Power/Performance Trade-offs in Real-Time SDRAM Command Scheduling

Sven Goossens, Karthik Chandrasekar, Benny Akesson, and Kees Goossens

Abstract—Real-time safety-critical systems should provide hard bounds on an applications' performance. SDRAM controllers used in this domain should therefore have a bounded worst-case bandwidth, response time, and power consumption. Existing works on real-time SDRAM controllers only consider a narrow range of memory devices, and do not evaluate how their schedulers' performance varies across memory generations, nor how the scheduling algorithm influences power usage. The extent to which the number of banks used in parallel to serve a request impacts performance is also unexplored, and hence there are gaps in the tool set of a memory subsystem designer, in terms of both performance analysis, and configuration options.

This article introduces a generalized close-page memory command scheduling algorithm that uses a variable number of banks in parallel to serve a request. To reduce the schedule length for DDR4 memories, we exploit bank grouping through a pairwise bank-group interleaving scheme. The algorithm is evaluated using an ILP formulation, and provides schedules of optimal length for most of the considered LPDDR, DDR2, DDR3, LPDDR2, LPDDR3 and DDR4 devices. We derive the worst-case bandwidth, power and execution time for the same set of devices, and discuss the observed trade-offs and trends in the scheduler-configuration design space based on these metrics, across memory generations.

Index Terms—dynamic random access memory (DRAM), Memory control and access, Real-time and embedded systems

1 Introduction

The past decade has seen the introduction of many new SDRAM generations, in an effort to ward off the memory wall problem [1]. New generations bring advances in peak performance, increasing the memory clock frequency to improve maximum bandwidth, while exploiting the improvements of CMOS technology to reduce the operating voltage and limit the energy usage. Specialized low-power architectures with a strong focus on energy efficiency also emerged to cater to use-cases involving battery-powered applications, leading to the introduction of wider, low-power memory devices [2]. The drive behind these developments is improvement of the typical or average-case performance of applications using the memory.

The hard real-time application domain is less focused on the average case. Instead, the worst-case behavior of the memory determines the guaranteeable bandwidth, which could be just a small fraction of the peak bandwidth expected by multiplying the transfer rate by the data bus width. Memory controllers used for hard real-time or safety-critical applications, like in automotive or avionics, have to use the SDRAM in such a way that application requirements are always satisfied in the worst case, i.e. guarantee enough bandwidth and a sufficiently low response time, possibly within a fixed energy or power budget. Here, we focus on the worst-case performance impact of the command scheduler and the low-level memory map that distributes data across the banks in the SDRAM.

Existing works focus on a single memory device or on one or two memory generations, and hence a multi-generation overview of the power/performance trade offs for real-time SDRAM controllers is not available. The goal of this article is therefore threefold. Firstly, we compactly summarize the command scheduling rules for LPDDR1/2/3, and DDR2/3/4 memories and provide an abstraction from the differences between them. Secondly, we define a common command scheduling algorithm for these memory generations to enable quantification of the worst-case power/performance of a memory device. Thirdly, we apply this scheduling algorithm to 12 devices and evaluate the worst-case power/performance trade-offs across 6 memory generations for different scheduler and memory map configurations.

This article contains the following contributions: 1) *A generalized, multi-generation close-page memory command scheduling algorithm*. High-level command-to-command constraints are defined (Section 3.1), abstracting from the generation-specific timings and allowing a generic solution for the scheduling problem. Two parameters, the number of banks interleaved (BI) and bursts per bank (BC), characterize the memory map that the scheduler uses. We show that each (BI, BC) combination has a different trade-off between bandwidth, power (energy efficiency) and request execution time (Section 3.2). DDR4, which introduces *bank groups* [3] as a new architectural feature with a corresponding set of new scheduling constraints, is identified as a generation that benefits from new scheduling algorithms and memory maps. We propose *2) a pairwise bank-group interleaving scheme for DDR4* that exploits bank grouping

- *Sven Goossens and Kees Goossens are with the Eindhoven University of Technology, Eindhoven, The Netherlands. Email: {s.l.m.goossens, k.g.w.goossens}@tue.nl*
- *Karthik Chandrasekar is with Delft University of Technology, Delft, The Netherlands.*
- *Benny Akesson is with Czech Technical University in Prague, Prague, Czech Republic.*

Manuscript received April 19, 1970; revised September 17, 1970.

for improved performance (Section 3.3). To evaluate the quality of the scheduling algorithm, we create *3) a parameterized Integer Linear Programming (ILP) formulation of the command scheduling problem*. It uses the same abstraction as the scheduling algorithms, and generates optimal close-page schedules for all considered memory types and (BI, BC) combinations. With it, we show that our scheduler obtains optimal schedule lengths for the majority of the considered memory devices and configurations (Section 4.1). Finally, we give *4) an overview of worst-case SDRAM performance trends* in terms of bandwidth, power, and execution time of a request, based on the proposed scheduling algorithms. We evaluate the performance differences across speed bins within generations and across memory generations (Section 4.2).

The rest of this article is organized as follows: Section 2 supplies background information on SDRAM and real-time memory controllers. Section 3 explains the contributions in detail, Section 4 evaluates the results, followed by related work in Section 5 and conclusions in Section 6.

2 BACKGROUND

This section briefly introduces SDRAM in Section 2.1 and real-time memory controllers in Section 2.2. Section 2.3 and Section 2.4 then introduce the concept of memory patterns, and burst grouping, respectively.

2.1 SDRAM

SDRAM devices contain a hierarchically structured storage array [4]. Each device consists of typically 8 or 16 *banks* that can work in parallel, but share a command and data bus. The execution of commands on different banks can therefore be pipelined. A bank consists of a memory array, divided into rows, each row containing a certain number of columns. A column is as wide as the number of pins on the memory device's data bus, and hence only one bank can drive the data pins at a time.

There are six main SDRAM commands. An activate (ACT) command opens a row in a bank, and makes it available for subsequent read (RD) and write (WR) operations. Each RD or WR command results in a *burst* of data. The number of words per RD or WR is determined by the burst length (BL) setting. Across memory generations the commonly supported value is 8. It takes BL/2 (command) clock cycles to transfer a burst, since we consider Double Data Rate SDRAMs. We abbreviate BL/2 by B in equations and figures in this article. Clock frequencies in this article refer to the command clock of the memory. Only one row per bank can be open at a time. The precharge (PRE) command closes a row, i.e. it stores it in the memory array, allowing for another row to be subsequently opened. An optional *auto-precharge* flag can be added to RD and WR commands, such that the associated row is closed as soon as the read or write is completed. A RD or WR with auto-precharge can be regarded as a RD or WR, followed by a regular PRE command from a timing perspective. The difference is that the precharge does not require an explicit slot in the schedule, i.e. it may overlap other commands. SDRAM is volatile, so it needs to be refreshed periodically by issuing a refresh (REF) command to retain its data. Finally, the NOP command does nothing. The figures in this article abbreviate ACT, RD and WR by A, R, and W, respectively, and encode NOPs as empty boxes.

Vendors characterize their memory chips by specifying their *timings*. The standards [3], [5], [6], [7], [8], [9] explain in detail what each timing represents within the SDRAM. For the purpose of this article, these details are less important, since we consider the memory as a black box, that we merely have to use according to its interface specification, and therefore we simply use the timings as a label for a specific amount of time. In this article, we typeset timings in SMALL CAPS. *Timing constraints* are built as mathematical expressions from these timings, and they define the minimum time between pairs of commands based on the state of the memory, which in turn is a consequence of earlier executed commands. Some constraints only restrict commands for a single bank, like RC and RCD for example, while others like Four Activate Window (FAW) and RRD, act at the device level. An SDRAM controller has to satisfy the timing constraints to operate correctly. To see all timings in context within their constraints, you may refer to Table 1 and Table 2.

DDR4 introduces *bank groups*: banks are clustered into (at least two) bank groups per device. Banks in a bank group share power-supply lines. To limit the peak-power per group, sending successive command to the same group is discouraged by making certain timings larger in this case. These timings are post-fixed with _L (long) or _S (short) for commands for the same or a different bank group, respectively. Successive RD or WR commands to the same group need to be separated by at least CCD_L cycles. Because CCD_L is larger than the number of cycles per data burst (B), performance is impacted by CCD_L unless bursts are interleaved across bank groups.

SDRAM devices can be used as standalone chips, as generally done in embedded Systems-on-Chips [10], [11], [12] for example. The memory is then used at its native Interface Width (IW), typically ranging from 4 up to 32 bits. Bigger and wider memories can be built by having multiple chips work in lock-step in a *rank*, executing the same commands, producing or consuming data in parallel. The combined data bus width can then be seen as the IW of the memory. Multi-device setups are typically used in general-purpose and high-performance computer systems. Ranks can share a command and data bus, as long as they do not drive the data bus simultaneously. A memory hierarchy may contain multiple independent groups of ranks called *channels*, each with an individual memory controller. Here, we focus on the memory performance per channel. For multi-channel considerations in real-time memory controllers you may refer to [13].

2.2 Real-Time Memory Controllers

A memory controller is the interface between one or more *memory clients* and the SDRAM. Clients interact with the memory controller through read or write *requests*. A request accesses a certain amount of data, depending on the properties of the client. The request size divided by IW and BL determines how many data bursts, and thus RD or WR commands should be generated for a request, with a minimum of one burst. The *access granularities* we consider in this article vary from a single SDRAM burst, up to multiple grouped bursts to a size of 256 bytes. This range thus includes typical cache-miss requests sizes (8 to 64 bytes), up to larger DMA [14] or accelerator-based [15] requests.

Requests are queued in the controller, selected by a (predictable) inter-client arbiter, and then translated into SDRAM commands that are executed by the SDRAM device. Memory controllers that leave a row open after a request is completed use an *open-page policy*, while those that close it as soon as possible use a *close-page policy* [16].

