Stride- and Global History-based DRAM Page Management

Mushfique Junayed Khurshid, Mohit Chainani, Alekhya Perugupalli and Rahul Srikumar
University of Wisconsin-Madison

***Abstract*—To improve memory system performance, besides greedily scheduling the correct memory command at the correct time, latencies can be hidden by the use of predictors for opening/closing pages. This paper presents a memory scheduler, which is enhanced by speculative precharging of banks and activation of rows using techniques used for data cache prefetching. A stride detector in the scheduler essentially detects constant strides in memory reference addresses and speculatively precharges banks or activates rows in the row buffer based on those strides. An additional Global History Buffer structure, similar to the one used for data cache prefetching, is also used to speculatively precharge banks. This greedy scheduler, issues speculative row activates or precharges, only in memory cycles when no other command residing in the read/write queue can be issued, so that any mis-speculation has a minimal effect on the overall performance. The schedulers performance is evaluated in terms of execution times for various traces in both 1-channel and 4-channel processor-memory configurations. Its performance is compared with that of FR-FCFS memory scheduler in this paper. This scheduler targets the Performance track of the Memory Scheduling Championship organized by The Journal of Instruction Level Parallelism. The authors Mushfique Junayed Khurshid (Email: khurshid@wisc.edu), Alekhya Perugupalli and Rahul Srikumar are Masters students, while Mohit Chainani is an undergraduate student, all in the ECE department of University of Wisconsin-Madison.**

***Index Terms*—DRAM, Memory Scheduling, Global History Buffer, Stride detection, Memory Scheduling Championship.**

I. Introduction

DRAM(Dynamic Random Access Memory) memories are very important, especially in many performance-critical applications such as, in telecommunications - packet buffer and lookup table memories, and also in digital imaging applications like 3D games and TVs etc [1]. Furthermore, we can observe that as systems are becoming more and more complex, interactions between systems like the processor and the memory are becoming increasingly significant. An inefficient memory performance can result in an overall inefficient system. So we can understand that the performance of the DRAM is critical. Performance of memories is improved in contemporary DRAM chips by organizing the DRAM memories into independent banks, hence enabling pipelining of several memory requests. Also each bank has a row buffer which stores the last accessed row in the bank. This improves the bandwidth of the memory, but in fact causes the memory latency to be dependent on the pattern of memory accesses.

Each bank stores several rows which contains columns of bits. A particular memory access time is given as the sum of the following delays - T_{PR} (Precharge time: time required to precharge the previously accessed row which is in the row buffer), T_{RA} (Row Activate time: time required to access one row and copy it into the row buffer) and T_{CA} (Column Access time: time required to access the specific column in the row in the row buffer). If a certain data access requires data from a row which is already activated and is in the row buffer, the time required to access that data will be significantly reduced and will simply be the T_{CA}. While, if an already activated row is closed before the next request arrives which requires access to a different row, the new request will require less time to access the memory, as it only requires the T_{RA}, and T_{CA}. In above two cases, the $T_{PR} + T_{RA}$ and T_{PR} respectively, are hidden by speculative precharging and activation of rows. This calls for predictors that can predict when to open certain rows, and when to close certain rows, to improve overall memory performance, and there had been research on memory controllers built with such predictors [1][2][3].

Constant stride detecting prefetching techniques has been used for predicting future memory addresses for prefetching for data caches [4][5]. Our scheduler detects constant strides, using such techniques, in the memory requests in the read/write queues to predict future memory references, which enables the scheduler to either open or close pages. Our scheduler, also uses a structure similar to the Global History Buffer for data cache prefetching [6], but uses this to only speculatively close pages. Apart from these predictor structures, the scheduler issues commands to the memory greedily. The predictors only speculatively issue commands to the memory in memory cycles where no non-speculative commands in the read/write queues are issuable. Hence the scheduler is basically divided into three entities :

1) Base Scheduler
2) Constant Stride Detecting Open/Close page predictor
3) Global History Buffer based Close page predictor

Each of the entities issue commands to the memory, and their issuing power are in the order as listed. The predictors only issue commands if no command is issuable by the Base Scheduler.

