which represent that thread's priority for issuing a read on that channel. When scheduling reads, the thread with the most credits is chosen. Reads that will be open row hits get a 50% bonus to their number of credits for that round of arbitration. When a column read command is issued, that thread's total number of credits for using that channel is cut in half. Each cycle all threads gain one credit. Write queue draining happens in an FR-FCFS manner (prioritizing row hits over row misses). The effect of this scheduler is that threads with infrequent DRAM reads will store up their credits for many cycles so they will have priority when they need to use them, even having priority for infrequent bursts of reads. Threads with many, frequent DRAM reads will fairly share the data bus, giving some priority to open-row hits. Thus, this algorithms tries to capture some of the considerations in the TCM scheduling algorithm [14].

Power-Down, scheduler-pwrdn.c : This algorithm issues PWR-DN-FAST commands in every idle cycle. Explicit power-up commands are not required as power-up happens implicitly when another command is issued. No attempt is made to first precharge all banks to enable a deep power-down.

Close-Page, scheduler-close.c : This policy is an approximation of a true close-page policy. In every idle cycle, the scheduler issues precharge operations to banks that last serviced a column read/write. Unlike a true close-page policy, the precharge is not issued immediately after the column read/write and we don't look for potential row buffer hits before closing the row.

First-Ready-Round-Robin, scheduler-frrr.c: This scheduler tries to combine the benefits of open row hits with the fairness of a round-robin scheduler. It first tries to issue any open row hits with the "correct" thread-id (as defined by the current round robin flag), then other row hits, then row misses with the "correct" thread-id, and then finally, a random request.

MLP-aware, scheduler-mlp.c: The scheduler assumes that threads with many outstanding misses (high memory level parallelism, MLP) are not as limited by memory access time. The scheduler therefore prioritizes requests from low-MLP threads over those from high-MLP threads. To support fairness, a request's wait time in the queue is also considered. Writes are handled as in FCFS, with appropriate high and low water marks.

3.3 Power Model

The simulator also supports a power model. Relevant memory system statistics are tracked during the simulation and these are fed to equations based on those in the Micron power calculator [1].

Memory Organizations. The power model first requires us to define the type of memory chip and rank organization being used. The input configuration file (more details on this in Section 4) specifies the number of channels, ranks, and banks. This organization is used to support a 4 GB address space per core. As more input traces are provided, the number of cores and the total memory capacity grows. Accordingly, we must figure out the memory organization that provides the required capacity with the specified channels and ranks. For example, for the 1channel.cfg configuration and 1 input trace file, we must support a 4 GB address space with 1 channel and 2 ranks. Each rank must support 2 GB, and we choose to do this with 16 x4 1 Gb DRAM chips. If 1channel.cfg is used with 2 input trace files, we support an 8 GB address space with the same configuration by instead using 16 x4 2 Gb DRAM chips. For the MSC, we restrict ourselves to the configurations in Table 2. USIMM figures out this configuration based on the input system configuration file and the number of input traces. It then reads the corresponding power and timing parameters for that DRAM chip from the appropriate file in the input/ directory (for example, 1Gb_x4.vi). The only timing parameter that shows variation across DRAM chips is tRFC.

System config file	Channels and Ranks per Channel	Number of cores	Memory Capacity	Organization of a rank
1channel.cfg	1 ch, 2 ranks/ch	1	4 GB	16 x4 1 Gb chips
1channel.cfg	1 ch, 2 ranks/ch	2	8 GB	16 x4 2 Gb chips
1channel.cfg	1 ch, 2 ranks/ch	4	16 GB	16 x4 4 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	1	4 GB	4 x16 1 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	2	8 GB	8 x8 1 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	4	16 GB	16 x4 1 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	8	32 GB	16 x4 2 Gb chips
4channel.cfg	4 ch, 2 ranks/ch	16	64 GB	16 x4 4 Gb chips

Table 2: Different memory configurations in our power model.

While the simulator can support more than 16 traces with 4channel.cfg and more than 4 traces with 1channel.cfg, the power model does not currently support models other than those in Table 2. The different allowed DRAM chips and the power parameters for each are summarized in Table 3.

