Automatic Generation of Efficient Predictable Memory Patterns

Benny Akesson, Williston Hayes Jr., Kees Goossens
Eindhoven University of Technology

***Abstract*—Verifying firm real-time requirements gets increasingly complex, as the number of applications in embedded systems grows. Predictable systems reduce the complexity by enabling formal verification. However, these systems require predictable software and hardware components, which is problematic for resources with highly variable execution times, such as SDRAM controllers. A predictable SDRAM controller has been proposed that addresses this problem using predictable memory patterns, which are precomputed sequences of SDRAM commands. However, the memory patterns are derived manually, which is a time-consuming and error-prone process that must be repeated for every memory device, and may result in inefficient use of scarce and expensive bandwidth.**

This paper addresses this issue by proposing three algorithms for automatic generation of efficient memory patterns that provide different trade-offs between run-time of the algorithm and the bandwidth guaranteed by the controller. We experimentally evaluate the algorithms for a number of DDR2/DDR3 memories and show that an appropriate choice of algorithm reduces run-time to less than a second and increases the guaranteed bandwidth by up to 10.2%.

***Index Terms*—predictability; real-time; SDRAM; memory controller; memory patterns; pattern generation; memory efficiency**

I. Introduction

Embedded systems get increasingly complex, both in terms of software and hardware. The number of applications is growing as well as their individual complexity, requiring increased system performance. To deliver on this expectation while limiting power consumption, industry is moving towards sophisticated heterogeneous multi-processor platforms that enable concurrent execution of applications [1], [2]. To reduce cost, resources, such as interconnects and memories, are shared between applications, further adding to complexity by making the timing behaviors of applications inter-dependent.

The increasing complexity is challenging in real-time systems, where some applications, such as a Software-Defined Radio [3], have *firm real-time requirements*. Failure to satisfy this type of requirement is highly undesirable and may result in significantly reduced quality of the application, failure to comply with a given standard, or even violate the functional correctness of the system [4]. It is hence imperative to guarantee that all firm real-time requirements are satisfied for all inputs and all possible combinations of concurrently executing applications. This results in a problem for many simulation-based verification approaches that are too slow to achieve full coverage of the possible cases.

Techniques for design-time verification of applications with firm real-time requirements based on formal models of computation, such as data-flow analysis [5], [6], have been proposed. These approaches enable *independent application verification*, which significantly reduces the verification complexity. However, this requires *predictable systems*, where the response times of both the applications and the platform resources are bounded [7]. SDRAM memories are essential resources that are challenging to use in predictable systems. The reason is that they have highly variable response times that depend on previous requests. SDRAM bandwidth is furthermore scarce and expensive due to pin constraints on the chip, and has to be efficiently utilized. Most memory controllers hence apply sophisticated techniques to maximize the average bandwidth. However, this makes them unable to guarantee bandwidth and response time to a memory *requestor*, which is a processor that accesses the memory on behalf of an application.

A predictable memory controller that provides bounds on bandwidth and response times has been proposed that enables formal verification of real-time multi-processor systems with shared SDRAM memories [8]. This is achieved by first using a predictable arbiter to schedule memory requests that are then dynamically mapped by an SDRAM back-end to a set of *predictable memory patterns*, which are statically computed sequences of SDRAM commands with known execution times. However, a drawback of this approach is that memory patterns are derived manually, resulting in three problems: 1) Making a pattern set is a time-consuming process that must be repeated for every type of SDRAM memory and pattern configuration. 2) Making patterns manually is error-prone, considering that a large number of timing constraints between successive commands must be satisfied for the SDRAM to execute correctly. 3) It is difficult to ensure that the generated patterns provide optimal bandwidth, due to the large size of the design space.

This paper addresses this issue by presenting *three algorithms* for generation of predictable memory patterns offering different trade-offs between run-time and guaranteed bandwidth. The first algorithm uses a branch and bound search that is guaranteed to find the optimal patterns subject to our design decisions. The second algorithm is a heuristic that works cycle-by-cycle and schedules SDRAM commands as-soon-as-possible (ASAP) when their timing constraints are satisfied. The last algorithm is a heuristic that schedules commands bank-by-bank. We experimentally compare the algorithms for a range of DDR2/DDR3 SDRAM memories with a variety of pattern configurations and show that an appropriate choice of algorithm reduces run-time to less than a second and increases the guaranteed bandwidth by up to 10.2%.

The remainder of this paper is organized as follows. Section II reviews related work. The SDRAM architecture is introduced in Section III, and we explain the problem of providing guarantees on bandwidth and response times with these memories. Section IV then recapitulates the concept of memory patterns and explains how they address the problem. The main contributions of this paper are in Section V, which presents three algorithms for automatic generation of efficient memory patterns. Section VI experimentally compares the different algorithms in terms of run-time and provided bandwidth, before we present our conclusions in Section VII.

II. Related work

Most SDRAM controllers are either statically or dynamically scheduled, depending on the type of systems they target. Statically scheduled controllers execute precomputed schedules of SDRAM commands that have been determined at design time. These controllers are predictable, since the response time and bandwidth provided to a requestor can be bounded at design time by analyzing the schedule. However, the predictability of these controllers comes at the expense of flexibility, as it requires the entire trace of memory requests to be known at design time. Statically scheduled controllers are hence suitable for real-time systems executing single applications that are not input-dependent [9], but do not scale to systems with multiple concurrently executing applications.

