Architecting High-Speed Command Schedulers for Open-Row Real-Time SDRAM Controllers

Leonardo Ecco and Rolf Ernst
Institute of Computer and Network Engineering — TU Braunschweig, Germany
{ecco,ernst}@ida.ing.tu-bs.de

***Abstract*—As SDRAM modules get faster and their data buses wider, researchers proposed the use of the *open-row* policy in command schedulers for real-time SDRAM controllers. While the real-time properties of such schedulers have been thoroughly investigated, their hardware implementation was not. Hence, in this paper, we propose a highly-parallel and multi-stage architecture that implements a state-of-the *open-row* real-time command scheduler. Moreover, we evaluate such architecture from the hardware overhead and performance perspectives.**

I. Introduction and Related Work

A SDRAM controller is mainly comprised of two orthogonal and possibly independently developed parts: the command scheduler, which generates commands to serve incoming requests, and a *physical (PHY)* layer, which handles circuit-level communication. This paper focuses on the former.

SDRAMs have a bank-based structure and a two-stage access protocol that relies on an intermediate level of buffering. Traditionally, SDRAM command schedulers for real-time systems relied on the *close-row* policy [1], which only exploits buffering locality within the boundary of a single request. Nevertheless, for fast SDRAM modules with wide data buses, the traditional approach is outperformed by a combination of bank privatization and *open-row* policy[1] [3], which attempts to exploit buffering locality over the boundary of a single request. (Both approaches have benefits and drawbacks. A comparison is available at [3] and is not the goal of this article).

The real-time properties of *open-row* schedulers have been widely investigated. However, their hardware implementation was not. This paper addresses such challenge. In summary, its **main contributions** are: (1) We propose a highly-parallel multi-stage architecture that implements a state-of-the-art *open-row* command scheduler for real-time SDRAM controllers [4], [5]. (2) We synthesize our scheduler for a 65 nm UMC process and show that it can achieve high operational frequencies. (3) We show that the multi-stage architecture has minimal impact on the performance of the implemented command scheduling algorithm.

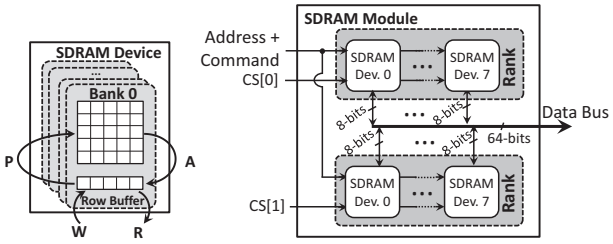
II. Background

In this section, we firstly describe SDRAM devices, ranks and modules. Then, we describe the *open-row* real-time SDRAM controller whose command scheduler we implement.

A. SDRAM Devices, Ranks and Modules

For all intents and purposes, we employ the word SDRAM to refer to DDR3 SDRAM devices [6]. In this subsection, we firstly discuss SDRAM devices. Then, we describe how individual devices are used to form ranks, which are in turn used to build a module.

We depict the logical structure of a SDRAM device and the commands used to operate it in Fig. 1a. Each SDRAM device is divided into banks. For DDR3, the number of banks is 8. Each bank contains a matrix-like structure and a row buffer (the internal level of buffering mentioned in the introduction).



(a) SDRAM device. (b) SDRAM module.

Fig. 1. SDRAM system.

There are four commands used to move data into/from a row buffer: A, P, R and W. The *activate* (A) command loads a matrix row into the corresponding row buffer, which is known as *opening a row*. The *precharge* (P) command writes the contents of a row buffer back into the corresponding matrix, which is known as *closing a row*. The *read* (R) and *write* (W) commands are used to retrieve or forward words from or into a row buffer. We use the acronym CAS (Column Address Strobe) to refer to both *read* and *write* commands.

CAS commands operate in bursts, which means that each of them transfers more than one word. The exact amount of words transferred by a CAS command is determined by the the burst length (BL) parameter, which is 8 for DDR3 [6]. A single CAS command occupies the data bus for $t_{BURST} = BL/2 = 4$ cycles and transfers $BL \cdot W_{BUS}$ bits, where W_{BUS} represents the width of the data bus.