II. Scheduler

A. Base Scheduler

Everytime the memory switches from writing/reading a column to the row buffer, to reading/writing a column from the row buffer, a write to read delay t_{WTR}/read to write delay t_{RTW} is required for bus direction switching. Hence our scheduler serves read and write requests in bursts. Write requests are served in bursts whenever number of write requests in the queue exceeds the High Watermark of 40, until the number of write requests in the queue reach the Low Watermark of 20 - which we refer to as the write drain mode. In this write drain mode, only write requests are served. But in case no write request commands are issuable in one memory cycle, the scheduler only attempts to serve the precharge or activate row commands pending in the read queue, as long as the target bank involved does not conflict with any pending write commands. However, when in non-write mode, when no read request commands are issuable in a memory cycle, the base scheduler attempts to issue ONLY precharge commands from the write queue, whose bank does not conflict with any pending read requests. This is because a row activate command from the write queue may slow down a future read request due to conflicting bank. Since in a program, number of reads is generally higher than number of writes, the probability of this occuring is higher than the other way round. In write drain mode, Column Read commands from the read queue are not entertained because of the bus switching delay caused. Same is true for Column Write commands, when there are no issuable read commands in non-write mode. When attempting a command from the read queue in non-write drain mode, and a command from the write queue in write drain mode, the scheduler follows a FR-FCFS policy (gives first priority to row buffer hits, otherwise FCFS).

We can observe that in many stages of the scheduling, the scheduler makes a decision based on whether a particular bank is used by any pending read/write request or not. A table called Isused is used to keep track of whether a particular bank is used by a pending request or not. The table has an entry for every bank in the memory. The Isused entry is zero if the bank is not used by any pending request, is one if the bank is used by one or more pending read requests, is two if it is used by one or more pending write requests and is three if it is used by atleast one read and atleast one write request. This way, the scheduler keeps track at any time, which banks are targeted by any the pending read/write requests in the read/write queue and make decisions accordingly.

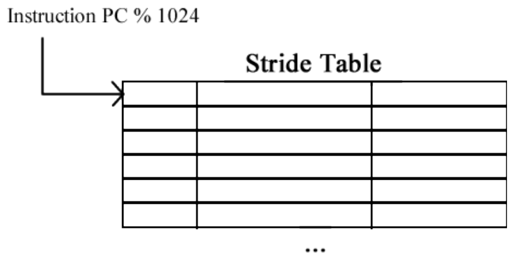
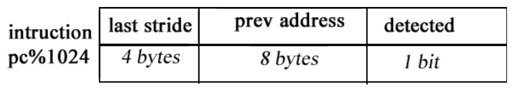


Fig. 1. Stride Table structure that records detected strides, indexed by Instruction PC



	last stride	prev address	detected
intruction pc%1024	*4 bytes*	*8 bytes*	*1 bit*

Fig. 2. Each entry of a stride table, and the size of each field in the entry

B. Constant Stride Detecting Open/Close page predictor

For constant stride detection, a Stride Table as in Figure 1 is maintained. It can contain 1024 entries, and is indexed by the 10 LSBs of the Instruction Program Counter of the memory request. Each entry, as in Figure 2, in the stride table contains a last stride value, previous address accessed, and a detected bit. When the scheduler looks through the read/write queue, existing constant strides between memory references having same instruction pc are recorded using this structure. When a stride value for a particular memory reference is same as the last stride value in the Stride Table entry, the detected bit is set. The Figure 2 shows the stride table entries and its data types.

Initially, as the base scheduler looks through the read/write queue for row buffer hits, the corresponding stride table entry values are ALL initialized to zero. When the base scheduler looks through the read/write queue again for any other issuable commands, the constant stride detection in the current read/write queue takes place. When no commands are issuable, the predictor basically runs through all the corresponding Stride table entries for the read/write requests in the read/write queues when in non-write/write mode. For every entry for which the detected bit is set, the predictor attempts opening or closing rows of the banks corresponding to physical addresses of $a + s$, $a + 2s....a + ds$, where a is the previous address value in the stride table entry, s

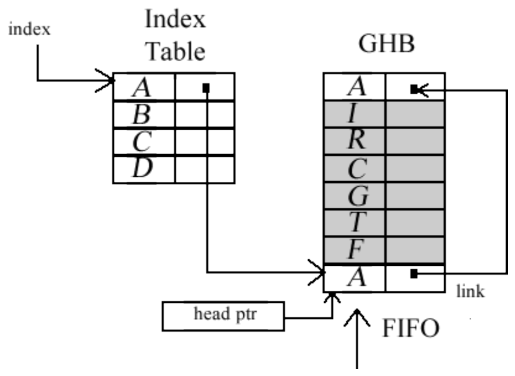


Fig. 3. Global History Buffer structure used for close page prediction

number	thread id	instruction PC	physical address	link ptr
4 bytes	*4 bytes*	*8 bytes*	*8 bytes*	*8 bytes*

Fig. 4. Entry in the Global History Buffer and size of each field in the entry

is the stride value in the stride table entry and d is the depth of striding speculation. Our analysis showed an optimum depth of 7, which we chose for our scheduler. If a particular speculative memory access is in a different channel, the corresponding access details are stored in a separate array called the TobeIssued array, which can be read in by the scheduler of that particular channel, and hence executed, only when it cannot issue any other command with the basic scheduler or the predictors.