Dynamically scheduled memory controllers, on the other hand, schedule SDRAM commands at run-time based on available requests. These controllers target high efficiency and flexibility to fit in high-performance systems with dynamic applications. Several of these controllers feature sophisticated mechanisms to reduce average response times or increase the average available bandwidth. Examples involve preference for requests that target open rows in the memory banks [10]–[13], or that fit with the current direction (read/write) of the data bus [12]–[15]. The problem with these controllers is that the interactions between all these mechanisms are complex, making it difficult to derive useful bounds on bandwidth and response times. Most dynamically scheduled controllers are hence unsuitable for systems with firm real-time requirements.

A hybrid memory controller that combines aspects of both statically and dynamically scheduled approaches is proposed in [8]. This controller uses five types of *predictable memory patterns*, which are statically computed sub-schedules of SDRAM commands that are dynamically combined at run-time. This results in a predictable controller that is more flexible than statically scheduled designs. However, the memory patterns must be manually derived for every type of SDRAM device, which is time-consuming and error-prone, and may result in inefficient use of scarce bandwidth. This paper addresses this problem by proposing three algorithms for predictable memory pattern generation. The algorithms are experimentally evaluated to determine the trade-off between guaranteed bandwidth and run-time of the algorithms.

III. SDRAM overview

An SDRAM memory comprises a number of banks (n_{banks}), each containing a memory array with a matrix-like structure, consisting of rows and columns. Each bank has a row buffer that can hold one open row at a time, and read and write operations are only allowed to the open row.

A bank has two states, idle and active. The bank is activated from the idle state by an *activate* (ACT) command that loads the requested row into a row buffer, which stores the most recently activated row. Once the bank has been activated, *read* (RD) and *write* (WR) bursts can be issued to access the columns in the row buffer. These bursts have a programmable *burst length* (BL) of either 4 or 8 words for DDR2/DDR3 SDRAM that are transferred from/to the memory with a data rate of two words per clock cycle. Finally, a *precharge* (PRE) command is issued to return the bank to the idle state. This stores the row in the buffer back into the memory array. Read and write commands can be issued with an *auto-precharge* flag, resulting in an automatic precharge at the earliest possible moment after the data transfer is completed. In order to retain data, all rows in the SDRAM have to be refreshed regularly, which is done by precharging all banks and issuing a *refresh* (REF) command. If no other command is required during a clock cycle, a *no-operation* (NOP) command is issued.

The SDRAM architecture makes the response time of requests and the provided bandwidth highly variable for three reasons. 1) A request targeting an open row can be served immediately, while it otherwise first needs the current row to be closed and the required row to be opened. 2) The data bus is bi-directional and requires several cycles to switch from read to write and vice versa. 3) The memory must occasionally be refreshed before executing the next request, resulting in several additional cycles without data transfer. The impact of these factors may cause the time to serve an SDRAM burst to vary by an order of magnitude from a few clock cycles to a few tens of cycles. This makes it very challenging to tightly bound response times and the provided bandwidth.

The bandwidth to and from a memory ideally corresponds to the product of the width of the memory interface, the clock frequency of the memory, and the data rate. This is referred to as the *peak bandwidth* of the memory. However, for the previously mentioned reasons, the peak bandwidth of SDRAMs cannot be fully utilized. This is captured by the concept of *memory efficiency*, which is the fraction of clock cycles with useful data on the data bus [16]. The product of the memory efficiency and the peak bandwidth determines the *net bandwidth*, which is the bandwidth that is useful to the requestors after considering all types of overhead.

IV. Memory patterns

Providing useful bounds on response time and net bandwidth to SDRAM requestors at design time is a challenging task. This section first provides an overview of how the hybrid memory controller proposed in [8] accomplishes this using predictable memory patterns and then explains the general structure of each of the pattern types, which is a prerequisite for the memory pattern generation presented in Section V.

A. Memory pattern overview

Scheduling SDRAM commands is not trivial, since there is a considerable number of timing constraints that must be satisfied before a command can be issued. These timing constraints are specified minimum delays between issuing particular SDRAM commands, such as two activates to the same (tRC) or different ($tRRD$) banks, an activate and a read or a write in the same bank ($tRCD$), two reads or two writes ($BL/2$), a read and a precharge ($tRTP$), or a precharge and an activate to the same bank (tRP). Some of these constraints are illustrated in Figure 1. The complete list of timing constraints and their particular delays are specified for all DDR2/DDR3 memories in [17], [18].