There are several timing constraints that dictate a minimum distance between consecutive commands. For illustrations and a detailed explanation, we refer the reader to [4], which also provides a discussion about refreshes.

We now discuss ranks and modules. A group of SDRAM devices operating under the same chip-select signal is referred to as a rank. From the perspective of a SDRAM controller, a rank is treated as a single SDRAM device with larger banks and a larger number of data bus pins. SDRAM modules are printed circuit boards which contain one or more ranks (an example of a dual-rank module is depicted in Fig. 1b). Because all ranks in a module share the data bus, the scheduling of CAS commands to different ranks must respect *inter-rank* timing constraints. For details and illustrations, see [5].

B. Open-Row Scheduler Under Consideration

In this subsection, we provide an overview of the *open-row* real-time SDRAM controller proposed in [4], [5], whose command scheduler we implement in this work. We depict

[1]Performance-optimized COTS controllers also employ the *open-row* policy. With regard to a real-time counterpart, there are two main differences: firstly, they buffer incoming write requests and serve them opportunistically. Secondly, they perform intra-bank reordering to increase buffering-locality exploitation. Both features lead to poor worst-case bounds. A detailed discussion is available at [2].

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

the logical architecture of the SDRAM controller in Fig. 2. Notice that the architecture is generic and supports an arbitrary module configuration, i.e. number of ranks (nR) and number of banks per rank (nB).

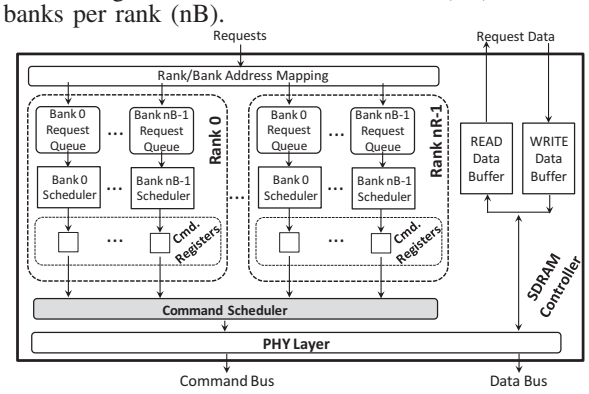


Fig. 2. Logical architecture of the SDRAM controller. The command scheduler (which is referred to as channel scheduler in [4], [5]) is highlighted.

Each bank scheduler generates the commands used to serve its oldest pending request (either a P-A-CAS sequence, or only a CAS command depending on whether the request hits or misses at the row buffer). If a pending command from a bank scheduler can be immediately executed without violating any *exclusively intra-bank* timing constraint (see [4]), such command is placed into the corresponding command register. The command scheduler, the focus of this paper, arbitrates between all command registers.

From the real-time perspective, an implementation of the scheduling algorithm described in [4], [5] must obey the following rules:

- CAS commands have priority over non-CAS commands, i.e. *activates* and *precharges*.
- Each non-CAS command can be stalled by at most other $(nR \cdot nB) - 1$ non-CAS commands, i.e. one time for each interfering command register in the system.
- CAS commands are arbitrated using the read/write bundling approach. An example of read/write bundling is depicted in Fig. 3.

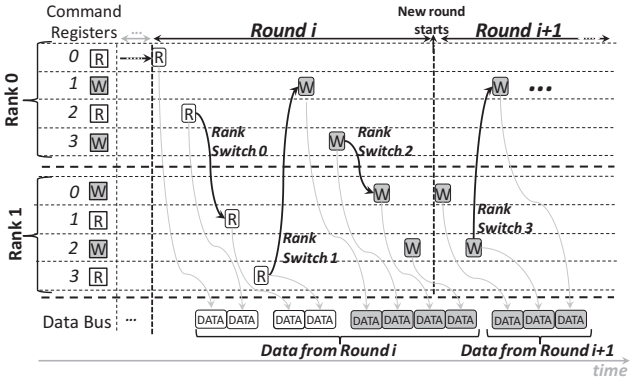


Fig. 3. Example of scheduling of CAS commands using read/write bundling. The example assumes that command registers are immediately replenished after the command scheduler executes (empties) a register. Notice that in each scheduling round, at most one CAS command for each of the command registers is executed.