A memory controller has to *schedule* the commands it generates to the SDRAM. The scheduling process is complex, since all timing constraints need to be satisfied for each individual command, while each scheduling decision changes the memory state and thus the constraints that need to be taken into account for future decisions. Additionally, a scheduler may have to choose between multiple schedulable commands without a clear indication of the impact on performance and future scheduling options. This leads to a type of emergent behavior that is hard to predict, and thus memory performance is hard to bound in the general case; for many commercial controllers no analytical bounds can be provided. However, real-time memory controllers should provide hard bounds on the time to serve all requests to assure application-level deadlines are always satisfied.

2.3 Memory Patterns

This article reasons about real-time memory controller performance in terms of *memory patterns*, as defined in [17]. A memory pattern is a small series of scheduled SDRAM commands with known length and function. Memory patterns implement a close-page policy. Six basic patterns exist: *1) read* and *2) write* patterns activate, issue read or write commands respectively, and then precharge. An underlying assumption is that subsequent requests may use these patterns to access a different row in the same bank(s) in the worst-case, and are hence necessarily serviced sequentially. Read and write patterns are constructed based on that notion, i.e. they should be repeatable after themselves without violating timing constraints, even when they target the same banks. Two switching patterns, *3) read-to-write switch* and *4) write-to-read switch*, made up of only NOPs, are inserted between read and write patterns when the data bus direction has to change. The *5) refresh* pattern consists of a refresh command surrounded by the appropriate number of NOPs to satisfy the timing constraints related to it. It can be scheduled after a read or write pattern. Finally, the idle time of the controller can be discretized explicitly into *6) idle or power-down* patterns [18]. A collection of these six patterns is called a *pattern set*.

Some close-page real-time controllers use variations of memory patterns in their architecture [19], [20], scheduling patterns from such a set instead of individual commands. Others define patterns only in their worst-case analysis [21], [22], knowing the behavior of their architecture is bounded by them. In both cases, the analysis complexity is greatly reduced.

The worst-case data bus utilization on the SDRAM interface can be determined for a given pattern set, as shown in [17]. We refer to this metric as *memory efficiency* (see Figure 1). The basic idea is to find the sequence of patterns, like continuously writing or alternating reads and writes for example, that transports the least amount of data in a unit of time. No assumptions on the inter-client arbiter, nor on the types of requests (read/write) are made when determining this efficiency, which makes this metric independent of the arbiter type. The memory efficiency is the guaranteed fraction of the peak bandwidth, and hence multiplying those two numbers yields the *worst-case bandwidth* that the controller can distribute amongst its clients.

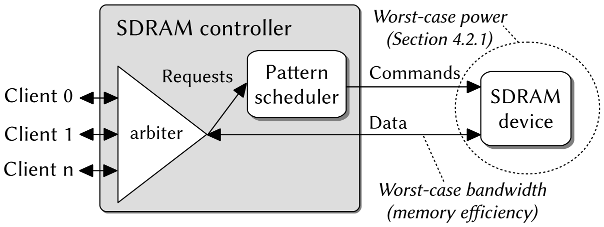


Fig. 1. Simplified controller architecture, showing two of the performance metrics we use in this article to characterize SDRAM performance.

2.4 Burst Grouping

Accessing an SDRAM with single-burst requests is highly inefficient in the worst case. Assuming a close-page policy is used, the memory has to spend at least RC cycles per burst, or around 50 ns for the memories considered in this article, due to overhead of activating and precharging. This is significantly more than the actual data transfer time (4 cycles assuming a burst length of 8), which is 5 ns for a DDR3-1600 for example, a factor 10 difference (see Figure 2a). The worst-case bandwidth is thus a lot smaller than the peak bandwidth obtained by only considering the data rate, and this efficiency gap grows as the memory clock frequency increases.

To increase worst-case efficiency, real-time controllers typically use a (minimum) *access granularity* that is larger than single bursts. Instead, multiple bursts are grouped together to form a single atomic access. Each request is mapped to an integer multiple of these atoms. The relative order of bursts within one such access or *atom* is fixed. This gives each memory access (guaranteed) properties that improve the worst-case. Grouping bursts creates:

1. *Bank parallelism*: The atom is interleaved over multiple banks that work in parallel to produce or consume data. The time between accesses to the *same bank* is increased, giving it more time to precharge and activate while other banks are accessed, leading to improved efficiency.
2. *Consecutive bursts access the same row*: Multiple bursts are fetched from the same row in the same bank within an atom, in essence generating guaranteed locality across those bursts.

Some real-time memory controllers interleave requests over all available banks [19], [20], [21], [22]. [20] also considers the number of bursts per bank as configuration parameter, but not the number of banks. This article generalizes these concepts, and shows the configuration trade-offs when both degrees of freedom are used.

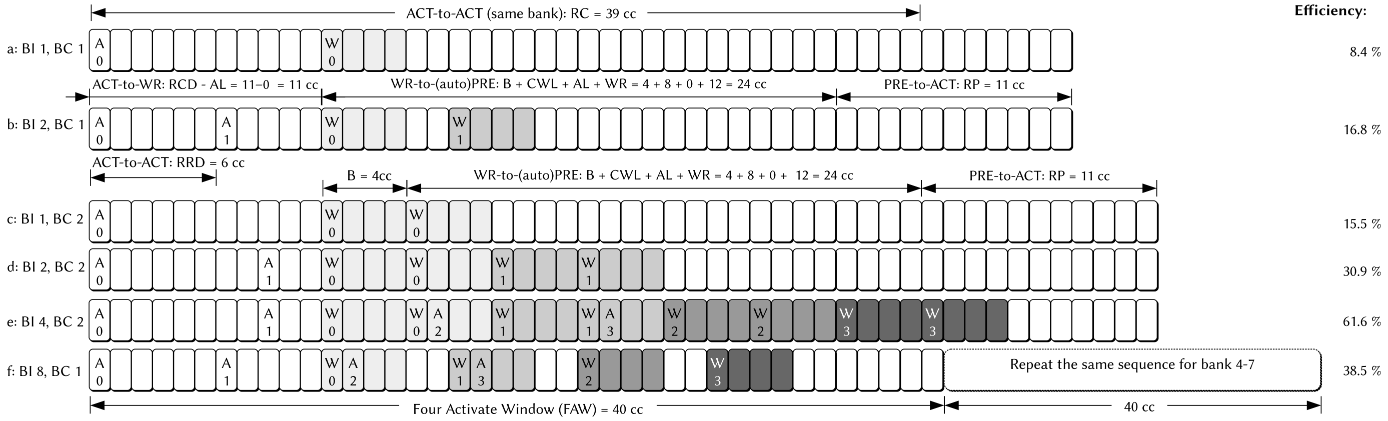

ACT-to-ACT (same bank): RC = 39 cc
Efficiency:
a: BI 1, BC 1
8.4 %
ACT-to-WR: RCD - AL = 11−0 = 11 cc
WR-to-(auto)PRE: B + CWL + AL + WR = 4 + 8 + 0 + 12 = 24 cc
PRE-to-ACT: RP = 11 cc
b: BI 2, BC 1
16.8 %
ACT-to-ACT: RRD = 6 cc
B = 4cc
WR-to-(auto)PRE: B + CWL + AL + WR = 4 + 8 + 0 + 12 = 24 cc
PRE-to-ACT: RP = 11 cc
c: BI 1, BC 2
15.5 %
d: BI 2, BC 2
30.9 %
e: BI 4, BC 2
61.6 %
f: BI 8, BC 1
Repeat the same sequence for bank 4-7
38.5 %
Four Activate Window (FAW) = 40 cc
40 cc

Fig. 2. The effects of burst grouping for a DDR3L-1600 device [23]. Each box is a command and has an optional bank id, shading indicates data bus activity, empty boxes are NOPs. a) shows a single burst access, b) demonstrates bank interleaving, d) combines bank interleaving with an increased burst count per bank. Efficiencies are derived based on the algorithm in [17], which also takes read/write switches and refresh overhead into account.

3 Pattern-Based Command Scheduling

This section shows how memory patterns are generated for hard real-time memory controllers. Section 3.1 abstracts the different timing constraints from the scheduling algorithm for the considered memory generations. Section 3.2 and Section 3.3 discuss pattern generation in general and for DDR4, respectively, and Section 3.4 explains the connection between the pattern configuration and memory map. Section 3.5 introduces an ILP formulation of the scheduling problem we later use to evaluate the pattern generation heuristics. The produced schedules are the basis for the worst-case performance evaluation in Section 4.

3.1 Generalized Command Scheduling Rules

The command scheduling rules have not changed much across the different generations of SDRAM that were introduced over the years [3], [5], [6], [7], [8], [9]. The exact timings that govern the memory behavior vary, but these details can be hidden from scheduling algorithms, since they only have to know about the minimum delay between sets of commands. We introduce a function d, which serves as the interface for these algorithms to obtain the minimum relative delay between two commands, cmd_a and cmd_b. Based on 5 properties of these command and the memory type, it indexes a lookup table to determine the delay. The first two properties describe the types of the commands, $type_a, type_b \in \{\text{ACT, RD, WR, PRE, REF}\}$, while the remaining boolean properties specify the relative physical location of the bank at which the commands are targeted, i.e. if they go to the same or a different rank, bank group (DDR4) or bank, respectively:

$$\begin{aligned} d(cmd_a, cmd_b) = LUT(&type_a, type_b, \\ &sameRank, sameBankGroup, \\ &sameBank, memoryType) \qquad (1) \end{aligned}$$

By using this function, scheduling algorithms can be written in a compact, memory-type-agnostic manner, as demonstrated in Algorithm 1.