Command scheduling in the hybrid controller is based on predictable memory patterns, which are precomputed sub-schedules of SDRAM commands that are designed to satisfy the timing constraints of the memory. These patterns are

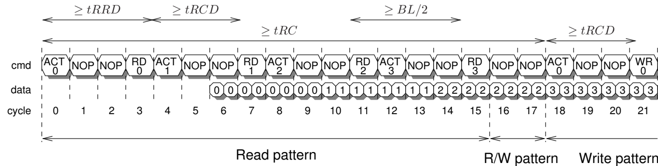


Fig. 1. Read pattern with $BL = 8$ followed by a read/write switching pattern and a partial write pattern (DDR2-400).

dynamically scheduled at run-time, depending on incoming requests, thus increasing flexibility over statically scheduled controllers. A *memory pattern set* consists of five types of patterns: 1) a read pattern, 2) a write pattern, 3) a read/write switching pattern, 4) a write/read switching pattern, and 5) a refresh pattern. The read and the write pattern are referred to as *access patterns*, while the remaining patterns are called *auxiliary patterns*. The patterns are created such that multiple read or write patterns can be scheduled in sequence. However, a switching pattern is required between a read and a write pattern, and vice versa. The refresh pattern is scheduled periodically and can be followed by either a read or a write pattern without a preceding switching pattern. The mapping from requests to patterns is illustrated in Figure 2. The actual command sequence of a read pattern followed by a read/write switching pattern and a write pattern (first four cycles) for a DDR2-400 memory is shown in Figure 1. Note that the data from a bank arrives on the data bus a few cycles after the corresponding read command, due to the read latency in the memory device. The data from the last banks may hence arrive during the following patterns.

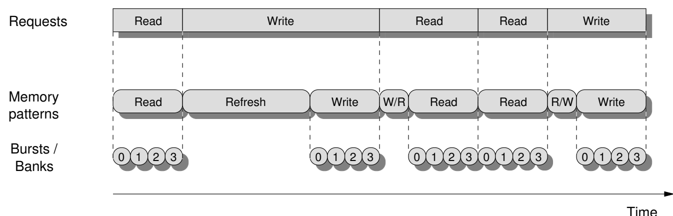


Fig. 2. Mapping from requests to patterns to SDRAM bursts.

To bound the bandwidth and response times provided by the controller, information is required about the temporal behavior of the memory patterns, defined in Definition 1. The definition considers the lengths of the patterns in the set, corresponding to the number of SDRAM commands in each pattern. One command is issued by the memory controller per clock cycle, which implies that the time to issue a pattern is known at design time.

Definition 1 (Memory pattern set): A memory pattern set is defined as (t_{read}, t_{write}, t_{rtw}, t_{wtr}, t_{ref}), where the parameters correspond to the lengths of the read pattern, the write pattern, the read/write switching pattern, the write/read switching pattern, and the refresh pattern, respectively.

The use of predictable memory patterns combined with a predictable arbiter enables the hybrid memory controller to provide guarantees on net bandwidth and response times to its requestors. Response time is bounded according to a three step process: 1) The predictable arbiter bounds the maximum number of interfering requests before a particular requestor is scheduled. 2) For a given set of memory patterns, it is shown in [16] how to determine the worst-case combination of patterns these interfering requests may map to. 3) It is known how many cycles it takes to execute this worst-case combination of patterns, since they are generated at design time. Net bandwidth is bounded in a similar manner, since the time to execute the worst-case combination of patterns is known together with how much data they transfer to and from the memory.

B. Pattern structure

We now present the general structure of the different patterns in a pattern set. There are many possible patterns for each memory device that implement this structure. For now, we keep the discussion general and consider any patterns of the different types that satisfy the scheduling rules and do not violate the timing constraints of the memory device. We refer to these patterns as *valid patterns*. We return in Section V to discuss how to construct valid patterns that are efficient.

The controller uses an interleaving memory map. This means that read and write accesses to successive logical addresses map to SDRAM bursts for the different banks in sequence. An access pattern consists of a read or a write burst to each of the banks in turn, as illustrated in Figure 2. Interleaving over the banks in this manner is an efficient way to access the memory, since it is possible to activate and precharge one bank while reading or writing to another. To be able to schedule access patterns of the same type immediately after each other, as shown in the figure, they must be completely independent of each other. It is hence not possible to assume that the correct rows are open in any of the banks. An access pattern must hence contain an activate and a precharge command for each bank to return them to a neutral state, thus implementing a close-page policy. Access patterns also contain a fixed number of SDRAM bursts to every bank. The number of issued SDRAM bursts per bank is a pattern parameter, referred to as the *burst count* (BC), which enables a trade-off between the guaranteed net bandwidth and response times provided by the patterns [16]. The example access patterns in Figure 2 have a burst count of one, since there is only a single SDRAM burst per bank in the patterns.

The switching patterns are used to provide sufficient time for the SDRAM to reverse the direction of the data bus. These patterns only consist of NOP commands, and the length is determined by the minimum number of cycles required between read and write commands, which are defined by the specification of the memory device. Note that it is possible to have switching patterns with a length of zero cycles if the distance between the last read command in a read pattern and the first write command in a write pattern, or the other way around, is already sufficient.

The refresh pattern contains a single refresh command, preceded and succeeded by a number of NOPs. There have to be enough NOPs before the refresh command to allow all banks to precharge after the last read or write pattern. After the refresh command is issued, there have to be at least $tRFC$ NOPs to allow the refresh operation to complete before the next pattern is issued.

V. MEMORY PATTERN GENERATION

After explaining the idea behind memory patterns and their overall structure, this section explains how to generate efficient patterns that implement this structure. First, we discuss some design decisions that significantly reduce the search space, while having negligible impact on the efficiency of the generated patterns. We then proceed by explaining the conditions that have to be satisfied for an access pattern to be considered valid and complete. We then move on to present three access pattern generation algorithms with different trade-off between the efficiency of the generated pattern sets and run-time. In this work, we focus our efforts on generating patterns for a given burst count. How to determine a suitable burst count for a set of requestor requirements is discussed in [19]. Lastly, we conclude by discussing how to generate auxiliary patterns for the access patterns provided by the algorithms.