The idea of read/write bundling is to perform aggressive command reordering in order to minimize data bus turnarounds (inside the same rank) and rank switches, which have a negative effect in worst-case performance.

III. Command Scheduler Architecture

In this section, we propose a highly-parallel and multi-stage architecture that implements the command scheduler described in Section II-B. For that purpose, we first clarify that SDRAM command schedulers are designed to operate at the same clock frequency as the data bus from the controlled SDRAM module.

In such scenarios, fitting all the logic required to arbitrate between pending commands in one cycle becomes unfeasible. Nevertheless, a compromise can be made: we can exploit the timing relationship between consecutive SDRAM commands in order to distribute the computation over a predetermined number of cycles (and, hence, the multi-stage expression). For instance, each CAS command occupies the data bus for $t_{BURST} = 4$ cycles. Hence, after executing a CAS command, our architecture has at least 4 cycles to decide the next CAS command to be executed (assuming CAS commands of the same type and to the same rank). This means that, in comparison with an ideal implementation in which decisions are made in one cycle, each CAS command will have an extra latency of $4 - 1 = 3$ cycles. However, in a system backlogged with requests, such extra delay is mitigated.

A. Overview

We depict a block diagram of the command scheduler in Figure 4. The arbitration of commands is performed in two layers: firstly, commands are arbitrated inside their corresponding arbiters (in the figure, notice the demultiplexers used to route a command to the proper arbiter, e.g. *activates* go to the Activate Arbiter). Then, winners from the first layer of arbitration compete with each other in the Command Bus Arbiter.

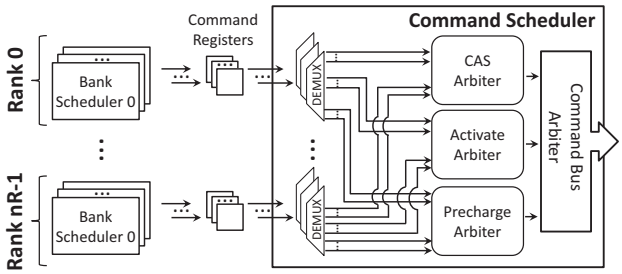


Fig. 4. Command Scheduler.

We structure the remainder of this section into two parts: firstly, we discuss the CAS Arbiter in detail, as it contains an important optimization for command scheduling in real-time systems, i.e. read/write bundling [4], [5]. Then, we provide a superficial discussion of the Activate, Precharge and Command Bus Arbiters. We omit a thorough discussion about them because our space is limited.

B. CAS Arbiter

We depict the logical architecture and an example of operation of the CAS Arbiter in Fig. 5 and discuss its logical blocks below:

- **CAS Arbiter Inputs:** the inputs from the CAS arbiter come directly from the demultiplexing layer employed in the command scheduler (see Fig. 4). In this context, we represent the absence of a command with the letter *X*. In the figure, this is the case for two inputs of Rank 0 and means that the command registers under consideration have either pending *activates* or *precharges* (which are routed to different command arbiters, as depicted in Fig. 4), or no pending command at all.