Parameter	**1Gb x4**	**1Gb x8**	**1Gb x16**	**2Gb x4**	**2Gb x8**	**4Gb x4**	**4Gb x8**
VDD	1.5	1.5	1.5	1.5	1.5	1.5	1.5
IDD0	70	70	85	42	42	55	55
IDD2P0	12	12	12	12	12	16	16
IDD2P1	30	30	30	15	15	32	32
IDD2N	45	45	45	23	23	28	28
IDD3P	35	35	35	22	22	38	38
IDD3N	45	45	50	35	35	38	38
IDD4R	140	140	190	96	100	147	157
IDD4W	145	145	205	99	103	118	128
IDD5	170	170	170	112	112	155	155

Table 3: Voltage and Current parameters of chips used [3–5]

Power Equations. The power equations are as follows and are based on equations in the Micron power calculator [1] and the Micron Memory System Power Technical Note [18]:

$$ReadPower = (I_{DD4R} - I_{DD3n}) * V_{DD} * \%Cycles\,when\,data\,is\,being\,Read$$

$$WritePower = (I_{DD4W} - I_{DD3n}) * V_{DD} * \%Cycles\,when\,data\,is\,being\,Written$$

$$RefreshPower = (I_{DD5} - I_{DD3n}) * V_{DD} * T_{RFC}/T_{REFI}$$

$$ActivatePower = Max.\,Activate\,Power * T_{RC}/(Average\,gap\,between\,ACTs)$$

$$Max.\,Activate\,Power = ((I_{DD0} - (I_{DD3N} * T_{RAS} + I_{DD2N} * (T_{RC} - T_{RAS}))/T_{RC}) * V_{DD})$$

Background Power is the combination of many components. These components are listed below

$$act\_pdn = I_{DD3P} * V_{DD} * \%(Time\ Spent\ in\ PowerDown\ with\ atleast\ one\ Bank\ Active)$$

$$act\_stby = I_{DD3N} * V_{DD} * \%(Time\ Spent\ in\ Active\ Standby)$$

$$pre\_pdn\_slow = I_{DD2P0} * V_{DD} * \%(Time\ Spent\ in\ PreCharge\ Powerdown\ Slow\ Mode)$$

$$pre\_pdn\_fast = I_{DD2P1} * V_{DD} * \%(Time\ Spent\ in\ PreCharge\ Powerdown\ Fast\ Mode)$$

$$pre\_stby = IDD2N * V_{DD} * \%(Time\ Spent\ in\ Standby\ with\ all\ Banks\ PreCharged)$$

Finally,

$$Background\ Power = act\_pdn + act\_stby + pre\_pdn\_slow + pre\_pdn\_fast + pre\_stby$$

Power dissipated in the ODT resistors is called the Termination Power. Termination Power not only depends on the activity in the Rank in question but also depends on the activity in other Ranks on the same channel. Power dissipated due to Reads and Writes terminating in the Rank in question is given by

$$ReadTerminate = pds\_rd * \%Cycles\ when\ data\ is\ being\ Read\ from\ this\ Rank$$

$$WriteTerminate = pds\_wr * \%Cycles\ when\ data\ is\ being\ Written\ to\ this\ Rank$$

$$ReadTerminateOther = pds\_termRoth *$$
$$\%Cycles\ when\ data\ is\ being\ Read\ from\ other\ Ranks$$

$$WriteTerminateOther = pds\_termWoth *$$
$$\%Cycles\ when\ data\ is\ being\ Written\ to\ other\ Ranks$$

We use the same rank configuration as assumed in the Micron Technical Note [18], hence we assume the same ODT power dissipation. The values of pds_rd, pds_wr, $pds\_termRoth$, $pds\_termWoth$ are taken from the Micron Technical Note [18].

$$Total\ Chip\ Power = ReadPower + WritePower + ActivatePower +$$
$$BackGroundPower + RefreshPower + ReadTerminatePower +$$
$$WriteTerminatePower + ReadTerminateOther + WriteTerminateOther$$

The above DRAM chip power must be multiplied by the number of DRAM chips to obtain total memory system power.