A. Design decisions

The number of valid access patterns for a given burst count grows exponentially with the pattern length. To manage the size of this design space, we make five important design decisions. The first design decision is that *we assume that shorter access patterns result in higher bandwidth and lower response times than longer ones*. The benefit of this assumption is that it allows the pattern generation algorithms to focus on independently finding the shortest read and write patterns for the given burst count before deriving the corresponding auxiliary patterns. The assumption holds for most pattern sets, but sometimes longer access patterns result in shorter auxiliary patterns that together cause a marginal increase in memory efficiency and response times [19]. However, the maximum impact of this is estimated to be less than 1% reduction in memory efficiency and a few clock cycles of latency under any circumstances.

The second design decision is *not to distinguish the identity of the banks*. This means that we do not consider two access patterns as different if all commands to two banks are swapped, a decision that affects neither bandwidth nor response times. However, this decision has a significant impact on the set of valid patterns, since we do not have to consider identical patterns that access the banks in different orders.

The third design decision states that *we always start an access pattern with an activate command* and hence ignore all patterns starting with one or more NOP commands. The idea behind this decision is to prune a large number of inefficient patterns from the design space. The rationale is that the purpose of an access pattern is to issue a number of read and write bursts to the SDRAM. These bursts cannot be issued until their corresponding banks have been activated. Inserting NOPs in the beginning of an access pattern makes the access pattern longer, typically reducing bandwidth and increasing response times.

The fourth design decision is to *issue the last burst to a bank in an access pattern with the auto-precharge flag*. This reduces the number of non-NOP commands in the access patterns, limiting the design space. The use of auto-precharge may furthermore reduce the length of the pattern, since it reduces contention on the command bus of the memory.

The last design decision is to *issue all BC bursts to one bank before proceeding to the next*. A bank is ready to receive the next read or write command $BL/2$ cycles after the first. No read or write command can be issued to any other bank before this time, since it would cause a conflict on the data bus. Keeping all bursts to a bank close together may give a bank more time between the activate command and the first read or write command, as well as more time to precharge after the last read or write command before the following activate command. This makes it easier to satisfy the timing constraints of the memory device, potentially resulting in shorter patterns.

B. Access pattern termination

We now show how to decide when an access pattern is valid and complete, which determines what the generation algorithms actually have to do. An access pattern is valid and complete when it satisfies five conditions. We proceed by explaining these conditions and how to satisfy them.

The first termination condition requires all necessary commands to be included in the pattern. An access pattern consists of one activate command and BC read or write commands per bank. There are no precharge commands, since the last SDRAM burst in the pattern is issued with the auto-precharge flag according to the fourth design decision. After all commands have been scheduled, NOPs are added to the end of the generated pattern to prevent timing constraints from carrying over into a repeated pattern, violating their independence.

The second condition is that the activate-to-activate constraint must be satisfied for all banks. This condition implies that there must be at least tRC clock cycles between successive activates to a bank when an access pattern is repeated after itself. Since there is only one activate command per bank in an access pattern, this constraint is automatically satisfied if the length of the pattern is greater than or equal to tRC.

The third condition is that any window of $tFAW$ cycles can maximally contain four activate commands. This constraint has to be considered during the pattern generation, but NOPs may additionally have to be added at the end of pattern to allow it to be repeated after itself without violating this constraint.

The fourth termination condition requires that the data produced on the data bus by the last burst in an access pattern does not collide with the data from the first burst in the next. This requirement is satisfied if the corresponding access commands are separated by at least $BL/2$ clock cycles, which is the time required to finish the burst.

The last condition requires that there must be at least tRP clock cycles between a bank is precharged and reactivated. To satisfy this requirement, we must know in which clock cycles the precharges of the banks actually happen. This procedure works differently for read and write patterns. For a read pattern, the precharge cycle a bank is determined by finding the cycle with its activate command, t_{act}, and the cycle with its last read command, t_{read}^{last}. The precharge cycle is then computed according to Equation (1). Note that the precharge cycle is computed with respect to the start of the read pattern and may be greater than the total length of the pattern, indicating that the precharge finishes during the execution of a later pattern. The procedure is similar for write patterns, although Equation (2) is used instead. Both these equations are derived from [17], [18].

$$t^{pre}_{read} = \begin{cases} \max(t^{last}_{read} + \frac{BL}{2} + \max(tRTP, 2) - 2, \\ \quad t_{act} + tRAS) & \text{DDR2} \\ \max(t^{last}_{read} + tRTP, t_{act} + tRAS) & \text{DDR3} \end{cases} \quad (1)$$

$$t^{pre}_{write} = t^{last}_{write} + tWL + \frac{BL}{2} + tWR \quad (2)$$

C. Branch and bound scheduling

After specifying the task of access pattern generation, we present three algorithms for efficient access pattern generation. The first of the three access pattern generation algorithms is a branch and bound (B&B) algorithm. This algorithm is based on a depth-first traversal of the set of valid patterns satisfying the design decisions in Section V-A. It is guaranteed to find the shortest possible access patterns, as its bounding conditions exclude only longer patterns. We start by giving a brief introduction to the branching part of the algorithm, before explaining how it bounds the search space.