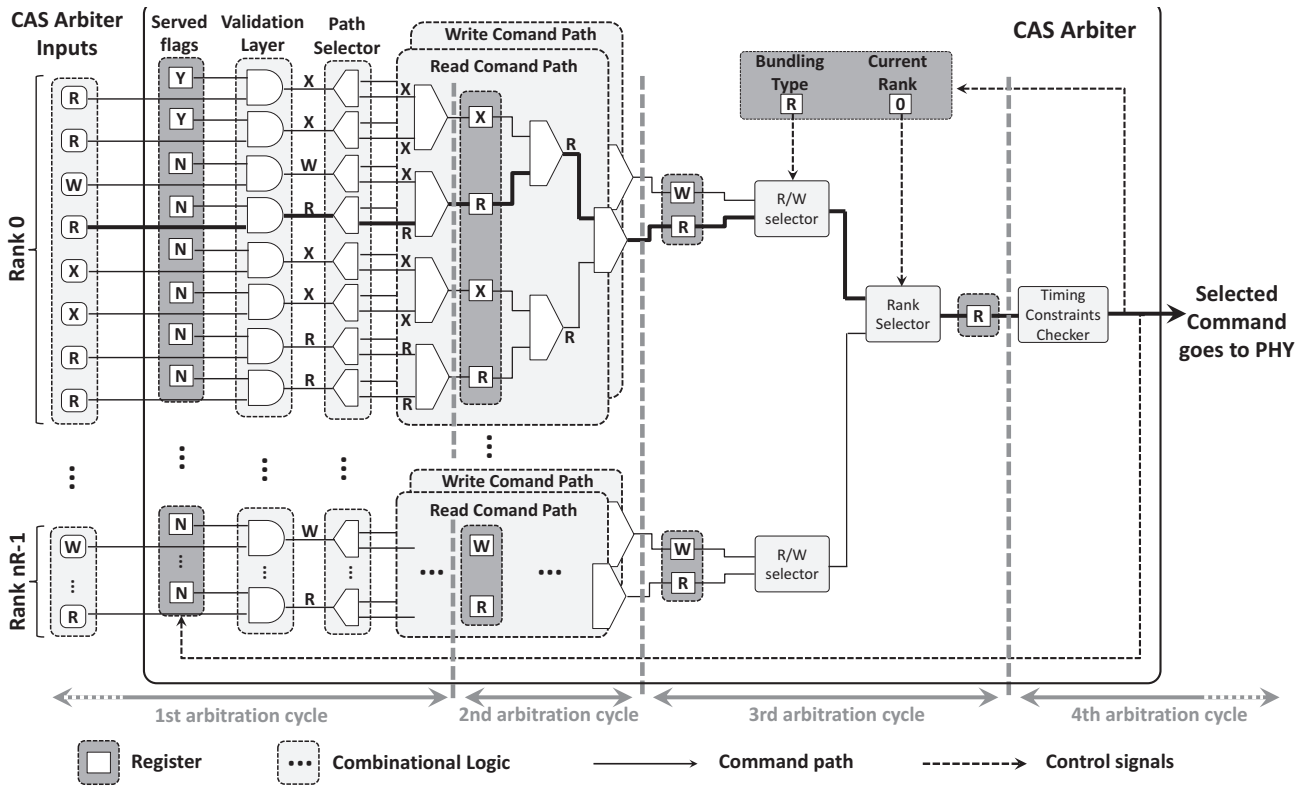


Fig. 5. Logical architecture and example of operation of the CAS Arbiter. The path taken by the command that wins the arbitration is highlighted. Moreover, the proportions used in the figure were chosen in order to improve readability and not to accurately depict the amount of combinational logic between each stage of arbitration. In practice, each arbitration stage has a similar critical path, which is not the case in the figure.

- **Served Flags**: for each command register, there is a *flag* that indicates whether the CAS command in the register is eligible for schedulability in the current scheduling round (see Fig. 3). In the figure, the *flags* assume either a *Yes (Y)* or a *No (N)* value. When a CAS command is executed, the corresponding flag is changed to *Yes*. Then, when the current scheduling round is over, all *flags* are set to *No*. This enforces that at most one CAS command for each register is executed per scheduling round and is required for real-time guarantees to be extracted.
- **Validation Layer**: forwards the pending command to the next layer if the corresponding *served flag* is *Yes*. Otherwise, forwards an *X*.
- **Path Selector**: forwards *validated read* and *write* commands to the Read and Write Command Paths, respectively. Notice that at least one of the outputs of a Path Selector will be an *X*, e.g. if the input is a *read* command, the output to the Write Command Path will be *X*.
- **Read and Write Command Paths**: arbitrate between all pending *validated* commands and employ fixed-priority. Inside the Read and Write Command Paths, each of the 5-sided polygons represents a fixed-priority arbiter.
- **Bundling Type and Current Rank Registers**: determine which CAS commands can be executed. In the example given in the figure, only *read* commands from Rank 0 are eligible for execution.
- **R/W and Rank Selectors**: select between multiple pending commands according to the value of the registers mentioned in the previous item.
- **Timing Constraints Checker**: holds the command that won the arbitration until it can no longer cause a timing constraint violation. In summary, all classes of constraints with the exception of the *exclusively intra-bank* (handled by the bank schedulers) must be accounted for. More details are given in [4], [5].