System Power Model. When computing energy-delay-product (EDP), we must multiply system power with the square of system execution time. For our 4-channel configuration, we assume that our system incurs 40 W of constant power overheads for processor uncore components, I/O, disk, cooling, etc. Each core (including its private LLC) incurs a power overhead of 10 W while a thread is running and 0 W (perfect power gating) when a thread has finished. The rest comes from the memory system, with the detailed estimation described above. Our 1-channel configuration is supposed to represent a quarter of a future many-core processor where each channel on the processor chip will be shared by many simple cores. Consequently, our system power estimate with this configuration assumes only 10 W for miscellaneous system power (a total 40 W that is divided by 4) and 5 W peak power per core (since the core is simpler). Similar to the 4-channel configuration, a core is power-gated once a thread has finished executing. In either system power model, the memory system typically accounts for 15-35% of total system power, consistent with many reported server power breakdowns [7, 8, 15–17].

4 Workloads and Simulator Execution

The USIMM simulator can take multiple workload traces as input. Each workload trace represents a different program running on a different core, with memory accesses filtered through a 512 KB private LLC. The JWAC MSC will later construct and release a specific set of workload traces, including commercial workload traces, that will be used for the competition. The initial USIMM distribution has a few short traces from single-thread executions of the PARSEC suite that can be used for testing.

The simulator is executed with multiple arguments. The first argument specifies the configuration file for the processor and memory system. The remaining arguments each specify an input trace file. As many cores are assumed as the number of input trace files. The traces only contain the instruction types and memory addresses accessed by a program, but no timing information (the timing is estimated during the simulation). Based on the address being touched by a memory instruction, the request is routed to the appropriate memory channel and memory controller.

Some of the traces are derived with publicly available benchmarks. These benchmarks are executed with Windriver Simics [2] and its g-cache module to produce the trace. Some of the traces are derived from commercial workloads. To keep simulation times manageable, the traces released for the JWAC MSC will be for a few million instructions, but representative of behavior for a longer execution.

Each thread's trace is restricted to a 4 GB space. When multiple traces are fed to USIMM, the address space grows and each trace is mapped to its own 4 GB space within this address space. This is implemented by adding bits to the trace address (corresponding to the core id). These additional bits are interpreted as part of the row address bits. Thus, as more

cores are added, the DRAM chips are assumed to have larger capacities.

Modeling a shared cache would require us to pre-determine the threads that will share the cache. We therefore assume that each thread's trace is filtered through a private LLC. Since each core and trace is now independent, we can construct arbitrary multi-core workloads by feeding multiple traces to USIMM. When generating a trace for a multi-threaded application, we must confirm that a memory access is included in a thread's trace only after checking the private LLCs of other threads.

The JWAC MSC will focus on two main system configurations. The first uses a smaller scale processor core and a single memory channel, while the second uses a more aggressive processor core and four memory channels. The two configurations are summarized in Table 4 below, with the differences in bold. While a single channel appears underprovisioned by today's standards, it is more representative of the small channel-to-core ratio that is likely in future systems.

Parameter	**1channel.cfg**	**4channel.cfg**
Processor clock speed	3.2 GHz	3.2 GHz
Processor ROB size	128	**160**
Processor retire width	2	**4**
Processor fetch width	4	4
Processor pipeline depth	10	10
Memory bus speed	800 MHz (plus DDR)	800 MHz (plus DDR)
DDR3 Memory channels	1	**4**
Ranks per channel	2	2
Banks per rank	8	8
Rows per bank	32768 $\times$ NUMCORES	32768 $\times$ NUMCORES
Columns (cache lines) per row	128	128
Cache line size	64 B	64 B
Address bits (function of above params)	32+log(NUMCORES)	34+log(NUMCORES)
Write queue capacity	64	**96**
Address mapping	row:rank:bank:chnl:col:blkoff	**row:col:rank:bank:chnl:blkoff**
Write queue bypass latency	10 cpu cycles	10 cpu cycles

Table 4: System configurations used for the JWAC MSC.

The JWAC MSC has three competitive tracks: (1) Performance, (2) Energy-Delay Product (EDP), and (3) Performance-Fairness Product (PFP). The simulator reports the delay to finish each program, as well as the power consumed in each rank for the entire simulation. To measure the quality of a scheduling algorithm, it is executed for all workloads for both system configurations. The metric for each track is computed across all these simulations.

For the Performance track, delay for a scheduler is measured as the sum of execution times for all involved programs in all simulations.

EDP for a scheduler is measured as the sum of EDPs from each simulation, where each simulation's EDP is measured by multiplying the system power for that simulation and the

square of delay to finish the last program in that workload.