The algorithm starts an access pattern with an activate command in the first cycle, according to our third design decision. It then looks at the commands that can be scheduled the following cycle. For each command that respects the timing constraints of the memory, a copy of the pattern is made and each command is appended to the end of a copy. The algorithm repeats this process cycle by cycle until the first pattern is complete. At this point, it stores the completed pattern and returns to one of the remaining copies and continues its search until there are no unfinished copies remaining.

The set of valid patterns complying with our design decisions is very large and grows exponentially with the length of the patterns. To speed up execution of the algorithm, we implemented two bounding conditions that limit the size of the design space. The first bounding condition is a sliding cut-off point based on the pattern length. We keep track of the length of the shortest pattern found so far, and stop pursuing any branches longer than this value. This condition significantly reduces the run-time and memory use of the algorithm, while trivially not excluding the shortest pattern. The second bounding condition is an extension of the first. Whenever, the algorithm branches, it looks at the list of commands remaining to be scheduled, and performs a quick sanity check to see if the finished pattern can be shorter than or equal to the current shortest pattern in a best-case scenario [19]. If this check fails, then no further branches along this path are pursued. Just like the first condition, this significantly reduces run-time and memory usage of the algorithm without excluding the shortest pattern from the search space.

After the search is complete, there is at least one access pattern of each type with the shortest length. Out of these, we choose the read and write pattern where the last read or write command is issued as early as possible. This allows the access pattern to hide more of the precharge time, potentially resulting in shorter refresh pattern and switching patterns.

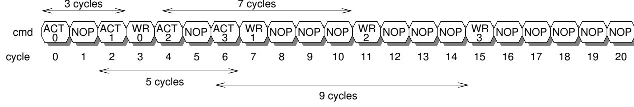
The benefit of the B&B algorithm is that it is guaranteed to find the shortest possible access patterns and choose the one that provides the shortest auxiliary patterns. The drawback of the algorithm is that it may take a long time to search the design space, despite the help of our two bounding conditions. This problem begins to show itself as clock frequencies of memory devices increase. This is because the timing constraints of a memory become longer, measured in clock cycles, resulting in longer patterns [16]. Similarly, increased burst count increases the number of commands to schedule, creating more options and longer patterns. For practical purposes, this algorithm is suitable up to DDR3-1600 with $BC = 2$. After this point, the run-time of the algorithm moves into months and years. This motivated us to look for a faster algorithm.

D. As-soon-as-possible scheduling

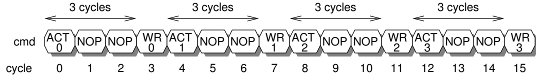
The second algorithm is a heuristic that attempts to improve run-time over the previous algorithm. The idea behind the algorithm is to schedule memory commands as-soon-as-possible (ASAP), since this intuitively leads to the shortest access patterns. According to our design decision, the algorithm starts by putting an activate command in the first cycle. It then proceeds one cycle at a time by choosing a command that can be scheduled without violating the timing constraints of the memory. If there are multiple candidate commands, a simple priority scheme is used to make the choice. This contrasts to the previous algorithm that pursues all possible options. This priority scheme first considers read and write commands, since these are the commands that put data on the data bus, thereby increasing efficiency. Activate commands are considered as second, since these enable future read or write commands, and hence future data transfer. If none of these commands are available, a NOP command is scheduled.

A consequence of the ASAP scheduling algorithm is that the activate commands are scheduled early in the pattern, as seen in Figure 3a. The reason is that activates to different banks can be scheduled every $tRRD$ clock cycles, which is not a very long time (two cycles for the DDR2-400 in the figure). However, the read and write commands must be separated by at least $BL/2$ clock cycles, causing the distance between an activate command and its corresponding read or write command to increase, as shown in the figure. This creates a problem, since a bank needs time to precharge after the last read or write command has completed, before it can be reactivated. The earliest reactivation occurs when the pattern is repeated after itself. The critical constraint is hence the time between the last read or write command in the pattern until the activate command in the repeating pattern. The earlier the activate command, the less time available to precharge. This is why the pattern generated by the ASAP scheduling algorithm requires five extra NOP commands to be inserted at the end of the pattern, while the more balanced pattern, shown in Figure 3b, does not. Clearly, scheduling commands as early as possible is not always beneficial.

The advantage of the ASAP scheduling algorithm is that it runs extremely fast. It generates a schedule in less than a second for any memory and reasonable burst count, clearly addressing the problem with the B&B algorithm. However, the advantage in speed comes at the cost of bandwidth, mainly due to the problem with prematurely scheduled activate commands. Although the ASAP scheduling algorithm provides a different trade-off between run-time and bandwidth, we consider it rather inefficient, since SDRAM bandwidth is a scarce and



(a) The ASAP algorithm results in increasingly large distances between activate commands and their corresponding write commands.



(b) A pattern with balanced distances between activate commands and write commands.

Fig. 3. Premature activate commands result in longer access patterns.

expensive resource. We hence look into a third algorithm, hoping to find a suitable middle ground.

E. Bank scheduling

The bank scheduling approach is a heuristic that builds on the lessons learned from ASAP scheduling algorithm. The idea behind the algorithm is to keep an activate command as close as possible to its corresponding read or write command, thereby preventing the precharge-to-activate constraint from extending the length of the pattern.