The operation of the CAS Arbiter is controlled by the Bundling Type and Current Rank registers. Such registers are updated when no valid command reaches the output of the arbiter according to the rules described in [4], [5].

C. Activate, Precharge and Command Bus Arbiters

Both the Activate and the Precharge Arbiter also employ multi-stage architectures. For the former, the extra cycles to make a decision are hidden due to the t_{RRD} constraint (see [4]), which calls for at least a 4-cycles delay between consecutive *activates* to the same rank. For the latter, the overhead is simply not hidden. Finally, in order to implement the scheduling rules described in Section II-B, the Command Bus Arbiter prioritizes the output of the CAS Arbiter. If there are no pending CAS commands, a round-robin scheme is used to arbitrate between the Activate and the Precharge Arbiters.

IV. Evaluation

In this section, we evaluate our multi-stage command scheduler architecture. For that purpose, we consider systems with 1, 2, 4 and 8 ranks (with 8 banks per rank).

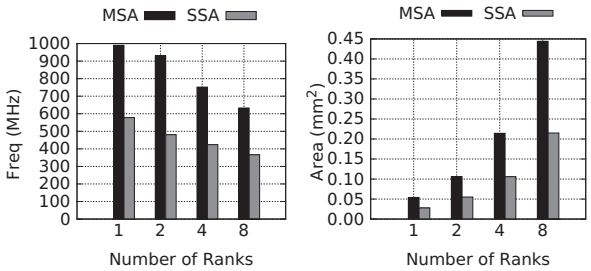
Our evaluation is divided into two parts. In the first part, we synthesize our *multi-stage architecture (SSA)* using a 65 nm UMC process and present area and frequency results. Moreover, we compare the results with the ones obtained for a *single-stage architecture (SSA)*, which fits all logic required to make a scheduling decision into a single clock cycle.

In the second part, we compare the multi-stage architecture

with a *hypothetical single-stage architecture (HSSA)* from the performance perspective. In this sense, we employ the word *hypothetical* to refer to a single-stage architecture that is able to operate at frequencies as high as the ones from the MSA. This comparison is important because the multi-stage architecture slightly modifies the latency of commands.

A. Synthesis Results

We synthesize the MSA and the SSA using the Synopsys Design Compiler tool, setting *speed* as optimization goal and a target clock frequency of 1 GHz, i.e. a clock period of 1.0 nanosecond. We depict the frequency and area results in Figs. 6a and 6b, respectively.



(a) Frequency. (b) Area.

Fig. 6. Synthesis results for a 65 nm process from UMC.

We make two observations about the results: firstly, the MSA architecture considerably outperforms the SSA from the frequency perspective. Such improvement comes at the cost of increased area. And secondly, as the number of ranks increase, the *rank selectors* become a bottleneck and the maximum operating frequency drops for both the MSA and the SSA. Overcoming such bottleneck can be achieved either by (further) pipelining the *rank selectors* or using a smaller manufacturing process. Both options are left as future work.

B. Experimental Comparison with HSSA

In this subsection, we experimentally compare the MSA with the HSSA. Our evaluation, which is trace-based, consists in performing cycle-accurate simulations and recording the following data: (1) the average data bus utilisation that each architecture is able to maintain. The higher the utilisation, the better. And (2) the cumulative SDRAM latency that applications making requests experience. As well detailed in [4], [5], the cumulative SDRAM latency refers to the sum of the latency of all requests performed by an application.