Based on the JEDEC specifications [3], [5], [6], [7], [8], [9], we collected the delays that this function should produce for 6 SDRAM generations in Table 1 and Table 2, effectively condensing hundreds of pages of documentation into the bare minimum required to create memory command scheduling algorithms. If a combination of inputs is not mentioned in the tables, then it is either unconstrained, or not allowed by the bank state-machine. For DDR4, the timings that are post-fixed with _x depend on the $sameBankGroup$ argument, selecting the _L or _S versions of the timing if $sameBankGroup$ is true or false, respectively, as required by the specification [3]. For other memory types, the non-postfixed version of the timing is used. The FAW timing is not mentioned in the table, because it is a window-based constraint, and not a simple delay. It has to be taken into account separately on a per-rank basis for all memory types except LPDDR, which has no FAW constraint.

The remainder of this article evaluates the worst-case performance under a close-page policy. The impact of multi-rank effects on this metric is minimal, as quantified in our discussion of [24] in Section 5. For brevity, and because multi-rank operation is not standardized, the tables only show constraints for the cases where the command pair is sent to the same rank. Commands across ranks are generally not constrained, unless they use the (shared) data bus, i.e. RD and WR commands. An additional delay has to be taken into account in those cases to assert only one rank at a time drives the bus.

3.2 Pattern Generation

Memory patterns for all devices in the previously mentioned SDRAM generations can be created by respecting the constraints from Section 3.1, but the distribution of the bursts in the pattern across banks and the relative order of commands for different banks remain to be decided. Instead of interleaving each request over all banks in the memory [19], [20], [21], [22], or exploiting no inter-request bank parallelism at all [24], [25], *we treat the number of banks involved in executing a request as a free parameter*. With this extra degree of freedom, both the number of banks that

TABLE 1
Common constraints across SDRAM types.

$type_a$	$type_b$	sameBank	Constraint
ACT	ACT	True	RC
ACT	ACT	False	RRD_x
ACT	PRE	True	RAS
ACT	RD/WR	True	RCD - AL
PRE	ACT	True	RP
PRE	REF	don't care	RP
REF	ACT	don't care	RFC

TABLE 2
SDRAM-type specific constraints[1].

Memory type	$type_a$	$type_b =$ PRE sameBank = True	$type_b =$ RD sameBank = don't care	$type_b =$ WR sameBank = don't care
LPDDR	RD	B	B	B + CL
LPDDR	WR	B + DQSS + WR	B + DQSS + WTR	B
DDR2	RD	B + AL - 2 + max(RTP, 2)	B	B + RTW
DDR2	WR	B + WL + WR	B + CL - 1 + WTR	B
DDR3	RD	AL + max(RTP, 4)	B	B + RL - CWL - AL + 2
DDR3	WR	B + CWL + AL + WR	B + max(0, CWL + WTR)	B
DDR4	RD	AL + RTP	CCD_x	B + RL - CWL - AL + PA
DDR4	WR	B + CWL + AL + WR	B + max(0, CWL + WTR_x)	CCD_x
LPDDR2/3	RD	B + max(0, RTP - D)	B	B + RL - WL + DQSCK$_{max}$ + 1
LPDDR2/3	WR	B + WL + WR + 1	B + WL + WTR + 1	B

[1] B represents the burst transfer time, equal to the burst length (BL) divided by 2 (double data rate). For DDR2, RTW = 2 if BL = 4, and 6 if BL = 8. For DDR4, PA depends on the selected read/write preamble. If a read- or write preamble of 1 cycle is used, PA = 2. With a preamble of 2 cycles, PA = 3. For LPDDR2/3, D = 1, 2, or 4 for LPDDR2-S2, LPDDR2-S4, or LPDDR3 devices, respectively.

are accessed by a pattern (Bank Interleaving (BI)) and the number of bursts per bank (Burst Count (BC)) are configurable, generating a range of possible *pattern configurations* characterized by a (BI, BC) combination. BI can be equal to or smaller than the number of banks in the memory. In the worst-case, successive requests access a different row in the same set of (BI) banks. BI and BC effectively define the low-level memory map for bursts (see Section 3.4).

Figure 2 illustrates how the schedules change for different (BI, BC) combinations using examples. Figure 2b demonstrates the benefit of bank interleaving. Two bursts are interleaved, i.e. BI = 2. The ACT-to-RD/WR delay (RCD) of bank 1 is (partially) hidden by the data access to bank 0.

Increasing BC enables hiding the ACT-to-ACT constraint between banks (RRD_X). This is relevant for all memory devices for which the maximum activate command rate is lower than the read/write command rate (RRD_X $>$ B), as is the case in Figure 2b, where it causes the two cycle gap in the data transfer between the burst to bank 0 and bank 1. Figure 2c and Figure 2d show how this issue is resolved when BC is increased.

For memories that have more than 4 banks, configurations with BI $\leq$ 4 are of particular interest, since they deal better with the FAW constraint. With at most 4 activate commands within a pattern, the FAW can only play a role if multiple consecutive patterns are considered. This allows NOPs inserted at pattern edges to satisfy the FAW constraint to overlap with NOPs that resolve other constraints, like RC and RP for example, and hence these configurations are more efficient. Figure 2e and f illustrate this: the schedules contain the same number of bursts, but Figure 2e avoids the FAW penalty, increasing the efficiency from 38.5% to 61.6%. Efficiency generally increases with the number of bursts per pattern, because the activate/precharge overhead, expressed by the idle-times at the start and end of the pattern, is amortized over more and more data. In practice, the access granularity and by extension the product of BI and BC, is limited by the size of the requests the memory clients generate, since there is no point in fetching data when it has to be discarded later because the client is not interested in it. Therefore, configurations are only straightforwardly interchangeable and comparable if they implement an access granularity that is equal to or smaller than the size of the requests.

We use the *bank-scheduling heuristic* described in [26] as a starting point for the order and placement of the commands, because it has been shown to perform well for DDR2/3 memories. In this heuristic, read and write patterns are created independently. Within these patterns, read or write commands are scheduled as soon as possible, accessing banks in ascending order. Activate commands are scheduled as late as possible, but just in time to not delay the read or write commands. Typically this is RCD cycles before the first read or write to the associated bank, or earlier if this cycle is already taken by another command. Patterns start with bank activation, and the final access to a bank has an auto-precharge flag. This heuristic is extended to include the new BI parameter, and we refer to bank scheduling with variable bank interleaving as BS BI. Algorithm 1 shows the most relevant read and write pattern generation functions, a complete executable version can be found in [27]. For BS BI, 'useBsPbgi' should be set to false.

Algorithm 1 builds up the pattern in the set $\mathbf{P}$, of which each element is a 3-tuple representing the type, bank, and clock cycle (cc) of a command. Record/struct-like semantics are used to access the elements in the tuple, i.e. $x.cc$ accesses the clock cycle element of the tuple. Most functionality is based on the EARLIEST function, which returns the earliest cycle at which a command cmd$_b$ may be scheduled, given the location of the commands in (partial) pattern $\mathbf{P}$. It uses the 'd' function (Equation (1)), which symbolizes the lookup action in Table 1 and Table 2.

The nested loops (lines 4-8) generate 1 activate and BC read or write commands per bank. The ADDACTANDRW function schedules an activate command using ADDACT before the first burst to a bank (lines 17-18). Additionally, it schedules the read/write commands as soon as possible (lines 16, 19). ADDACT first finds a range of possible locations for an ACT command. A lower bound (lb) for the location is based on the ACT-to-ACT constraints (lines 22-23). An upper bound (ub) is determined by the minimum distance between the planned location of the RD/WR

Algorithm 1 Bank scheduling heuristic for BS BI and BS PBGI

```
 1: function PATTERNGEN(BI, BC, rdOrWr, useBsPbgi)
 2:     BGi := 2 if BI > 1 and useBsPbgi == true else 1
 3:     $\mathbf{P}$ := {} // The pattern
 4:     for all bankPair $\in \{0 \ldots BI/\text{BGi} - 1\}$ do
 5:         for all burst $\in \{0 \ldots \text{BC} - 1\}$ do
 6:             for all offset $\in \{0 \ldots \text{BGi} - 1\}$ do
 7:                 bnk := bankPair ·BGi + offset
 8:                 $\mathbf{P}$ := ADDACTANDRW(bnk, rdOrWr, burst, $\mathbf{P}$)
 9:     $\mathbf{P}'$ := $\mathbf{P}$ // A copy with explicit precharges.
10:     for all bnk $\in \{0 \ldots \text{BI} - 1\}$ do
11:         preCc := EARLIEST((PRE, bnk, 0), $\mathbf{P}$)
12:         $\mathbf{P}'$ :=$\mathbf{P}' \cup$ {(type: PRE, bank: bnk, cc: preCc)}
13:     return MAKEREPEATABLE($\mathbf{P}$, $\mathbf{P}'$)

14: function ADDACTANDRW(bnk, rdOrWr, burst, $\mathbf{P}$)
15:     rw := (type: rdOrWr, bank: bnk, cc: 0)
16:     rwCc := EARLIEST(rw, $\mathbf{P}$)
17:     if burst == 0 then
18:         $\mathbf{P}$, rwCc := ADDACT(rw, rwCc, $\mathbf{P}$)
19:     return $\mathbf{P} \cup$ {(type: rdOrWr, bank: bnk, cc: rwCc)}

20: function ADDACT(rw, rwCc, $\mathbf{P}$)
21:     act := (type: ACT, bank: rw.bank, cc: 0)
22:     lb := EARLIEST(act, $\mathbf{P}$)
23:     lb := lb + REMAININGFAWCYCLESAT(lb, $\mathbf{P}$)
24:     while true do
25:         ub := rwCc − d(act, rw)
26:         $\mathbf{S}$ := $\{i \in \{\text{lb} \ldots \text{ub - 1}\} \mid cmd.cc \neq i \;\forall\; cmd \in \mathbf{P}\}$
27:         if $|\mathbf{S}| \neq 0$ then
28:             $\mathbf{P}$ :=$\mathbf{P} \cup$ {(type: ACT, bank: rw.bank, cc: $\max(\mathbf{S})$)}
29:             return $\mathbf{P}$, rwCc
30:         rwCc := rwCc + 1

31: function MAKEREPEATABLE($\mathbf{P}$, $\mathbf{P}'$)
32:     len := $\max(\{cmd.cc \;\forall\; cmd \in \mathbf{P}\})$ + 1
33:     for all cmd$_b \in \mathbf{P}$ do
34:         len := max(len, EARLIEST(cmd$_b$, $\mathbf{P}'$) - cmd$_b$.cc )
35:     while not FAWSATISFIEDACROSS(len, $\mathbf{P}$) do
36:         len := len + 1
37:     return $\mathbf{P}$, len

38: function EARLIEST(cmd$_b$, $\mathbf{P}$)
39:     // d() is a lookup in Table 1-2 (it returns -inf if
40:     // the command combination is not mentioned).
41:     return max({cmd$_a$.cc + d(cmd$_a$, cmd$_b$) $\forall$ cmd$_a \in \mathbf{P}$}
                ∪{0})
```

command (rwCc) and the ACT command (line 25). Set $\mathbf{S}$ contains the cycles within this range which are not occupied by other commands (line 26). If this set is not empty, then the largest option is chosen, scheduling the ACT as late as possible (lines 27-28). Otherwise the RD/WR command is postponed by a cycle, and by extension the upper bound for the ACT shifts forward, until a suitable location is found (lines 24, 30).