The bank scheduling algorithm works by scheduling one bank at a time, as opposed to working cycle-by-cycle. It starts by putting an activate command to the first bank in the first cycle, and a corresponding read or write command at the earliest possible convenience, being $tRCD$ cycles later. Each additional burst to the bank is then scheduled $BL/2$ cycles apart to constantly keep data on the data bus. This finishes the scheduling of the first bank. For each successive bank, the algorithm finds the position of the latest read or write command, and tries to schedule the next read or write $BL/2$ cycles later when the data bus is free. The new read or write command can be scheduled in this position if its activate command can be scheduled $tRCD$ cycles earlier. This depends on whether the cycle already has a scheduled command, and whether the four-activate window (FAW) constraint is satisfied. If the activate cannot be scheduled in the requested cycle, the algorithm tries to schedule the read or write command in a later cycle by iteratively repeating this test. Once the first read or write command to the bank has been scheduled, the others follow with a separation of $BL/2$ clock cycles. An illustration of the algorithm is provided in Figure 4.

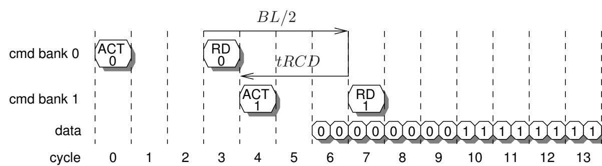


Fig. 4. Conceptual illustration of the bank scheduling algorithm for $BC = 1$.

The patterns generated by the bank scheduling algorithms achieve very regular distances between the activates and their corresponding read and write commands, addressing the problem found with the ASAP scheduling approach. In fact, the write pattern shown in Figure 3b was generated using this approach. The run-time of the algorithm is similar to the ASAP scheduling algorithm, and hence sufficiently fast. It furthermore generates pattern sets that provide equal bandwidths to those created by the B&B algorithm. Bank scheduling hence provides a very favorable trade-off between run-time and memory efficiency, compared to the other algorithms.

F. Generating auxiliary patterns

The auxiliary patterns can be generated as soon as the access patterns are given by any of the access pattern generation algorithms. We start by showing how to generate the refresh pattern, followed by the switching patterns. The refresh pattern starts with a number of NOPs that allow the banks to finish precharging after the latest access pattern. The time required to finish precharging all banks depends on the distance between the precharge cycle of the last bank, t_{read}^{pre} or t_{write}^{pre} from Equations (1) and (2), and the end of the read or write pattern, since this determines how much of the precharging time that is hidden by the access pattern itself. The number of NOPs required to precharge all banks may be different after a read and a write pattern, since the values of t_{read}^{pre} and t_{write}^{pre} are unrelated. It is hence possible to derive two refresh patterns, one that follows read patterns, and one that follows write patterns. However, reducing the refresh pattern for one of these cases with a few clock cycles has very little impact on both bandwidth and response times and is hence not considered by the hybrid memory controller. The refresh command is hence placed in cycle $tRP + (t_{read}^{pre} - t_{read})$, or $tRP + (t_{write}^{pre} - t_{write})$, whichever is larger. This is followed by a refresh command and $tRFC$ NOPs that are required to satisfy the refresh-to-activate constraint before the next pattern is issued. The equation for computing the length of refresh patterns is therefore:

$$t_{ref} = tRP + tRFC + \max(t_{read}^{pre} - t_{read}, t_{write}^{pre} - t_{write}) \quad (3)$$

The switching patterns only consist of NOP commands that allow the direction of the data bus to be reversed. We first explain how to compute the read/write switching pattern and then proceed with the write/read switching pattern. The number of NOPs in the read/write switching pattern depends both on the SDRAM generation and the burst length. Equation (4) shows the minimum number of clock cycles between a read and a write command for different memories and burst lengths. This equation is derived from the memory specifications [17], [18]. We compute the number of NOPs in the read/write switching pattern by subtracting the number of cycles between the read and write commands that are already built into the read and the write patterns. The length of the read/write switching pattern is hence computed according to Equation (5). The computation of the write/read switching pattern is computed in a similar manner. The minimum delay between the write and the read command is shown in Equation (6) and the length of the pattern is determined in Equation (7).

$$\delta_{read} = \begin{cases} 4 & \text{DDR2 with } BL = 4 \\ 6 & \text{DDR2 with } BL = 8 \\ tCL + \frac{tCCD}{2} + 2 - tWL & \text{DDR3 with } BL = 4 \\ tCL + tCCD + 2 - tWL & \text{DDR3 with } BL = 8 \end{cases} \quad (4)$$

$$t_{rtw} = \max(\delta_{read} - (t_{write}^{first} + t_{read} - t_{read}^{last}), 0) \tag{5}$$

$$\delta_{write} = tWL + \frac{BL}{2} + tWTR \tag{6}$$

$$t_{wtr} = \max(\delta_{write} - (t_{read}^{first} + t_{write} - t_{write}^{last}), 0) \tag{7}$$

VI. Experimental results

This section experimentally evaluates the worst-case memory efficiency and the run-times of the different algorithms for three different memories with a variety of burst counts. We start by explaining the setup used in the experiment, followed by a discussion of the results for each of the memories in turn.

