Classification and Analysis of Predictable Memory Patterns

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

***Abstract*—The verification complexity of real-time requirements in embedded systems grows exponentially with the number of applications, as resource sharing prevents independent verification using simulation-based approaches. Formal verification is a promising alternative, although its applicability is limited to systems with predictable hardware and software. SDRAM memories are common examples of essential hardware components with unpredictable timing behavior, typically preventing use of formal approaches. A predictable SDRAM controller has been proposed that provides guarantees on bandwidth and latency by dynamically scheduling memory patterns, which are statically computed sequences of SDRAM commands. However, the proposed patterns become increasingly inefficient as memories become faster, making them unsuitable for DDR3 SDRAM.**

This paper extends the memory pattern concept in two ways. Firstly, we introduce a *burst count* parameter that enables patterns to have multiple SDRAM bursts per bank, which is required for DDR3 memories to be used efficiently. Secondly, we present a *classification* of memory pattern sets into four categories based on the combination of patterns that cause worst-case bandwidth and latency to be provided. Bounds on bandwidth and latency are derived that apply to all pattern types and burst counts, as opposed to the single case covered by earlier work. Experimental results show that these extensions are required to support the most efficient pattern sets for many use-cases. We also demonstrate that the burst count parameter increases efficiency in presence of large requests and enables a wider range of real-time requirements to be satisfied.

***Index Terms*—predictability; SDRAM; memory controller; memory patterns; burst count; classification**

I. Introduction

Embedded system design gets increasingly complex. More and more resources, such as processing elements and memories, are added to enable parallel execution of an increasing number of applications [1]. Some of these applications have hard real-time requirements that must be satisfied to prevent significant quality degradation, or even to guarantee the functional correctness of the system [2]. Platform resources are shared between applications to reduce cost. Resource sharing creates interference between applications, making their temporal behaviors inter-dependent. This creates a verification problem, since real-time requirements are typically verified by slow system-level simulation. All possible combinations of concurrently executing applications must hence be verified, causing the *verification complexity to grow exponentially* with the number of applications [3]. An alternative to simulation-based verification is to formally verify that real-time requirements are satisfied using a performance analysis framework, such as network calculus [4] or data-flow analysis [5]. Formal verification covers all interactions between applications, greatly reducing the verification effort. However, this approach requires *predictable systems*, where worst-case response times are known for all software and hardware components [6].

SDRAM memories are prominent examples of essential hardware components with unpredictable behavior. These memories are very common, since they provide large amounts of storage at low cost. The challenge of SDRAM is that the time to serve a request is highly variable and depends on previous requests. This makes it difficult to derive useful bounds on the latency and bandwidth provided to a memory *requestor*, which is a processing element that accesses the memory on behalf of an application. The bandwidth provided by these memories is furthermore a scarce resource that must be efficiently utilized, as it is one of the main performance bottle-necks [2]. Most existing memory controllers are either statically or dynamically scheduled. Statically scheduled controllers are not flexible enough to deal with dynamic application behavior, while dynamically scheduled controllers are typically unpredictable and cannot provide hard guarantees on bandwidth and latency. Predator [7] is a hybrid memory controller that provides bandwidth and latency guarantees, while increasing flexibility over earlier predictable controllers. This is accomplished using *predictable memory patterns*, which are statically computed sub-schedules of SDRAM commands that are dynamically scheduled at run-time. However, the provided theory only supports a limited set of memory patterns that are inefficient with fast memories, such as DDR3 SDRAM.

The four main contributions of this paper are: 1) We introduce *burst count* as a pattern parameter that considers that newer faster memories require larger accesses with multiple bursts per bank to be accessed efficiently. This enables a wider range of real-time requirements to be satisfied. 2) We present a *classification* of memory pattern sets into four categories based on the combination of patterns that cause worst-case bandwidth and latency to be provided. 3) *Bounds on bandwidth and latency* are derived that cover all pattern types and burst counts, as opposed to the single case covered in [7]. 4) We experimentally demonstrate and discuss *memory efficiency trends* for DDR2 and DDR3 memories. Possible applications of this work include integration with performance verification tools, such as worst-case execution time estimators.