C. Global History Buffer based Close page predictor

This predictor, as mentioned before, uses a structure similar to the Global History Buffer structure used for data cache prefetching [6] as in Figure 3. This structure has got two levels:

1) Index Table (IT): This has 1024 entries. This is accessed with a key, the 8 LSBs of which are the xor of the instruction pc value and the memory address of the memory request, and the top 2 MSBs of which is the thread id of the memory request. The entries in the IT points to corresponding Global History Buffer entry.
2) Global History Buffer (GHB): It has 512 entries, and it is a FIFO table (circular buffer), which contains the 512 most recent miss addresses. Each entry in the GHB has a link pointer that connects it to the previous GHB entry that has the same Index Table key.

Our GHB only keeps the addresses of read requests. We attempted to keep another GHB for write requests also, but it did not give us any performance gain. Combining the GHBs accurately would be more complex, as the read and write queues are separate and to combine their requests into GHB would require a merge and sort (in terms of time stamp), which would require too much resources and would be impractical for a speculative action. Also, since the scheduler schedules reads and writes in bursts, a combined read-write GHB is unnecessary. The scheduler pushes the new memory requests in the Global History Buffer at the beginning of every memory cycle. Then, in case the Base Scheduler and the Constant Stride Detecting Open/Close page predictor fails to issue any memory command, the GHB based close page predictor tries to issue speculative precharges. It goes to the head of the GHB table (which is the latest memory request pushed), and follows its link pointer to any previous occurrence of it in the GHB table. If it finds it, the predictor moves downward along the GHB (which gives it the next miss addresses, essentially potential memory request predictions - greyed entries in Figure 3) and speculatively closes pages (as long as its bank does not conflict with that of any read/write pending requests. The predictor attempts issuing commands till a depth of 20 and for a width of 3. Depth of 20 means for each previous occurence of a miss address, we attempt to close pages corresponding to 20 instructions following that occurence. While a width of 3 means that 3 previous occurences are tried. A GHB entry is shown in Figure 4. It contains the GHB entry number, the thread id (since prediction must be same thread as the latest memory reference), the physical address of the memory reference, the instruction PC value and the link pointer.

TABLE I
MEMORY REQUIREMENTS FOR SCHEDULER

Entity	**Size**
Isused	16*16*32*2B=16KB
Stride Table	1024*(12B+1b)=(12K+128)B
ToBeIssued	16*20B=320B
GHB	512*32B=16KB
Index Table	1024*8B=8KB
Total	**(52K+448)B**

III. IMPLEMENTATION

The implementation of the base scheduler is pretty straight forward, since it is similar to FR-FCFS, with write drain mode as presented earlier. The Stride Detecting consists of a an indexed Stride Table. The TobeIssued array has 16 entries (as many as maximum number of channels), which stores pending speculative memory commands of different channels as predicted by the constant stride prediction. The GHB structure implementation can be very similar to that in the data cache prefetching case[6]. A simple state machine maintains the GHB in a FIFO manner, updating its contents every memory cycle. The new address is added to the GHB and its head is made to point to it. The link address of

TABLE II
BENCHMARK TRACES

Trace	Abbreviation
PARSECs blackscholes	bl
PARSECs facesim	fa
PARSECs ferret	fe
PARSECs fluidanimate	fl
PARSECs freqmine	fr
PARSECs stream	st
PARSECs swaptions	sw
Server-class transaction processing workload -1	c1
Server-class transaction processing workload -2	c2
PARSECs canneal (4 threads)	MTc

TABLE III
WORKLOADS AND TOTAL EXECUTION CYCLES

No.	Workload	1-channel	4-channel
1	MTc	4039625764	1713795068
2	bl-bl-fr-fr	1462590705	767061157
3	c1-c1	819168979	474230867
4	c1-c1-c2-c2	2360998257	1182314394
5	c2	427719792	272913013
6	fa-fa-fe-fe	2223741393	1000324222
7	fl-sw-c2-c2	2427112764	1214903047
8	st-st-st-st	1585047277	820932361
9	fl-fl-sw-sw-c2-c2-fe-fe		2785766266
10	fl-fl-sw-sw-c2-c2-fe-fe-bl-bl-fr-fr-c1-c1-st-st		6178767813
	Grand Total	**31757013139**	

this new GHB head is made to point to the Index Table entry. Finally the Index table entry is updated to the newly added entrys address. As GHB entries are evicted, we detect its Index Table entry, and either delete that entry if it is same as the address to be evicted, and if not, we traverse along the links corresponding to this Index Table entry along the GHB, until we find the miss address which is linked to the address to be evicted. We label its link as NULL. We are given a budget of 68KB, and the Table I shows how the scheduler design meets the criterion.