A. Experimental setup

The experiment uses the tooling of the hybrid memory controller proposed in [8] together with three different memories with different speeds: DDR2-400, DDR2-800, and DDR3-1600. All memories have a capacity of 512 Mb and 16-bit interfaces. The DDR2 memories have four banks, and the DDR3 memory eight. All memory efficiencies, e, are bounded using the approach presented in [16]. Note that this is worst-case efficiency assuming large requests, making the results independent of the application workload. All three algorithms generate a set of patterns for burst counts 1, 2, and 4 with a burst length of 8 words. We also generate pattern sets with burst count 1 and burst length 4 for the DDR2 memories. To reduce the run-time of the B&B algorithm, the lengths of the access patterns generated by the bank scheduling algorithm were used as initial shortest patterns. This significantly reduces the search space without the possibility of removing the shortest access patterns.

B. DDR2-400

First up is the DDR2-400 memory. Table I lists the lengths of the resulting patterns for the different algorithms. We have merged the results for the B&B algorithm and the bank scheduling algorithm, since they consistently provide the exact same pattern lengths for all tested memories. The table shows that all algorithms provide patterns with the same length for $BL = 4$. In fact, they even provide the exact same patterns. The reason is that the low burst count and short burst length results in short patterns, where the memory timings do not allow a lot of options. In contrast with $BL = 8$, we observe that the ASAP scheduling algorithm generates write patterns that are five cycles longer than those generated by the other algorithms. As explained in Section V-D, this is because scheduling the activate commands as soon as possible causes the distance to the corresponding write commands to gradually increase, causing a problem with precharges. Having a longer write pattern is not completely without advantages. We observe that the patterns generated by the ASAP algorithm often has shorter write/read switching patterns and refresh patterns. The reason is that the five NOPs at the end of the write patterns hide some of the time required to switch direction of the data bus, or to precharge all banks.

TABLE I
Results for the DDR2-400 memory.

BL/BC	4/1	8/1	8/2	8/4
t_{read}	11	16	32	64
t_{write}	13	16	32	64
t_{rtw}	0	2	2	2
t_{wtr}	0	4	4	4
t_{ref}	27	32	32	32
e	60.5%	82.5%	89.6%	93.6%

(a) B&B & Bank scheduling

BL/BC	4/1	8/1	8/2	8/4
t_{read}	11	16	32	64
t_{write}	13	21	37	69
t_{rtw}	0	2	2	2
t_{wtr}	0	0	0	0
t_{ref}	27	27	27	27
e	60.5%	74.9%	85.0%	91.1%

(b) ASAP scheduling

Table I shows how the bounds on memory efficiency and net bandwidth vary between the different algorithms for the DDR2-400 memory. The B&B algorithm and the bank scheduling algorithm perform identically, having generated patterns with the same length. However, the patterns with $BL = 8$ generated by ASAP scheduling algorithm provide lower efficiency than the other algorithms, due to the longer write patterns. This difference is most pronounced for the patterns with $BC = 1$, where ASAP scheduling results in a reduction in memory efficiency by $1 - 0.749/0.825 = 10.2\%$. This shows that the choice of algorithm may have considerable impact on how efficiently the memory controller uses the scarce and expensive SDRAM bandwidth.

As far as the run-times of the algorithms are concerned, the ASAP scheduling and bank scheduling algorithms provided all results in a matter of seconds. The B&B algorithm managed to produce patterns with low burst counts in comparable time. However, the pattern set with $BC = 4$ took 8 days to generate. Such a long run-time clearly motivates the existence of the heuristic algorithms.

C. DDR2-800

We proceed by looking at the results for the DDR2-800, the fastest device in the generation of DDR2 memories. The patterns generated for this memory are listed in Table II. The difference between the algorithms is that ASAP scheduling again generates longer write patterns for some values of burst count and burst length. The increase is slightly less severe than for the DDR2-400, since the precharging constraints are more favorable for this memory.

Looking at the memory efficiency for the different algorithms in Table II, we observe that the ASAP scheduling algorithm is not performing worse than the B&B algorithm and bank scheduling. In fact, the longer write patterns result in that the memory efficiency is marginally *increased* by 0.1% for $BC = 2$! This is explained by observing that increasing the write pattern with three cycles removes three cycles from the write/read switching pattern, eliminating the disadvantage. The slight increase in efficiency stems from that the longer write pattern also allows the refresh pattern to be three cycles shorter. This demonstrates the disadvantage of the first design decision in Section V-A, although the impact is negligible.

TABLE II
RESULTS FOR THE DDR2-800 MEMORY.

BL/BC	4/1	8/1	8/2	8/4
t_{read}	22	22	33	65
t_{write}	22	22	33	65
t_{rtw}	0	0	1	1
t_{wtr}	1	3	5	5
t_{ref}	57	57	58	58
e	0.349	0.668	0.872	0.924

(a) B&B & Bank scheduling

BL/BC	4/1	8/1	8/2	8/4
t_{read}	22	22	33	65
t_{write}	22	22	36	68
t_{rtw}	0	0	1	1
t_{wtr}	1	3	2	2
t_{ref}	57	57	55	55
e	34.9%	66.8%	87.3%	92.4%

(b) ASAP scheduling

TABLE III
RESULTS FOR ALL ALGORITHMS WITH THE DDR3-1600 MEMORY.