This paper is organized as follows. In Section II, we review related work. Section III introduces the SDRAM architecture and explains why it is difficult to provide useful bounds on bandwidth and latency. The original concept of memory patterns is recapitulated in Section IV. Section V then extends the idea by parameterizing the number of bursts in a pattern, and presenting a classification of memory pattern sets. Bandwidth and latency bounds are then derived for all types of pattern sets in Section VI and Section VII, respectively. Experimental results are provided in Section VIII, followed by conclusions in Section IX.

II. Related work

Most SDRAM controllers are either statically or dynamically scheduled, depending on the type of systems they target. Statically scheduled controllers, such as [8], execute precomputed schedules of SDRAM commands that have been computed at design time. These controllers are predictable, since the latency and bandwidth provided to a requestor can be bounded at design time by analyzing the schedule. For this reason, statically scheduled memory controllers are most frequently used in embedded systems with hard real-time requirements. However, the predictability of these controllers comes at the expense of flexibility. The precomputed schedules limit the applicability to applications whose requestors have regular access patterns and where the request sizes and read/write ratio do not change during a use-case. Furthermore, many schedules have to be computed and stored, as the number of use-cases grows exponentially with the number of applications [3]. These properties prevent statically scheduled controllers from scaling to larger systems with more requestors and more dynamic applications.

Dynamically scheduled memory controllers, on the other hand, generate and schedule SDRAM commands dynamically based on available requests. These controllers target high efficiency and flexibility to fit in high-performance systems with dynamic applications whose behaviors may not be known at design time. Several of these controllers feature sophisticated mechanisms to reduce latency or improve efficiency. Examples involve preference for requests that target open rows in the memory banks [9]–[13], or that fit with the current direction of the data bus [11]–[15]. The problem with these controllers is that the interaction between all these mechanisms is complex, making the controllers unpredictable. For this reason, neither of the mentioned controllers provides bounds on latency or bandwidth. A predictable dynamically scheduled controller targeting hard real-time requirements is presented in [16]. However, this approach is limited to systems that are performance monotonic [17], which means that local reductions in execution time cannot result in longer overall execution times. It is shown in [18] that this property does not hold for all systems.

A hybrid memory controller that combines elements of statically and dynamically scheduled designs is presented in [7]. Predictability is achieved by using statically computed memory patterns that can be dynamically scheduled using any predictable arbiter. However, the provided theory only supports a limited subset of memory patterns that becomes increasingly inefficient as SDRAM memories become faster. This paper addresses these limitations by proposing more general and efficient patterns.

III. SDRAM overview

This section explains why SDRAM memories are challenging to use in systems with real-time requirements. First, Section III-A provides a brief overview of the SDRAM architecture. Section III-B then discusses the concept of memory efficiency and presents five reasons for the variable bandwidth and latency of SDRAMs.

A. SDRAM architecture

An SDRAM device comprises a number of banks, each containing a memory array with a matrix-like structure, consisting of rows with word-sized columns. A simple illustration of this architecture is shown in Figure 1. As an example, a 512 megabit (Mb) DDR2-400 [19] chip with a word width of 16 bits has 4 banks, each with 8192 rows containing 1024 word-sized elements. Since each column holds an element of 16 bits, it follows that a row contains 2 kilobytes (KB) of data.

On an SDRAM access, the logical address of the request is decoded into a physical address (bank, row and column) using a memory map. 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 of either 4 or 8 words for DDR2/DDR3 SDRAM. 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. A formal definition of an SDRAM memory is provided in Definition 1.

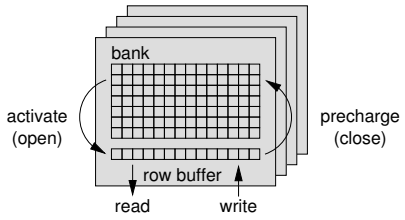


Fig. 1. The SDRAM architecture and some important SDRAM commands.

Definition 1 (SDRAM memory): An SDRAM memory is defined as (n_{banks}, w, f, d, BL), where n_{banks} is the number of banks, w is the width of the data bus in bytes, f is the clock frequency of the memory in MHz, d is the data rate (number of data words that can be transferred during a clock cycle), and BL is the programmed burst length in words.