For each multi-programmed experiment, we compute the slowdown for each program, relative to its single-thread execution; the fairness metric for that experiment is the ratio of the max slowdown to the min slowdown (a number typically between 0 and 1, with 1 being extremely fair). This corresponds to the *StrictF* metric described by Vandierendonck and Seznec [32], and is not very throughput-aware. Our final PFP metric is therefore derived by dividing the average fairness across all multi-programmed workloads by the sum of delays of all involved programs.

The JWAC MSC will provide a spreadsheet with equations to compute all metrics, once simulation outputs are entered.

5 Infrastructure Shortcomings

As with all simulators, USIMM incorporates a few approximations. We will attempt to fix some of these shortcomings in future releases of the simulator.

- Trace-based simulators have the inherent shortcoming that the traces are fixed and not influenced by the timing computed during the simulation. This is especially true for multi-threaded simulations. In reality, a thread in a multi-threaded execution may spin for more or less time at a barrier depending on execution delays in each thread; however, in a trace-based simulation, the behavior of every trace is fixed and unaware of when a barrier is actually fulfilled.
- Our current trace format does not capture register operands, nor does it capture data dependences between instructions. We universally assume that all workloads exhibit a core IPC of 2.0 if they had a perfect last level cache. By ignoring data dependences, we assume that all memory instructions within a ROB exhibit near-perfect memory-level parallelism (MLP). This is clearly incorrect when (for example) handling pointer-chasing codes that exhibit nearly zero MLP.
- We assume that each trace is independent and accesses a 4 GB address space. When we execute a multi-programmed workload, we must ensure that each application maps to a different portion of the physical address space. This is done by adding a few more bits to the address to identify the core. These bits are interpreted as part of the row address bits. This assumption has a few (perhaps unintended) side-effects. As more cores (and traces) are used in one simulation, the rows per bank and hence DRAM chip capacity grows. Also, the trace for each thread of a multi-threaded application are mapped to a different region, even though they may refer to the same set of addresses. For multi-threaded workloads, the simulator will therefore under-estimate the row buffer hit rate.

- Since the EDP metric only measures the time to finish the last program, a scheduling policy that prioritizes the bottleneck program over the others may appear better than another scheduling policy that is more fair.

6 Summary

USIMM is a simulation infrastructure that models the memory system and interfaces it with a trace-based processor model and a memory scheduling algorithm. This report summarizes the state of the simulator as of February 2012. We plan to continue adding more details to the simulator and fixing some of its short-comings.

The most up-to-date weblink for obtaining the latest version of the simulator is:
`http://utaharch.blogspot.com/2012/02/usimm.html`

Users are welcome to post comments on the above web page. For questions, email the USIMM developers at usimm@cs.utah.edu. For code updates and announcements, please subscribe to the usimm-users@cs.utah.edu mailing list by visiting
`http://mailman.cs.utah.edu/mailman/listinfo/usimm-users`

References

[1] Micron System Power Calculator. `http://goo.gl/4dzK6`.

[2] Wind River Simics Full System Simulator. `http://www.windriver.com/products/simics/`.

[3] Micron DDR3 SDRAM Part MT41J128M8 . `http://download.micron.com/pdf/datasheets/dram/ddr3/1Gb_DDR3_SDRAM.pdf`, 2006.

[4] Micron DDR3 SDRAM Part MT41J256M8. `http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf`, 2006.

[5] Micron DDR3 SDRAM Part MT41J1G4. `http://download.micron.com/pdf/datasheets/dram/ddr3/4Gb_DDR3_SDRAM.pdf`, 2009.

[6] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In *Proceedings of PACT*, 2010.

[7] L. Barroso and U. Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[8] P. Bose. The Risk of Underestimating the Power of Communication. In *NSF Workshop on Emerging Technologies for Interconnects (WETI)*, 2012.

[9] B. Fanning. Method for Dynamically Adjusting a Memory System Paging Policy, 2003. United States Patent, Number 6604186-B1.

[10] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of ISCA*, 2008.

[11] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems - Cache, DRAM, Disk*. Elsevier, 2008.

[12] O. Kahn and J. Wilcox. Method for Dynamically Adjusting a Memory Page Closing Policy, 2004. United States Patent, Number 6799241-B2.

[13] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *Proceedings of HPCA*, 2010.

[14] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of MICRO*, 2010.

[15] J. Laudon. UltraSPARC T1: A 32-Threaded CMP for Servers, 2006. Invited talk, URL: http://www.cs.duke.edu/courses/fall06/cps220/lectures/UltraSparc_T1_Niagra.pdf.