What remains is to determine the location of the precharge commands (lines 9-12), which are stored in a separate copy of the pattern ($\mathbf{P}'$) since they are implemented using auto-precharge flags and hence are not explicitly scheduled. Their location is still relevant, because the precharges can constrain commands that follow them. Given the commands in $\mathbf{P}'$, MAKEREPEATABLE finds the minimum length the pattern should have to be repeatable after itself without violating regular constraints (lines 32-34) and the FAW constraint (lines 35-36) spanning across pattern incarnations. Each time the length is increased, one NOP is implicitly added to the end of the pattern. Finally, the scheduled commands and the length of the pattern are returned and the algorithm ends.

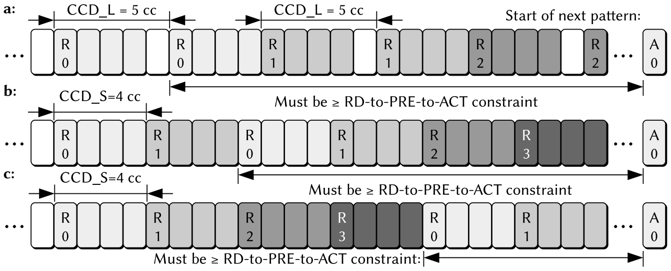


Fig. 3. (Partial) DDR4-1866 read pattern. Odd and even banks are in a different bank group. Schedule *a* does not use (BS PBGI), while *b* does. *c* shows how the distance to the next activate in a following pattern reduces as more bank groups are interleaved, resulting in longer (less efficient) patterns.

3.3 DDR4 Pattern Generation

To generate more efficient DDR4 patterns and avoid the CCD_L constraints (see Section 2 and [3]), read or write commands should be interleaved across bank groups. To this end, we propose a *pairwise bank-group interleaving* heuristic, as demonstrated in Figure 3. Two banks from different bank groups are paired together. In contrast to regular bank scheduling, which finishes all BC bursts to a bank before switching to the next bank, the read or write commands of such a pair are interleaved per burst. The remaining rules of the heuristic are the same as described in Section 3.2. We refer to bank scheduling with pairwise bank group interleaving as BS PBGI. Setting 'useBsPbgi' to true in Algorithm 1 generates the proposed interleaving.

If BI ≥ 2, BC ≥ 2, then this heuristic behaves differently from BS BI. Pairwise interleaving reduces the length of the data transferring portion of the pattern by $(\text{CCD\_L} - \text{CCD\_S}) \cdot$ BI $\cdot$ (BC $-$ 1) cycles. The advantage of interleaving only two instead of for example four bank groups, is that the last access to the first bank pair happens relatively early in the pattern. This allows the RD/WR-to-PRE-to-ACT delay to happen (partially) in parallel with accesses to other bank pairs. Interleaving more than two banks reduces the overlap, and could hence lead to patterns that require more NOPs at the end of the pattern to satisfy the constraints required to repeat the pattern, without any benefits.

3.4 Memory Maps

The order of the data bursts within a memory pattern is fixed, which means they are mapped to consecutive physical memory addresses by definition. The pattern configuration (BI, BC) thus has a direct influence on the decoding of the (least significant) portion of the physical address, as it partially determines which bits should be selected for the bank, row and column address. This is illustrated in Figure 4, which shows how the lower $log_2(BI \cdot BC \cdot BL \cdot IW/8)$ bits are mapped to the column and bank address Least Significant Bits (LSB), respectively. The connection to the memory

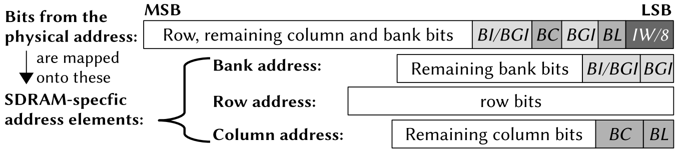


Fig. 4. Memory map from physical address to the SDRAM-specific address components. For example, $log_2(BI/BGI)$ bits from the corresponding position in the address are used in the similarly marked position in the bank address.

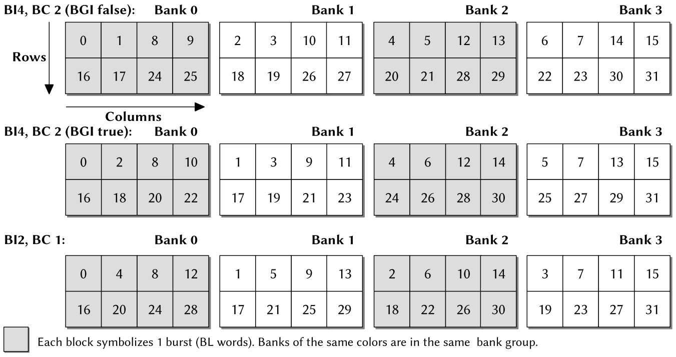


Fig. 5. Three memory map examples, showing where the bursts of requests to consecutive physical addresses (separated by the access granularity) are written. The third configuration, using $(2, 1)$ behaves the same regardless of the BGI setting.

map practically limits the possible values for BI and BC to powers of two. The remaining portion of the address can be mapped freely such that separate memory regions are generated for different clients (spatial partitioning).

Figure 5 shows three data layout examples in an extremely small 4-bank memory, resulting from different (BI, BC) and 'useBsPbgi' settings (written as BGI in the figure). We map the bits that are not bound to the pattern configuration (the white part in Figure 4) such that for consecutive physical addresses the bank changes first, followed by the column, and finally the row.

3.5 ILP-Based Command Scheduling Problem

Section 3.2 and Section 3.3 describe two heuristics, BS BI and BS PBGI, which generate close-page memory patterns. The length of these patterns can be used as a measure for their quality, since it determines the worst-case memory performance, as later shown in Section 4.2. The commands in a pattern are chosen and fixed once a (BI, BC) combination is selected, so we can define a pattern as optimal in terms of length if there is no other permutation of this set of commands satisfying pattern scheduling rules and timing constraints resulting in a shorter pattern.

To generate these optimal patterns, we use a parameterized ILP formulation of the command scheduling problem. Based on a (BI, BC) combination and an implementation of Equation (1), we create an ILP problem that when solved, finds the optimal pattern size and the location of the commands within the pattern. Any memory controller that uses a close-page policy and relies on patterns in analysis or implementation, like [19], [20], [21], [22] can use this formulation to improve the schedules it uses, or to extend its scope to different memory devices or generations.

We use the ILP formulation as a means to evaluate the heuristics, and the translation to a formal problem definition itself is not particularly interesting given the scope of this article. Therefore, we only describe the formulation in natural language. The parameterized ILP problem is available in [27], as well as the exact instantiation we used in the evaluation. Figure 6 illustrates a subset of the properties of the formulation.

1. Create a set of variables representing the locations of the commands in the pattern that should be generated as a function of the selected (BI, BC) combination: BI ACT commands, $BI \cdot BC$ RD/WR commands, and BI PRE commands (as auto-precharge flags).
2. Add variables representing the location of an *extra activate command* for each bank in this set. These activate commands represent the start of a second instance of the pattern, which should be schedulable immediately after the first instance (read/write patterns should be repeatable after themselves (Section 2.3)).
3. Given this set of memory commands, assign a *single* location in the schedule to each command such that:
   a. An ACT to bank 0 is scheduled in cycle 0.
   b. No two commands are scheduled in the same cycle. Auto-precharges are exempted from this rule, since they do not require a slot in the schedule.
   c. The relative delays between any pair of commands is at least as large as prescribed by Equation (1), and there are at most four ACTs in each FAW window.
   d. The command sequence for each bank start with an activate, followed by BC read or write commands, followed by a precharge. This still allows different banks to be used in parallel, i.e. one can be activating while another is used for reading or writing. The extra activate commands added in step 2 should happen after the precharge to the associated bank.
   e. Commands for the second instance of the pattern cannot be scheduled before the extra activate to bank 0, and commands for the first instance need to be scheduled before the extra activate to bank 0. This activate command itself and all precharge commands are exempt from this rule. Precharges may be (automatically) pipelined across patterns, because auto-precharge flags are used and they are hence scheduled automatically.
   f. Both instances of the pattern are the same. A set of constraints enforces that the distance between the extra activate command to a bank and the extra activate to bank 0 is equal to the distance between the first activate command to that bank and cycle 0.
4. To limit the search space and eliminate equivalent symmetric solutions, we add the following constraints (see Figure 6):
   a. The order of read or write commands to the same bank is fixed, because we do not want to distinguish different bursts to the same bank.
   b. Banks are activated and precharged in ascending or-

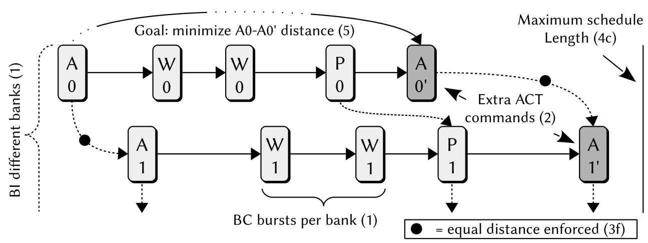


Fig. 6. ILP precedence constraints. An edge between two commands means that the source command must be scheduled before the destination command. Round brackets refer to rules in the ILP description.

der. For DDR4, we map consecutive bank ids to different bank groups (and wrap around once the bank groups run out).

c. An upper bound on the length of the patterns in cycles can be found based on the BS BI or BS PBGI (DDR4) heuristics. Both provide a valid bound, so we use whichever is the smallest.

d. A lower bound for the pattern length is the size of a schedule where the commands for bank 0 are scheduled as soon as possible. A lower bound for the location of the extra activate commands of other banks is derived from this bound.