IV. EXPERIMENTAL RESULTS

The memory scheduler is simulated using the simulation infrastructure - USIMM (Utah Simulated Memory Module) to evaluate the scheduler [7]. The USIMM provides for two system configurations [7]. One is a smaller scale processor with one memory channel, while the other one is a more aggressive processor with 4 memory channels. Various different work loads are run to evaluate the performance. The traces, from PARSEC [8] and others, are tabulated in Table II along with their abbreviations. All are single thread traces except for the one as labelled.

Different combinations of the above traces are executed in both 1-channel and 4-channel configurations as multi-threaded workloads. The metric used is the SUM of Execution times of ALL the individual threads in a particular workload. The results are tabulated in Table III. The trace abbreviations as in Table II are used to represent the multi-threaded workloads. The execution time is given in terms of processor cycles.

These workloads were also run using an FR-FCFS scheduler, in the two memory system configurations to compare with our scheduler. Overall, our scheduler gave a 3.69% decrease in execution time, when compared to FR-FCFS. The percentage decrease in execution time, used as a metric here, is given by the following equation:

$$\%\text{ decrease in Exec. time} = \frac{E_{FR-FCFS} - E}{E_{FR-FCFS}} * 100$$

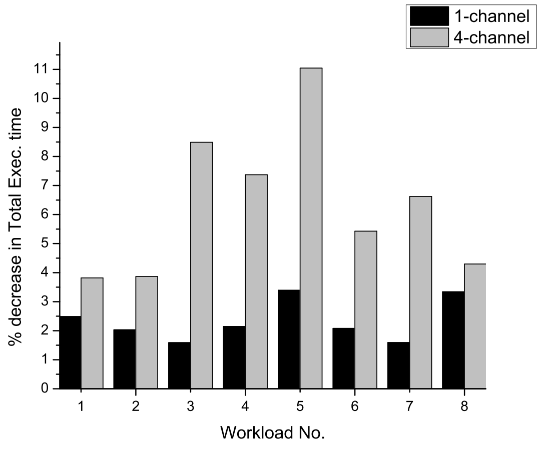


Fig. 5. Decrease in Execution time for various workloads in both 1-channel and 4-channel configurations

$E_{FR-FCFS}$ is the total execution time for FR-FCFS scheduler, while E is the total execution time for our scheduler. The percentage decrease in execution time for our memory scheduler when compared to FR-FCFS, for each of the workloads as numbered in Table III and for both 1-channel and 4-channel configurations are shown in Figure 5. Even though that on average our scheduler showed a 3.69% less time for all workloads, we can observe that the percentage decrease in execution times for all workloads in 1-channel and 4-channel configuration is quite unbalanced. The performance improvement over FR-FCFS is much higher in 4-channel configuration than in 1-channel configuration. We get a maximum of 11.04% decrease in execution time for one workload in 4-channel configuration, when compared to FR-FCFS in the same configuration. On average, in 4-channel configuration, our scheduler reduces the execution time by 6.11% over FR-FCFS.

V. CONCLUSION

In this paper, we presented a memory scheduler, which greedily issues commands to the memory, and also uses techniques used for data cache prefetching to speculatively precharge/activate rows. We demonstrated that the scheduler is easily implementable, and its memory requirement also is below the given budget of 68KB. When compared with FR-FCFS memory scheduler, overall, our scheduler decreases the execution time by 3.69%. A maximum execution time decrease of 11.04% is observed, and in a 4-channel configuration, we observe an average execution time decrease of 6.11%. We can see that multiple memory channel systems make our scheduler give greater improvement over FR-FCFS than in single memory channel systems.

VI. Acknowledgment

The authors would like to thank Prof. Mikko Lipasti of University of Wisconsin-Madison for his helpful suggestions and guidance.

References

[1] V.V. Stankovic and N.Z. Milenkovic. DRAM Controller with a Complete Predictor: Preliminary Results, 2005.

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

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

[4] J.W.C. Fu and J.H. Patel. Stride directed prefetching in scalar processors. In Proceedings of the 25th International Symposium on Microarchitecture, 1992.

[5] S. Kim, A. Veidenbaum, Stride-directed Prefetching for Secondary Caches, In 1997 International Conference on Parallel Processor, 1997.

[6] K. Nesbit and J.E. Smith, Data Cache Prefetching Using a Global History Buffer. In Proceedings of the 10th Annual International Symposium on High Performance Computer Architecture, pp. 144-154, July 2001.

[7] USIMM: the Utah Simulated Memory Module. http://www.cs.utah.edu/ rajeev/pubs/usimm.pdf

[8] Christian Bienia. Ph.D. Thesis. Princeton University, January 2011.