[16] C. Lefurgy et al. Energy management for commercial servers. *IEEE Computer*, 36(2):39–48, 2003.

[17] K. Lim et al. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of ISCA*, 2009.

[18] Micron Technology Inc. *Calculating Memory System Power for DDR3 - Technical Note TN-41-01*, 2007.

[19] M. J. Miller. Sandy Bridge-E: The Fastest Desktop Chip Ever (For Now). `http://forwardthinking.pcmag.com/none/290775-sandy-bridge-e-the-fastest-desktop-chip-ever-for-now`.

[20] S. Miura, K. Ayukawa, and T. Watanabe. A Dynamic-SDRAM-Mode-Control Scheme for Low-Power Systems with a 32-bit RISC CPU. In *Proceedings of ISLPED*, 2001.

[21] J. Mukundan and J. Martinez. MORSE: Multi-Objective Reconfigurable SElf-Optimizing Memory Scheduler. In *Proceedings of HPCA*, 2012.

[22] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of MICRO*, 2007.

[23] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling - Enhancing Both Performance and Fairness of Shared DRAM Systems. In *Proceedings of ISCA*, 2008.

[24] S.-I. Park and I.-C. Park. History-Based Memory Mode Prediction For Improving Memory Performance. In *Proceedings of ISCAS*, 2003.

[25] S. Pawlowski. Intelligent and Expandable High-End Intel Server Platform, Codenamed Nehalem-EX. Intel Developer Forum, `http://blogs.intel.com/technology/Nehalem-EX_Steve_Pawlowski_IDF.pdf`, 2009.

[26] T. Rokicki. Method and Computer System for Speculatively Closing Pages in Memory, 2002. United States Patent, Number 6389514-B1.

[27] B. Sander, P. Madrid, and G. Samus. Dynamic Idle Counter Threshold Value for Use in Memory Paging Policy, 2005. United States Patent, Number 6976122-B1.

[28] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. In *IEEE Micro*, volume 23, pages 84–93, 2003.

[29] V. Stankovic and N. Milenkovic. DRAM Controller with a Close-Page Predictor. In *Proceedings of EUROCON*, 2005.

[30] J. Stuecheli, D. Kaseridis, D. Daly, H. Hunter, and L. John. The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies. In *Proceedings of ISCA*, 2010.

[31] Tilera. Tilera Tile64 Product Brief. `http://www.tilera.com/sites/default/files/productbriefs/PB010_TILE64_Processor_A_v4.pdf`.

[32] H. Vandierendonck and A. Seznec. Fairness Metrics for Multi-Threaded Processors. *IEEE CAL*, Feb 2011.

[33] Y. Xu, A. Agarwal, and B. Davis. Prediction in dynamic sdram controller policies. In *In Proceedings of SAMOS*, 2009.

Appendices

A Changes in Version 1.1. – March 21st 2012

To switch from version 1.0 to version 1.1, download the new version and copy a previously written scheduler.c and scheduler.h to the src/ directory in the new version. USIMM Version 1.1 incorporates the following changes (multiple new features and one bug fix):

- The trace format has changed in two ways. Instead of representing each non-memory instruction with an "N" on a new line (version 1.0), each memory instruction line starts with a number that represents the number of preceding non-memory instructions (version 1.1). Also, each memory read instruction line ends with the PC of the instruction that initiated the memory read.
- The instruction PC for a memory read is recorded in the ROB data structure and the request queue data structure. The simulator does nothing with this PC, but a scheduler might potentially find it useful.
- The input/ directory includes billion instruction traces for single-threaded executions for five PARSEC v2.0 pre-compiled binaries. The traces represent the start of the region of interest in each program. It takes tens of minutes to simulate about a billion cycles on modern machines. For short tests, users can simulate a subset of the entire trace. The "runsim" file has been updated to do an example simulation with these new traces.
- The main.c file has been updated to also accept traces for multi-threaded applications (a few multi-threaded applications will be included in the final competition workload). The individual traces of a multi-threaded application must follow a specific naming convention (starting with "MT0..", "MT1..", and so on). Typically, the addresses from each trace are given a unique prefix that matches their core ID. When a multi-threaded application is detected, the addresses from each of those trace files are given a prefix that matches the core ID for thread 0. In other words, addresses from each trace are given different prefixes, except when they belong to the same multi-threaded application.
- The scheduler is now allowed to issue an auto-precharge command in the same cycle as a column-read or column-write. This allows the row to be closed without consuming an additional command bus cycle. The scheduler does this through the is_autoprecharge_allowed( ) and issue_autoprecharge( ) commands.
- The scheduler is allowed to activate an arbitrary row even if there is no pending request in the queues for that row. This is done via the issue_activate_command( ) and is_activate_allowed( ) commands. The number of such speculative activations is tracked by stats_num_activate_spec. By default, we assume that all of these activations are done for future column-reads. This count is therefore used to influence the row buffer hit rates for memory reads.
- Bug fix: The function issue_refresh_command has been updated to correctly set the next_cmd timing constraints based on current DRAM state. In the earlier code, the dram_state was being set to REFRESHING before the state-dependent next_cmd times were being calculated. This change has a negligible impact on performance.