B. Memory efficiency

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*, defined in Definition 2. For example, a 16-bit DDR2-400 memory has a peak bandwidth of 800 MB/s, since it has a clock frequency of 200 MHz, a data rate of 2 words per clock cycle, and a data bus width of two bytes. However, the peak bandwidth of SDRAMs cannot be fully utilized, due to refresh and stall cycles caused by numerous

timing constraints. The constraints are typically minimum delays between particular commands, such as between two activates, between a read and a precharge, or between a precharge and an activate. This is captured by the concept of *memory efficiency*, which is the fraction of clock cycles with useful data on the data bus. A useful classification of memory efficiency into five categories is presented in [20]. The categories are: 1) refresh efficiency (e^{ref}), 2) read/write efficiency (e^{rw}), 3) bank efficiency (e^{bank}) , 4) command efficiency (e^{cmd}), and 5) data efficiency (e^{data}). Memory efficiency is the product of these five categories, as stated in Definition 3. This definition is used to determine the *net bandwidth*, defined in Definition 4, provided by the memory controller. This is the remaining bandwidth that is useful to the requestors after considering all types of overhead. An important trend in SDRAMs is that timing constraints, expressed in nanoseconds, do not change much between generations. A consequence of this is that the constraints measured in clock cycles are increasing as memories are clocked at higher frequencies, resulting in reducing efficiency for faster memories. This makes it increasingly challenging to provide useful bounds on net bandwidth.

Definition 2 (Peak bandwidth): The peak bandwidth of a memory device is defined as $b^{peak} = f \cdot d \cdot w$.

Definition 3 (Memory efficiency): Memory efficiency is defined as $e^{mem} = e^{ref} \cdot e^{rw} \cdot e^{bank} \cdot e^{cmd} \cdot e^{data}$.

Definition 4 (Net bandwidth): The net bandwidth of a memory device is defined as $b^{net} = b^{peak} \cdot e^{mem}$.

We proceed by briefly summarizing the five categories of memory efficiency and discuss their typical impact on net bandwidth. This gives an idea of why it is difficult to provide a useful bound on net bandwidth at design time.

1) Refresh efficiency: Refresh efficiency accounts for the cycles that are lost due to refreshing the memory array to prevent data loss. A refresh command must be issued every refresh interval ($tREFI$), which is 7.8 μs on average for all DDR2 and DDR3 devices at normal operating temperatures. The time required to complete this operation depends on the time required to precharge all banks and on the size of the memory device, as larger devices require more time to refresh. The refresh efficiency can be estimated at design time with reasonable accuracy and is typically between 95-99% for all DDR2 and DDR3 memories [20].

2) Read/write efficiency: SDRAMs have a bi-directional data bus that requires time to switch from read to write and vice versa. This results in lost cycles as the direction of the data bus is being reversed. The read/write efficiency depends on the number of read/write switches, which cannot be determined at design time in the general case. Bounding read/write efficiency by assuming a switch after every SDRAM burst results in less than 75% efficiency for any DDR2 memory and less than 65% for DDR3, indicating a severe loss of net bandwidth.

3) Bank efficiency: A read or a write command can be issued immediately to columns in the active row. However, if a command targets an inactive row, it first requires a precharge followed by an activate command. This hence requires a variable number of overhead cycles that depends on the state of the memory. This overhead is captured by bank efficiency, which is highly dependent on the logical addresses of requests and how they are mapped to the different rows and banks of the memory by the memory map. Therefore, it is not possible to give a general estimate on the impact of this efficiency. Bank efficiency can be bounded by assuming a bank miss for every SDRAM burst. However, this results in less than 40% bank efficiency for any DDR2 memory and 20% for DDR3, which are useless bounds for a scarce resource [7].

4) Command efficiency: Even though a DDR device transfers data on both the rising and the falling edge of the clock, commands can only be issued once every clock cycle. Sometimes a required activate or precharge command has to be delayed because another command is already issued in that clock cycle. This results in lost cycles when a read or write command has to be postponed due to a row miss. The impact of this is connected to the burst length, as smaller bursts result in more activate and precharge commands. Command efficiency is traffic dependent and can generally not be calculated at design time, but is estimated to be between 95-100% [20].