5. The optimization goal is to minimize the pattern length. Therefore, we minimize the location of the second activate in to bank 0, which marks the start of the next incarnation of the pattern.

The ILP formulation might create shorter patterns than BS BI and BS PBGI, because it does not restrict the relative ordering of bursts across banks nor the placement of bursts within the pattern, and it has no preferred location for activate commands (i.e. it could postpone them).

4 Evaluation

Two novel pattern generation heuristics have been presented. This section compares the schedules lengths they produce to those generated by the ILP instance for the same scheduling problem in Section 4.1. The worst-case performance of the selected memory devices as a function of the different (BI, BC) configurations is evaluated in Section 4.2.

The evaluation is limited to two devices per memory type for the purpose of this article. Each device is part of a speed bin defined by the associated JEDEC standard for the memory type. Speed bins are selected based on the commercial availability of the device and data sheets at the time of writing, the range of clock frequencies they cover (we select a device from the slowest and fastest bin if available), and comparability with speed bins of other memory types (select common speeds and data bus widths as much as possible). Table 3 shows the specifications of the selected devices. All devices are made by the same vendor, since this makes it more likely that consistent safety (σ) margins have been applied to the specifications in the data sheet to compensate for variation [28]. This is especially important for the IDD current measures we supply to the power model in Section 4.2, as it makes the evaluation across devices fairer. Furthermore, it is important to note which data sheet revision and *die revision* is used in the comparison, since DRAM manufacturers frequently update both documentation and the design of their chips. ([27] contains the detailed specifications used for the experiments).

TABLE 3
Memory specifications.

Name	Clock frequency	Data bus width	Capacity	Part number	Die revision
LPDDR-266	133 MHz	x16	1 Gb	MT46H64M16LF -75	B
LPDDR-400	200 MHz	x16	1 Gb	MT46H64M16LF -5	B
DDR2-800	400 MHz	x16	1 Gb	MT47H64M16 -25E	H
DDR2-1066	533 MHz	x16	1 Gb	MT47H64M16 -178E	H
DDR3-1066	533 MHz	x16	1 Gb	MT41J64M16 -178E	G
DDR3L-1600	800 MHz	x16	4 Gb	MT41K256M16 -125	E
LPDDR2-667	333 MHz	x32	2 Gb	MT42L64M32D1 -3	A
LPDDR2-1066	533 MHz	x32	2 Gb	MT42L64M32D1 -18	A
LPDDR3-1333	667 MHz	x32	4 Gb	EDF8132A1MC -15	1
LPDDR3-1600	800 MHz	x32	4 Gb	EDF8132A1MC -125	1
DDR4-1866	933 MHz	x8	4 Gb	MT40A512M8 -107E	A
DDR4-2400	1200 MHz	x8	4 Gb	MT40A512M8 -083E	A

4.1 Pattern Length Evaluation Using ILP

The length of a pattern determines the efficiency of a pattern set; a shorter pattern is preferred over a longer pattern if it transports the same amount of data. To evaluate the quality of the BS BI and BS PBGI heuristics, we use the ILP formulation from Section 3.5.

4.1.1 Evaluation for Non-DDR4 Memories

The ILP formulation and the BS BI heuristics are used to generate read and write patterns for the selected memories (Table 3), for all (BI, BC) combinations with access granularities up to 256 bytes, and we compared the resulting pattern lengths. For the non-DDR4 memories, we conclude that *the bank scheduling heuristic generates patterns of the same length as the ILP formulation for all considered devices except LPDDR3-1333*, and is hence optimal for most devices given the scheduling constraints.

For two LPDDR3-1333 configurations ($(4, 2)$ and $(2, 4)$) is BS BI non-optimal; the write patterns are 1 cycle too long in these cases. The optimal pattern has an idle cycle on the data bus the middle of the pattern, which the heuristic always tried to avoid. Given how exceptional this effect is (2 out of 120 non-DDR4 configurations are affected), and its relatively low cost (<2% length increase), we do not explore this further.

4.1.2 Evaluation for DDR4 Memories

For the DDR4 memories, we generate patterns using the BS BI and BS PBGI heuristics and the ILP formulation. Figure 7 displays the resulting write pattern lengths for a DDR4-1866 memory (the trends for the read patterns and the DDR4-2400 look the same). For access granularities where BI and BC are both larger than 1 (and could hence use bank-group interleaving), BS BI generates patterns that are on average 8% larger than the optimal length. If we consider both BS BI and BS PBGI then there are only two configurations left where neither BS BI or BS PBGI are optimal (the trends look similar for DDR4-2400).

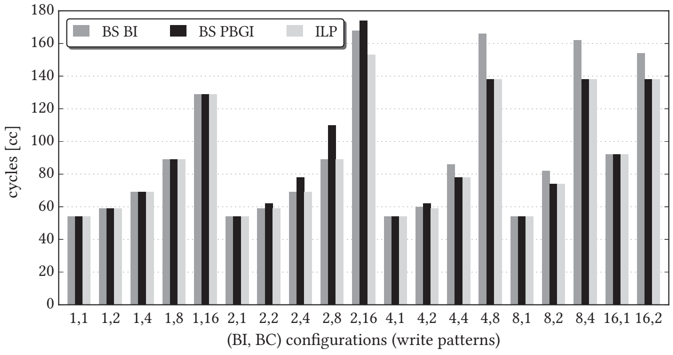


Fig. 7. Comparison of write pattern lengths for DDR4-1866, using bank scheduling (BS BI), bank scheduling with pairwise bank-group interleaving (BS PBGI) and the ILP formulation (ILP). Lower is better.

$(4,2)$ suffers from the same effect as the LPDDR3-1333 that was discussed earlier. The remaining configuration, $(2,16)$, use a complex bank interleaving order in the ILP's solution, of which the properties are dependent on the particular numerical values of the timing constraints for the memory device under consideration.

Since the run-time of both heuristics is negligible for all practical (BI, BC) combinations (it takes less than two seconds to generate the 196 pattern sets in Figure 8 on a 2.9GHz i5 processor), it is feasible to execute both heuristics for configurations where BI and BC are both larger than one, and then select the best result. Cases may exist where the read pattern is smaller in BS PBGI, while the write pattern is smaller in BS BI or vice versa. *We propose to select the pattern set that delivers most worst-case bandwidth in those cases*. Note that the read and the write pattern *have* to use the same bank interleaving order, to avoid permuting the data when sequentially reading and writing the same address.

For access granularities where BI and BC are both larger than 1, the average pattern generated by this procedure is 1.1% larger than optimal. Since the potential gains of creating a more refined heuristic that mimics the ILP solutions more closely are quite small, and because it is not straightforward to define it generically, *we propose to use a combination of* BS BI *and* BS PBGI *as a fast way to generate patterns*.

As one might expect, generating a solution through the ILP formulation is significantly more time consuming than using the heuristics; it takes about an hour (on a 3GHz i7 processor) for the biggest patterns with 32 bursts, the solving time growing roughly exponentially with the number of bursts. Using the ILP solution is therefore feasible as long as the access granularity and thus number of bursts is small enough, and the required number of iterations over different configurations and memory types is limited.

4.2 Worst-Case Performance Evaluation

The pattern generation algorithm that was introduced in Section 3.2 is used to evaluate the worst-case performance of the SDRAM devices we selected in Section 4, for different values of BI and BC. Section 4.2.1 explains the origin of the metrics displayed in the resulting plots, while Section 4.2.2 until Section 4.2.4 discuss the trade-off offered by the different configurations in detail.

4.2.1 Worst-Case Bandwidth, Energy, and Power Metrics

Each pattern configuration is identified by two numbers, BI and BC. The data point in Figure 8 are annotated with these two numbers. Note that the LPDDR memories have no BI=8 results because they only have 4 banks. Configurations are grouped by access granularity (using the marker shape), ranging from 8 to 256 bytes per pattern. Two performance metrics are shown in Figure 8:
1) *The worst-case bandwidth and associated efficiency* for a pattern configuration is determined using the analysis in [29]. The vertical axis displays this bandwidth, expressed in GB/s, with each vertical tick representing a 20% increase in efficiency, such that the graph covers a range from 0% to 100% efficiency.
2) To estimate *the power* associated with each pattern configuration we use the open-source DRAMPower tool [30], [31]. As input it takes a trace of SDRAM commands. From this trace, it determines the energy state of the memory in each cycle, based on the executed commands, and then it uses the IDD currents from the data sheet to derive the energy usage of the memory module. This model was verified using real hardware measurements and found to be more accurate than the (conventional) Micron's System Power Calculator [32]. For each pattern set, we generate a write and a read trace, by concatenating 1000 read or write patterns, respectively, interleaved with a periodic refresh pattern according to the refresh interval requirement (REFI) of the memory device. DRAMPower uses these traces to determine the average power consumed when continuously serving read or write requests, and hence this is an upper-bound for the average power. The larger of the read and write power is drawn on the horizontal axis in the graph, expressed in mW. In general, the worst case for bandwidth is not the same as for power.