B Changes in Version 1.2. – April 7th 2012

To switch from version 1.0 or 1.1 to version 1.2, download the new version and copy a previously written scheduler.c and scheduler.h to the src/ directory in the new version. In going from version 1.1 to version 1.2, only the code in file memory_controller.c has changed. We've also removed a few terms from the license to make it easier for groups to use the simulator. USIMM Version 1.2 incorporates the following changes:

- Bug fix: The clean_queue function in the file memory_controller.c has been updated to delete the read and write queue nodes after the corresponding request has been serviced. This fixes a memory leak in the simulator.
- Bug fix: The issue_request function in the file memory_controller.c has been updated to fix a bug in the calculation of the next_write variable for all banks on the channel after a read or write command is issued. In version 1.1, the next_write variable was set to a larger value than what it ideally should have been. The net effect of the change is that now a write command following a read or write command can be issued earlier than in version 1.1.
- Bug fix: In the function update_memory in the file memory_controller.c, a condition has been added to make sure that forced refreshes are not issued if the mandatory 8 (or more) refresh commands have already been issued in the refresh window.
- Bug fix: In the function issue_powerdown_command, before issuing a powerdown-slow command, the function is_powerdown_slow_allowed is now being called correctly to check the issuability of the command. Earlier, there might have been situations where a powerdown_slow would be issued even if only a powerdown-fast was allowed by the timing constraints (imposed solely by the refresh deadline).
- Bug fix: In the function issue_auto_precharge in the memory_controller.c file, the calculation of the commencement of the auto-precharge command is now updated to be the maximum of the next_pre timing set by earlier commands and the first cycle when a precharge can be issued following a read or write command.
- Bug fix: Initialized the user_ptr field in the request_t structure to NULL when a new read or write entry is enqueued in the corresponding queue. This variable can now be checked in the schedule function to determine if a read or write node has been recently enqueued; this allows the user to initialize other user-defined variables soon after a request is enqueued.

C Changes in Version 1.3. – April 16th 2012

To switch from version 1.0 or 1.1 or 1.2 to version 1.3, download the new version and copy a previously written scheduler.c and scheduler.h to the src/ directory in the new version.

The version 1.3 distribution now also includes a subset of the workloads that will be used for the final competition. Here are the noteworthy points about the workloads:

- The following ten benchmark traces are included in the distribution (13 files):
  1. *black:* A single-thread run from PARSEC's blackscholes.
  2. *face:* A single-thread run from PARSEC's facesim.
  3. *ferret:* A single-thread run from PARSEC's ferret.
  4. *fluid:* A single-thread run from PARSEC's fluidanimate.
  5. *freq:* A single-thread run from PARSEC's freqmine.
  6. *stream:* A single-thread run from PARSEC's streamcluster.
  7. *swapt:* A single-thread run from PARSEC's swaptions.
  8. *comm1:* A trace from a server-class transaction-processing workload.
  9. *comm2:* A trace from a server-class transaction-processing workload.
  10. *MT*-canneal:* A four-thread run from PARSEC's canneal, organized in four files, MT0-canneal to MT3-canneal.
- Benchmarks black, face, ferret, freq, stream have about 500 million instructions. These instructions were selected from 5 billion instruction traces with a methodology similar to that of Simpoint[1].
- Benchmarks fluid and swapt are also defined with the Simpoint-like methodology described for the other PARSEC benchmarks. The only difference is that the traces include 750 million instructions so they have execution times similar to the other benchmarks.
- Benchmarks comm1 and comm2 are roughly 500 million instruction windows that are representative of commercial transaction-processing workloads.