BL/BC	8/1	8/2	8/4[1]
t_{read}	64	70	133
t_{write}	64	70	133
t_{rtw}	0	0	0
t_{wtr}	4	9	9
t_{ref}	98	103	103
e	47.7%	84.5%	91.6%

[1] The B&B algorithm did not finish after 10 days for this setting.

Considering the run-times of the algorithms, just like for DDR2-400, all patterns were generated in a few seconds except for $BC = 4$, which took the B&B algorithm 32 minutes.

D. DDR3-1600

Our last memory in the experiment is a DDR3-1600. Apart from the difference in memory generation and frequency, it comes with 8 banks instead of 4. The generated patterns for this memory are shown in Table III. The results from all algorithms are merged, since they always provide patterns of the same lengths for this memory. A possible reason for this is that eight banks solves the precharging problem of the ASAP algorithm, since the last activate command slips further into the pattern. Eight bank memories also have the additional FAW constraint, which limits the number of activate commands in a window of $tFAW$ cycles. This constraint helps spacing the activate commands in the pattern more evenly, further mitigating the precharging issue. However, this constraint does not primarily make patterns shorter. Both access patterns with $BC = 1$ have five NOP commands in the end to ensure that the FAW constraint is satisfied also when the patterns are repeated after themselves. The additional banks also impact the run-time of the B&B algorithm. More banks imply more commands to schedule, creating more possible patterns. The B&B algorithm required 7 days to generate the pattern set with $BC = 1$, although the set with $BC = 2$ was generated in seconds. The algorithm had not successfully generated a pattern set with $BC = 4$ after 10 days when we terminated the experiment. As in the previous experiments, the other two algorithms produced all results in seconds.

VII. CONCLUSIONS

Predictable systems reduce the verification complexity of real-time systems, but they require predictable software and hardware components, such as interconnects and memories. A predictable SDRAM controller has been proposed that is based on predictable memory patterns, which are pre-computed sequences of SDRAM commands. However, these memory patterns must be manually derived, which is a time-consuming and error-prone process that must be repeated for each SDRAM device and may result in inefficient use of scarce and expensive bandwidth.

This paper presents three algorithms for automatic generation of memory patterns that provide different trade-offs between run-time and guaranteed bandwidth. The first algorithm uses a branch and bound (B&B) search and exhaustively covers all useful parts of the design space. The other two algorithms are heuristics that schedule commands as-soon-as-possible (ASAP) and bank-by-bank, respectively. We experimentally compare the algorithms for several DDR2/DDR3 memories and pattern configurations and draw four conclusions: 1) The choice of algorithm matters, since the difference between the best and the worst algorithm is up to 10.2% of guaranteed bandwidth. This is a significant improvement, since SDRAM bandwidth is a scarce and expensive resource. 2) The B&B search provides high bandwidth, but is too slow for faster memories with more banks. 3) ASAP scheduling executes in seconds, but often reduces bandwidth compared to B&B for memories with four banks, due to precharge constraints. 4) Bank scheduling provides the same bandwidth as the B&B algorithm in just a second, making this the preferred algorithm.

REFERENCES

[1] C. van Berkel, "Multi-core for Mobile Phones," in *Proc. DATE*, 2009.
[2] P. Kollig *et al.*, "Heterogeneous Multi-Core Platform for Consumer Multimedia Applications," in *Proc. DATE*, 2009.
[3] O. Moreira *et al.*, "Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor," in *Proc. EMSOFT*, 2007.
[4] L. Steffens *et al.*, "Real-Time Analysis for Memory Access in Media Processing SoCs: A Practical Approach," *Proc. ECRTS*, 2008.
[5] A. Hansson *et al.*, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *IET CDT*, 2009.
[6] S. Stuijk *et al.*, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Proc. DAC*, 2007.
[7] E. A. Lee, "Absolutely positively on time: what would it take?" *IEEE Trans. Comput.*, vol. 38, no. 7, 2005.
[8] B. Akesson *et al.*, "Predator: a predictable SDRAM memory controller," in *Proc. CODES+ISSS*, 2007.
[9] S. Bayliss and G. Constantinides, "Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search," in *Proc. FPT*, 2009.
[10] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers," *IEEE Micro*, vol. 29, no. 1, 2009.
[11] J. Shao and B. Davis, "A burst scheduling access reordering mechanism," in *Proc. HPCA*, 2007.
[12] C. Macian *et al.*, "Beyond performance: Secure and fair memory management for multiple systems on a chip," in *Proc. FPT*, 2003.
[13] K. Lee *et al.*, "An efficient quality-aware memory controller for multimedia platform SoC," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 5, 2005.
[14] S. Heithecker and R. Ernst, "Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements," in *Proc. DAC*, 2005.
[15] A. Burchard *et al.*, "A real-time streaming memory controller," in *Proc. DATE*, 2005.
[16] B. Akesson *et al.*, "Classification and Analysis of Predictable Memory Patterns," in *Proc. RTCSA*, 2010.
[17] *DDR2 SDRAM Specification*, JESD79-2F ed., JEDEC Solid State Technology Association, 2009.
[18] *DDR3 SDRAM Specification*, JESD79-3E ed., JEDEC Solid State Technology Association, 2010.
[19] B. Akesson, "Predictable and Composable System-on-Chip Memory Controllers," Ph.D. dissertation, Eindhoven University of Technology, 2010.