Dividing the worst-case power by the bandwidth the yields a measure for the *energy efficiency* of a pattern configuration. The diagonal isolines in Figure 8 connect points with equal energy efficiencies. Labels at the top and right of the graphs are associated to the closest isoline, showing the energy cost in pJ/bit. Note that this is the only metric that can be fairly compared across all graphs, since it removes the dependence on the clock frequency and the data bus width.

There are many ways to read this graph. Ideally, a configuration should be as close to the upper-left corner as possible, i.e. have high bandwidth and energy efficiency, and low power. Within one graph, comparing configurations with the same access granularity (marker) shows the effect of trading BI for BC. In the DDR3-1066 graph for example, $(4,2)$ is objectively better than the $(8,1)$, since they are transparently interchangeable from the client's point of view, but $(4,2)$ is better in the three plotted performance metrics.

Pairs of graphs that belong to the same memory type have their data points in approximately the same relative position, but both the power axis and bandwidth axis are scaled up with frequency. Comparing DDR2-1066 with DDR3-1066 shows a significant drop in power usage on a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2458859, IEEE Transactions on Computers

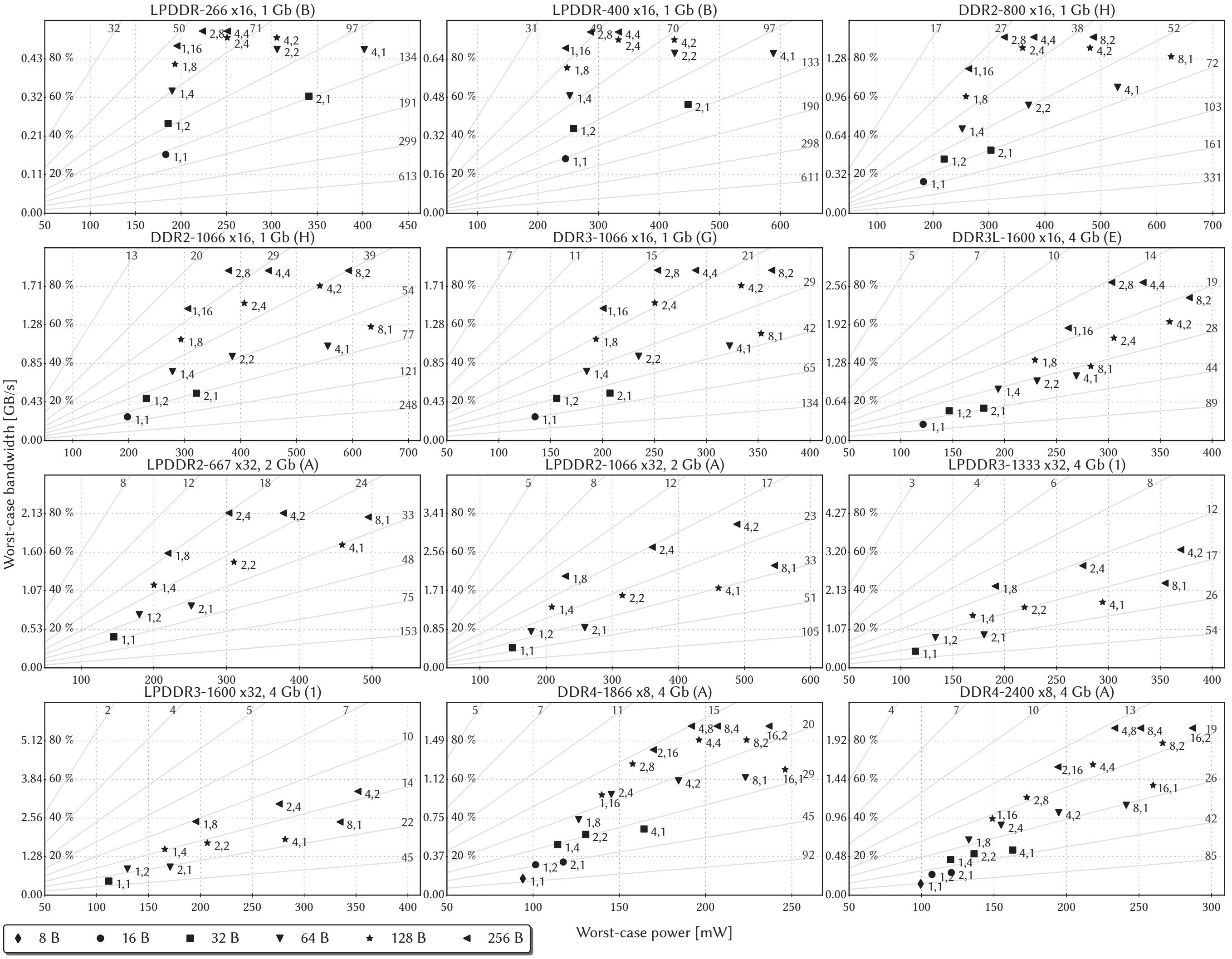


Fig. 8. Worst-case bandwidth vs. worst-case power. The diagonal isolines connect points of equal energy efficiency, labeled with [pJ/bit] (125 divided by these labels yields [GB/J]). %-labels relate to the peak bandwidth.

configuration-by-configuration basis, while the bandwidth remains almost constant, indicating that their timing constraints are very similar.

Another insightful way to look at Figure 8 is found by considering what a pattern consists of at the burst level: BI is a measure for the amount of bank-parallelism that is exploited, while BC is a measure for the page hit/miss ratio: there are $(\mathrm{BC}-1)$ hits per BC bursts even in the worst case. In terms of general memory scheduling, each configuration can thus be interpreted as an operating point of the memory as a function of the burst-level bank-parallelism and page hit/miss ratio.

4.2.2 Comparing Pattern Configurations in a Speed Bin

In this initial evaluation, we compare the relative performance of the configurations in Figure 8 on a per-memory basis. Four trends are identified:

1. *For all memory types except DDR4: configurations interleaving over more than 4 banks (BI > 4) are always worse than another configuration with a similar access granularity in terms of bandwidth and energy efficiency, and hence BI > 4 should not be used.* The inefficiency is caused by the relatively large FAW constraint having to be resolved within the pattern instead of across patterns where it overlaps with other constraints. If BI ≥ 8, then a pattern contains 8 or more ACT commands. Consequently it needs to be at least $2 \cdot$ FAW long to be valid, which is always larger than RC for all defined speed bins, and thus prohibitively expensive compared to using smaller BI with a larger BC instead.
2. For DDR4, a similar effect is visible if $BI > 8$. This can be explained using similar reasoning, considering that a pattern with 16 ACT commands is at least $4 \cdot$ FAW long, which is at least twice as large as RC for the currently defined speed bins. The FAW timing (in nanoseconds) is slowly reducing as the clock frequency increases [33], allowing more banks to be interleaved in the (relatively fast) DDR4 memories.
3. *For a constant access granularity, interleaving over more banks improves the worst-case bandwidth, but reduces the energy efficiency.* The reuse distance per bank increases as bank parallelism is exploited, improving efficiency (see Section 3.2). Energy efficiency reduces, since the relative

number of ACT and PRE commands increases, and they consume energy. This trend is overruled by trend 1 and 2.

4. *For a constant BI, increasing the access granularity (by increasing BC) improves the worst-case bandwidth and the energy efficiency*, since the cycles and energy spent on opening and closing a page is amortized over a larger number of bytes. Note that the efficiencies eventually saturate, leading to diminishing returns. The maximum access granularity is furthermore limited by the size of the requests the memory clients generate. Doubling the access granularity never doubles the efficiency, so throwing away excess data is never profitable.

A consequence of trends 3 and 4 is that *for a given access granularity, BI can be traded for BC, which corresponds to trading off worst-case bandwidth for energy efficiency*.

4.2.3 Comparing Multiple Speed Bins

By comparing configurations across speed bins, we can see that *for the same memory type and access granularity, the faster bins tend*[2] *to have a higher energy efficiency*, because the proportional growth of the worst-case bandwidth when switching to a higher speed bin is generally bigger than the proportional growth of the worst-case power within the observed frequency ranges. The $(2, 4)$ configurations for LPDDR2-667 and 1066 demonstrate this (with data points that conveniently sit on the isolines), for example, where the slower device requires 18 pJ/bit and the faster device uses 17 pJ/bit.

There are 3 reasons the worst-case bandwidth tends[2] to grow when the frequency increases. The first reason is the most obvious: at a higher frequency, each data burst requires less time, thus potentially reducing the pattern length if the data bursts are on the critical path through the pattern. The second reason is that manufacturers design the devices in the higher speed bins to run at the higher clock frequency of that bin, and as a result their timings in nanoseconds are also smaller. The third reason arises from the conservative discretization of memory timings into clock cycles. Even though the maximum error in the cycle-level approximation of the actual timing monotonically decreases with an increasing clock frequency, the actual error does not, such that a higher frequency might occasionally result in a bigger approximation error compared to a smaller frequency that just happens to fit better. The net effect rarely impacts the worst-case bandwidth negatively[2], but in those exceptional cases it makes no sense to run at these higher frequencies from a worst-case performance point of view.

The worst-case power generally increases as well when a higher clock frequency is used, but because a significant fraction of it is static and unaffected by the clock frequency, the energy efficiency generally improves.

Increasing the clock frequency has diminishing returns in terms of memory efficiency. The clock period shrinks faster than the pattern lengths, which implies a smaller fraction of the time is spent actually transporting data, assuming the number of bursts ($BI \cdot BC$) in the pattern remains constant. The fraction of time spent waiting for nanosecond-based constraints, for example related to activating and precharging, increases. *This means that the required* (BI, BC) *product to reach a certain memory efficiency grows*; this effect is visible in Figure 8, where the same configuration per memory type has a higher efficiency in the slower speed bin than in the fast speed bin.