For our evaluation, we consider a dual-rank DDR3-1866M module with a total of 16 banks and a 64-bit wide data bus. As for the workload, we employ 16 applications from Mibench [7] because we consider a bank privatization setup, i.e. one application per bank, as is usually employed with the *open-row* policy. To obtain the request traces, we execute the applications in an architecture simulator, exactly as described in [4], [5]. However, in order to speed up the costly cycle-accurate simulations, we only consider the first 5000 requests from each obtained trace.

During the simulation, *trace players* read the requests from the traces and generate the appropriate commands. Each *trace player* only generates commands for a new request after the previous one has been served, as to mimic a processor that tolerates at most one pending request, as assumed by the timing analysis from [4], [5].

We perform a simulation both for the MSA and the HSSA and depict the results in Figs. 7 and 8. The results show that the difference in performance between both architectures is

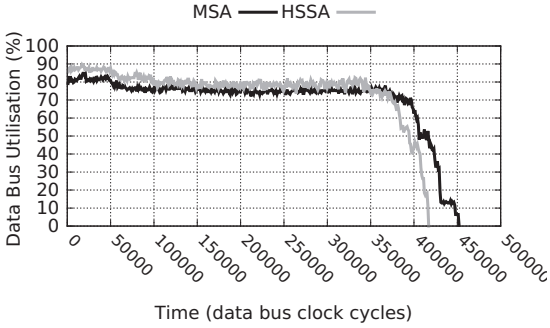


Fig. 7. Data bus utilisation over time. The drop to 0% marks the moment in which all requests are served.

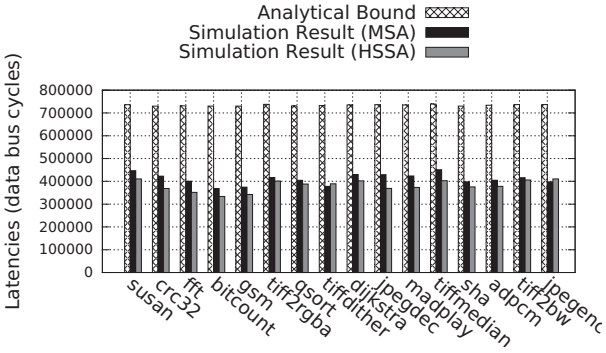


Fig. 8. Cumulative SDRAM latencies.

negligible. Hence, we conclude that the multi-stage architecture allows us to achieve high operational frequencies with negligible performance overhead in comparison with a HSSA.

V. Conclusion

In this paper, we present a highly-parallel and multi-stage architecture of an *open-row* command scheduler for real-time SDRAM controllers. For that purpose, we exploit the timing relationship between consecutive SDRAM commands in order to distribute the arbitration logic over a predetermined number of stages. The multi-stage approach allows us to synthesize our design at competitive clock frequencies, even with a non cutting-edge manufacturing process (65 nm from UMC). Finally, our experiments show that the multi-stage architecture does not significantly alter the performance of the implemented scheduling algorithm.

References

[1] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A Predictable SDRAM Memory Controller," in *Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. ACM Press New York, NY, USA, Sep. 2007, pp. 251–256.

[2] H. Yun, R. Pellizzoni, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, July 2015, pp. 184–195.

[3] Z. P. Wu, R. Pellizzoni, and D. Guo, "A composable worst case latency analysis for multi-rank dram devices under open row policy," *Real-Time Systems*, pp. 1–47, 2016.

[4] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *Real-Time Systems Symposium (RTSS), 2015 IEEE*, Dec 2015, pp. 53–64.

[5] L. Ecco, A. Kostrzewa, and R. Ernst, "Minimizing DRAM rank switching overhead for improved timing bounds and performance," in *Euromicro Conference on Real-Time Systems (ECRTS) 2016*, July 2016.

[6] *JESD79-3F: DDR3 SDRAM Specification*, JEDEC, Arlington, Va, USA, Jul. 2012.

[7] M. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.