[1]SimPoint [28] is used to generate traces which are representative of the benchmarks. SimPoint uses Basic Block Vectors(BBVs) [28] to recognise intervals of the execution which can be used to replicate the behavior of the benchmark. It assigns weights to each interval, which can be applied to the results obtained from each interval.

In our model, each benchmark is simulated for 5 billion instructions with interval sizes of 5 million instructions. A number of metrics are collected from each interval, including number of LLC misses, number of reads, number of writes, number of floating-point, integer, and branch instructions. The collection of these metrics forms the BBV that is used with SimPoint. The BBVs are classified into 10 clusters using SimPoint. The number of intervals selected from each cluster depends on the weight of each cluster. The final trace of 500M instructions is the combination of 100 intervals, taken from the large trace.

- The 4-thread canneal traces represent the first 500 million instructions executed by each thread once the region of interest is started. While all of the other traces were collected with 512 KB private LLCs for each single-thread program, the 4-thread canneal traces assumed a 2 MB shared LLC for the four cores. A write in a trace file represents the dirty block evicted by a block that is being fetched by that thread.

- The 10 traces are used to form 10 different workloads that will be used for the competition. All 10 workloads are run with 4channel.cfg, and the first 8 are also run with 1channel.cfg. The numbers from these 18 simulations will be used to compute the metrics that must be reported in the papers being submitted to the competition. The runsim file in the usimm directory lists all 18 simulations. The competition website will also have pointers to scripts, excel files, and latex table templates that can be used to compute and report the final metrics. The final competition results will be based on these 18 experiments and a few more; the additional experiments and workloads will be announced after the submission deadline. The 10 workloads are:

  1. comm2
  2. comm1 comm1
  3. comm1 comm1 comm2 comm2
  4. MT0-canneal MT1-canneal MT2-canneal MT3-canneal
  5. fluid swapt comm2 comm2
  6. face face ferret ferret
  7. black black freq freq
  8. stream stream stream stream
  9. fluid fluid swapt swapt comm2 comm2 ferret ferret
  10. fluid fluid swapt swapt comm2 comm2 ferret ferret black black freq freq comm1 comm1 stream stream

- The Fairness metric for the competition is being modified to be the following. Fairness is being defined as the maximum slowdown for any thread in the workload, relative to a single-program execution of that thread with an FCFS scheduler (a high number is bad). The final PFP metric will multiply the {average of maximum slowdowns across all experiments} and the {sum of execution times of all programs in those experiments}. For the PFP metric, only 14 of the 18 experiments will be used (the single-program comm2 workload and the multi-threaded canneal workload will not be used to evaluate fairness).

In going from version 1.2 to version 1.3, the code in files memory_controller.c, memory_controller.h and main.c have changed. USIMM Version 1.3 incorporates the following changes over version 1.2:

- Bug fix: The is_T_FAW_met is modified to correctly enforce the t_FAW condition. Earlier in a t_FAW rolling window, it would be possible for the scheduler to erroneously issue a maximum of five activations, (assuming the t_RRD timing condition was met). Now, the scheduler can issue a maximum of 4 activate commands in the t_FAW window.
- Bug fix: Changed the variable cas_issued_current_cycle to keep track of COL_RD or COL_WR commands issued to each bank. Earlier, the variable only kept track of whether a COL_RD or COL_WR had been issued in the current simulation cycle to a channel before issuing an autoprecharge. Also, the variable is now reset when an autoprecharge command is issued. This has no impact on correct implementations of the autoprecharge functionality. The change prevents schedulers from incorrectly issuing multiple auto-precharges to a channel in the same cycle and also prevents an autoprecharge to be sent to a bank that did not have a COL_RD or COL_WR issued to it that very cycle.
- Changes to statistics: New variables, stats_reads_merged and stats_writes_merged, counting the number of merged reads and writes respectively, have been exposed to the scheduler. The variables fetched and committed (which, respectively, contain the fetched and committed instruction counts for each simulated core) have been migrated from the file main.c to memory_controller.h to allow the scheduling algorithm to use this information. The simulator also now prints the sum of execution times on each core and the EDP metric for the simulation.