The effects of increasing the width of the data bus on efficiency mirror those of increasing the clock frequency, since it also reduces fraction of time spent transporting data. *Increasing the data bus width thus also has diminishing returns in terms of efficiency*. If the pin and wiring costs are of key importance in a particular design, then it may make sense to prefer devices with a smaller data bus width if they can sustain the required bandwidth and are sufficiently energy efficient.

4.2.4 Worst-case Execution Time of a Request

The worst-case execution time of a request, i.e. how long it occupies the memory resource, is solely dependent on the length of the patterns (assuming the size of the request matches the access granularity of the pattern). The worst-case response time (WCRT) is the difference between the arrival of a request in the controller and the departure of the response. It includes the time a request is delayed by its own previous requests, requests from other clients, interference from refreshes, and any additional time delay between the execution commands and the return of data resulting from pipelining in the memory and data response path through the controller. Quantifying each of these components is only possible for concrete architecture instances, with a known arbiter type, a specific number of clients, and assumptions on the temporal arrival behavior of requests. A spectrum of parameterized approaches to WCRT analysis is available [25], [29], [34]. Since this section focuses on the general trends across memory configurations and types, we do not commit to a specific architecture and WCRT analysis method. Instead, we focus on the parameters produced by the pattern-generation process, i.e. the duration of the individual patterns and the worst-case bandwidth, that are generally used within a WCRT analysis, based on related work.

Figure 9 shows the execution time of the memory patterns for various memory types and configurations (ordered by access granularity and BI). The first two groups of bars represent the entire configuration space for access granularities of 32, 64 and 128 bytes for the two LPDDR2 memories. The other groups show the 64 byte configurations of the fastest memories in our set for the remaining memory types. The *offset* bar shows the time it takes from the start of a read pattern until the final data word is put on the data bus by the memory. This may happen after the end of the read pattern, because commands are pipelined in the memory, and thus the offset bar is sometimes larger than the read bar. *Offset* represents the minimum time any read request has to wait for its data. The total stacked length of the bars per configuration can be interpreted as the WCRT of a read request that has to wait for a refresh, an interfering read and write pattern and the associated switching patterns, and finally its own data offset.

[2] In the set of experiments presented in Figure 8, only the pair of DDR2 memories shows a bandwidth reduction in a higher speed bin, when $BC = 1$ and $BI \in \{1, 2, 4, 8\}$. The maximum reduction is less than 4 percent.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2458859, IEEE Transactions on Computers

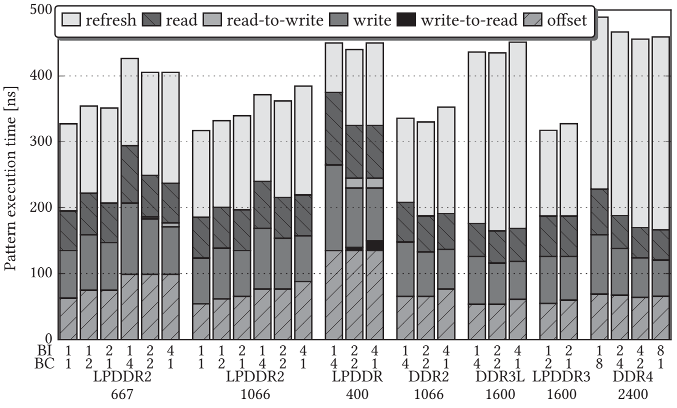


Fig. 9. Pattern execution times and data offset.

Comparing the configurations for LPDDR2-667, we can see that the pattern execution times grow as expected (Figure 2) when the access granularity grows. The length of the refresh pattern increases as patterns become more efficient. This happens because a refresh command can only be issued once *all* banks have been precharged, and NOPs are inserted at the start of the refresh pattern to assert this. Efficient pattern configurations exploit bank-parallelism and have their final read or write burst relatively close to the end of the associated pattern, thus increasing the required number of NOPs before the refresh command. The switching patterns grow with efficiency for similar reasons; they insert NOPs between the bursts close to end of a patterns and the beginning of patterns of the opposing types. Note that exchanging read and write pattern duration for longer refresh and switching patterns is not a zero-sum game, because refresh patterns have to be issued infrequently relative to read or write patterns, and the switching patterns are not always in the worst-case sequence of patterns that determines the efficiency[3].

Comparing similar configurations (same (BI, BC)) across the LPDDR2 memories, reveals that increasing the clock frequency reduces the execution time of the patterns, in line with the bandwidth trends. Switching patterns disappear, because the data-bus timings that dictate their length are specified in clock cycles and thus shrink in comparison to the analog timings that are based on nanoseconds, which dominate the read and write pattern length at higher clock frequencies.

The refresh pattern length for DDR3 and DDR4 is roughly twice as big as that of the LPDDR3 memory, but LPDDR2/3 memories need to be refreshed twice as often, and hence still exhibit approximately the same refresh overhead.

Finally, it is interesting to look at the global picture, considering the sensitivity of the worst-case execution time to the clock frequency. For example, when comparing the LPDDR2-677 and LPDDR3-1600 $(2, 2)$ configuration, the frequency grows with 140%, while the duration of a read-write-offset sequence reduces by only 49 ns or 20%. This highlights that both new memory technologies and higher clock frequencies have not given much benefit yet in terms of worst-case execution time.

5 Related Work

Many SDRAM command schedulers and/or controllers have been proposed in related work, employing a range of methods to improve different performance aspects. First, we discuss some of the works that focus more on average-case performance, and then discuss the analyzable, real-time capable techniques in detail.

Methods that improve average-case performance of SDRAM are abundant, exploiting locality [16], [35], grouping requests per thread [36], exchanging more information with the cache [37] and even using reenforcement learning to adapt the scheduling policy at run-time [38]. These techniques interact with the command scheduling in complex ways, relying on unpredictable request reordering schemes that are effectively impossible to analyze, which means no useful bounds on the real-time performance can be derived. This makes it impossible to use them in a real-time context.

Within the group of analyzable controllers, the amount of a priori information that is assumed to be available / exploitable by the command scheduler varies. [39] requires every single request to be known at design time to compute a static command schedule at design time. It is hence completely analyzable, but has very limited flexibility, since obtaining this information in a multi-core system is not possible in the general case.

The PRET controller [19] schedules commands according to a static periodic schedule. Banks are accessed one burst at a time (i.e. with a BC of 1), and consecutive bursts are guaranteed to be interleaved across banks, such that bank parallelism can be exploited. The degree of bank parallelism (their BI) is equal to the number of banks in the device. The RTCMC controller [22] dynamically schedules commands at run-time. Similarly to the PRET controller, it interleaves bursts over all 4 banks in the device, issuing one burst per bank (i.e. like a $(4, 1)$ configuration). Both PRET and RTCMC use a close-page policy, and target relatively slow DDR2 devices that have small read-write switching constraints compared to the memories considered in this article. This allows read-write interleaving at *burst* granularity without prohibitively large penalties, but does not scale well to faster memories as the read-to-write and write-to-read constraints grow and start to dominate the schedule length. The RRD constraint for these memories is also small enough, such that even with $BC = 1$ the efficiency is still reasonable, but this again changes when faster memories are considered.

The SMC controller [40] focuses on real-time steaming traffic, which allows the memory controller to deal with relatively large requests, up to the size of an entire page (1 KB for their device). This is equivalent to choosing BC such that the access granularity is 1 page, with BI 1. Their target device is a DDR1, which we did not consider in this article, but their results relating a larger access granularity to lower power usage are in line with the trends we observe.

[3] 70% of the tested configurations have only write (and refresh) patterns in its worst-case sequence, 30% alternates between read and write (and refresh) patterns.

The authors of [24] and [41] propose dynamically scheduled memory controllers with a corresponding worst-case analysis method, using an open and close-page policy, respectively. Benefiting from the use of an open-page policy in terms of lower worst-case bounds is only possible when bank privatization is used, as far as we are aware. [24] uses this approach, i.e. it assigns a private bank to each real-time client, and assumes that a static analysis is available to determine which loads and stores result in SDRAM accesses, and are hits or misses. Close-page controllers do not require such in-depth application knowledge, but are hence also not able to exploit it, which could lead to more pessimistic bounds depending on the use case. [22] analyses a similar privatization-based open-page approach, but concludes that giving a complete bank to each application (or thread) leads to scalability issues (the number of banks is limited), and therefore drops the idea in favor of a close-page $(4, 1)$ configuration. Rank interleaving is used in [24] to limit read-write switching overhead. We do not consider rank interleaving, since the switching overhead is not dominating the worst-case pattern sequence when a close-page policy is used in 70% of the configurations shown in Figure 8, and is hence not relevant for worst-case performance in those cases. In the remaining configurations, the average overhead is 5%, with a maximum of 16% for the slowest LPDDR memory. Taking into account that switching ranks has a penalty of about 2 cycles, which replaces a switching pattern with a length of the same order of magnitude, we estimate the average efficiency improvement obtained through rank interleaving is only 2 to 3%, within the 30% of the configurations that would benefit at all.

[25] specifically analyses a FR-FCFS arbiter, i.e. one that prioritizes row hits over row misses. In the worst-case, it assumes all requests of the application that is being analyzed are misses. It has consider that row-hits from other applications overtake these requests through reordering, so even though the analysis takes this open-page aspect into account, it is not beneficial for the derived worst-case bounds. It assumes each request maps to a single burst.

None of the related papers mentioned above (except [40]) take power into account, despite it being an important design constraint [42]. (BI, BC) trade-offs are not considered, and the analysis in each of the papers is limited to one or two memory generations, while we show a much broader range of memories and a suitable abstraction to deal with their relative differences. The real-time implications of the introduction of bank groups in DDR4 has also not been evaluated in related work.

A few other memory generation overview papers exists. [33] discusses some of the differences in DDR2/3/4 memory timings, but does not show the effect on worst-case performance. The authors in [43] shows the bandwidth/energy efficiency trade-offs for different memories when applied in data centers, but it considers a smaller set of SDRAM generations compared to this article, and does not focus on worst-case performance. [44] compares several (asynchronous) DRAM architectures and considers SDRAM as one special case within this family, but does not zoom in further. In [45], the focus lies on selecting a suitable memory for a design rather than giving a general performance overview, and it only considers LPDDRx memories.

This work is an extension of [46], where bandwidth/energy/execution time trade offs as a function of BI and BC were discussed for a single DDR3 device. It used techniques from [20], which proposes a memory controller that dynamically schedules precomputed sequences of SDRAM commands (memory patterns) according to a fixed set of scheduling rules. [20] did not consider power, uses a variable BC, but fixes BI to the number of banks in the memory, while we allow BI to be variable in this work.

6 Conclusions

This article showed how the memory command scheduling problem for real-time memory controllers can be generalized such that multiple memory generations can easily be supported. The proposed BS BI and BS PBGI command scheduling heuristics are configurable to interleave requests over a variable number of banks, and/or use multiple bursts per bank. BS PBGI is tailored for DDR4, and uses pairwise bank-group interleaving to reduce the schedule length. Both heuristics' outputs were compared to that of an ILP formulation of the same scheduling problem. A combination of both heuristics generates schedules within 2% of the optimal length for 12 devices from the LPDDR1/2/3, DDR2/3/4 generations, considering access granularities up to 256 bytes. We derived the worst-case bandwidth, power and execution time for the same set of devices and scheduler configurations, and evaluated the observed differences.

Acknowledgments

This work was partially funded by projects CATRENE CA505 BENEFIC, CA703 OpenES, CT217 RESIST; ARTEMIS 621429 EMC2 and 621353 DEWI, and the Ministry of Education of the Czech Republic under project number CZ.1.07/2.3.00/30.0034.

References

[1] S. McKee, "Reflections on the memory wall," in *Proc. Conf. on Computing frontiers*, 2004, pp. 162–167.

[2] B. Loop and C. Cox, "An analytical study of DRAM power consumption across memory technologies," in *Proc. Energy Aware Computing (ICEAC), Int. Conf. on*, 2011, pp. 1–3.

[3] JEDEC, "DDR4 SDRAM specification JESD79-4."

[4] B. Jacob, S. Ng, and D. Wang, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007.

[5] JEDEC, "DDR2 SDRAM specification JESD79-2F," 2009.

[6] ——, "DDR3 SDRAM specification JESD79-3E."

[7] ——, "Low power double data rate specification JESD209B."

[8] ——, "Low power double data rate 2 specification JESD209-2D."

[9] ——, "Low power double data rate 3 specification JESD209-3B."

[10] P. Kollig, C. Osborne, and T. Henriksson, "Heterogeneous multi-core platform for consumer multimedia applications," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2009, pp. 1254–1259.

[11] *RM57L843 16- and 32-Bit RISC Flash Microcontroller*, Texas Instruments Inc., 2014.

[12] P. van der Wolf and J. Geuzebroek, "SoC infrastructures for predictable system integration," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2011, pp. 1–6.

[13] M. D. Gomony, B. Akesson, and K. Goossens, "Architecture and optimal configuration of a real-time multi-channel memory controller," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2013, pp. 1307–1312.

[14] *P4080 QorIQ Multicore Communication Processor Reference Manual*, P4080RM Rev. 2 ed., Freescale.

[15] L. Steffens, M. Agarwal, and P. van der Wolf, "Real-time analysis for memory access in media processing socs: A practical approach," in *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2008, pp. 255–265.

[16] S. Rixner *et al.*, "Memory access scheduling," in *Comput. Architecture, Int. Symp. (ISCA)*, 2000, pp. 128–138.

[17] B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems*, ser. Embedded Systems Series. Springer, 2011.

[18] K. Chandrasekar, B. Akesson, and K. Goossens, "Run-time power-down strategies for real-time SDRAM memory controllers," in *Design Automation Conf. (DAC)*, 2012, pp. 988 –993.

[19] J. Reineke *et al.*, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Proc. CODES+ISSS*, 2011, pp. 99–108.

[20] B. Akesson and K. Goossens, "Architectures and modeling of predictable memory controllers for improved system integration," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2011, pp. 1–6.

[21] H. Shah, A. Raabe, and A. Knoll, "Bounding WCET of applications using SDRAM with priority based budget scheduling in MPSOCs," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2012, pp. 665–670.

[22] M. Paolieri, E. Quiñones, and F. J. Cazorla, "Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 1s, p. 64, 2013.

[23] *DDR3L SDRAM*, 4Gb_DDR3L.pdf - Rev. I 9/13 EN ed., Micron.

[24] Y. Krishnapillai, Z. Pei Wu, and R. Pellizzoni, "ROC: A rank-switching, open-row DRAM controller for time-predictable systems," in *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2014, pp. 27–38.

[25] H. Kim *et al.*, "Bounding memory interference delay in COTS-based multi-core systems," in *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, 2014, pp. 145–154.

[26] B. Akesson, W. Hayes, and K. Goossens, "Automatic generation of efficient predictable memory patterns," in *Embedded and Real-Time Computing Syst. and Applicat. (RTCSA)*, 2011, pp. 177–184.

[27] S. Goossens, "Power/performance trade-offs in real-time SDRAM controllers - code and datasets," http://www.es.ele.tue.nl/~sgoossens/sdram_trade_offs.

[28] K. Chandrasekar *et al.*, "Towards variation-aware system-level power estimation of DRAMs: an empirical approach," in *Design Automation Conf. (DAC)*, 2013, pp. 23:1–23:8.

[29] B. Akesson, W. Hayes Jr., and K. Goossens, "Classification and analysis of predictable memory patterns," in *Embedded and Real-Time Computing Syst. and Applicat. (RTCSA)*, 2010, pp. 367–376.

[30] K. Chandrasekar *et al.*, "Drampower: Open-source DRAM power & energy estimation tool," http://www.drampower.info.

[31] K. Chandrasekar, B. Akesson, and K. Goossens, "Improved power modeling of DDR SDRAMs," in *Digital System Design (DSD)*, 2011, pp. 99–108.

[32] K. Chandrasekar, "High-level power estimation and optimization of DRAMs," Ph.D. dissertation, Delft University of Technology, 2014.

[33] "DDR4 networking design guide introduction," Micron Technology Inc., Tech. Rep., 2014, tN-40-03.

[34] H. Shah, A. Knoll, and B. Akesson, "Bounding SDRAM interference: Detailed analysis vs. latency-rate analysis," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2013, pp. 1–6.

[35] J. Dodd, "Adaptive page management," 2006, US Patent 7,076,617.

[36] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, 2008.

[37] J. Stuecheli *et al.*, "The virtual write queue: coordinating DRAM and last-level cache policies," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 72–82, 2010.

[38] E. Ipek *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," in *Comput. Architecture, Int. Symp. (ISCA)*, 2008, pp. 39–50.

[39] S. Bayliss and G. Constantinides, "Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search," in *Field-Programmable Technology, Int. Conf. on*, 2009, pp. 304–307.

[40] A. Burchardt, E. Hekstra-Nowacka, and A. Chauhan, "A real-time streaming memory controller," in *Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, 2005, pp. 20–25.

[41] Y. Li, B. Akesson, and K. Goossens, "Dynamic command scheduling for real-time memory controllers," in *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2014, pp. 3–14.

[42] "International technology roadmap for semiconductors (ITRS) - system drivers," 2011, http://www.itrs.net/reports.html.

[43] K. Malladi *et al.*, "Towards energy-proportional datacenter memory with mobile DRAM," in *Comput. Architecture, Int. Symp. (ISCA)*, 2012, pp. 37–48.

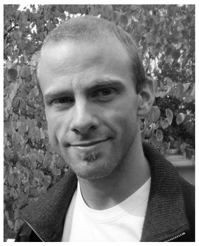
[44] V. Cuppu *et al.*, "A performance comparison of contemporary DRAM architectures," *SIGARCH Comput. Archit. News*, vol. 27, no. 2, pp. 222–233, 1999.

[45] M. Gomony *et al.*, "Dram selection and configuration for real-time mobile systems," in *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2012, pp. 51 –56.

[46] S. Goossens *et al.*, "Memory-map selection for firm real-time SDRAM controllers," in *Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2012, pp. 828 –831.

Sven Goossens Sven Goossens received a MSc in embedded systems from the Eindhoven University of Technology in 2010, and is currently a PhD candidate at the same university. His research interests include mixed time-criticality systems, composability and SDRAM controllers.

Karthik Chandrasekar Karthik Chandrasekar earned his M.Sc. degree in Computer Engineering from TU Delft in the Netherlands in November 2009. In October 2014, he received his PhD also from the same university. His research interests include Real-Time Embedded Systems, Power and Performance Modeling and Optimization, DRAM Memories and Memory Controllers. He is currently employed as a Senior Architect at Nvidia.



Benny Akesson Benny Akesson earned a M.Sc. degree in Computer Science and Engineering at Lund Institute of Technology, Sweden in 2005. In 2010, Dr. Akesson received his Ph.D. degree in Electrical Engineering at Eindhoven University of Technology, the Netherlands. He is currently employed as a Postdoctoral Researcher at the Czech Technical University in Prague. His main research interest is memory controllers for real-time systems.

Kees Goossens Kees Goossens received his PhD in Computer Science from the University of Edinburgh in 1993. He worked for Philips/NXP Research from 1995 to 2010 on networks on chip for consumer electronics, where real-time performance, predictability, and costs are major constraints. He was part-time professor at Delft university from 2007 to 2010, and is now professor at the Eindhoven university of technology, where his research focuses on composable (virtualised), predictable (real-time), low-power embedded systems. He published 3 books, 100+ papers, and 24 patents.