5) Data efficiency: Data efficiency is defined as the fraction of a memory access that actually contains requested data. This can be less than 100%, since SDRAM memories are accessed with a minimum burst length of 4 words for DDR2 and DDR3 SDRAM. The problem is not only related to fine-grained requests, but also to how data is aligned with respect to a memory burst. This is because a burst is required to access BL words from an address that is evenly divisible by the burst length. The data efficiency of a requestor can be computed at design time if the minimum access granularity of the memory, and the size and alignment of requests are known.

IV. Memory patterns

Providing useful bounds on latency and bandwidth to SDRAM requestors at design time is clearly a challenging task. This section summarizes how this is done by the hybrid memory controller proposed in [7].

Command scheduling in the hybrid controller is based on predictable memory patterns, which are precomputed sub-schedules of SDRAM commands that are known to satisfy the timing constraints of the memory. These patterns are 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: a read pattern, a write pattern, a read/write switching pattern, a write/read switching pattern, and a refresh pattern. The read and the write pattern are referred to as *access patterns*, while the remaining patterns are called *auxiliary patterns*. Five scheduling rules determine how the memory patterns may be dynamically scheduled by the memory controller. The rules are: 1) Memory patterns are scheduled in a non-preemptive manner, which means that a pattern that has been issued cannot be stopped until it has finished. 2) A read or a write pattern can be scheduled immediately after itself, or when the memory is idle. 3) A write pattern following a read pattern must be preceded by a read/write switching pattern. Similarly, a read pattern following a write pattern must be preceded by a

write/read switching pattern. 4) A read or a write pattern can be scheduled immediately after a refresh pattern. 5) A refresh pattern can be scheduled after a read pattern, a write pattern, another refresh pattern, or if the memory is idle. Figure 2 shows an example of how requests are mapped to memory patterns.

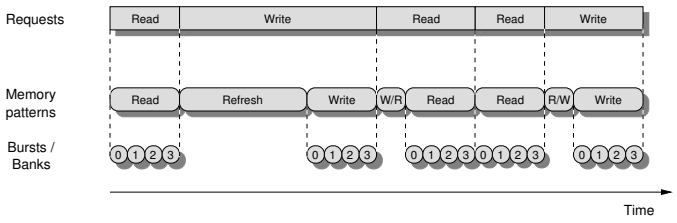


Fig. 2. Mapping from requests to patterns to SDRAM bursts.

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. It is not possible to assume that the correct rows are open in any of the banks, so an access pattern must contain an activate command for each bank. The read or write bursts are issued with the auto-precharge flag, implementing a closed page policy. This ensures that the banks are precharged as soon as possible after an access, resulting in shorter patterns. Figure 3 shows an example read pattern for a DDR2-400 memory. 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.

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

Memory pattern sets are formally defined in Definition 5. The definition considers the lengths of the patterns in the set, corresponding to the number of commands in the 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. This information in combination with the scheduling rules enables the hybrid memory controller to bound net bandwidth and worst-case latencies. Pattern sets are automatically generated by a tool, based on the timing constraints of the memory device. Three algorithms for memory pattern generation are presented in [21]. These algorithms first try to generate the shortest possible read and write patterns, given the timing constraints of the memory device. They then generate the shortest auxiliary patterns that satisfy the scheduling rules.

Definition 5 (Pattern set): A 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.

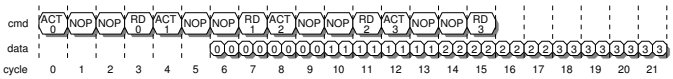


Fig. 3. Read pattern with $BL = 8$ for a DDR2-400.

V. Extensions

This section extends the memory pattern concept to address two important limitations. First, Section V-A introduces a pattern parameter called burst count that determines the number of SDRAM burst per bank in read and write patterns. This parameter enables memory efficiency to be increased for faster memories, such as DDR3. Section V-B then introduces a classification of pattern sets into four types, depending on the combination of patterns that result in the longest latency and lowest memory efficiency. This enables us to derive bounds on latency and bandwidth for all types of pattern sets and burst counts, as opposed to the single case covered by earlier work.

A. Burst count

A problem with the memory patterns proposed in [7] is that they become increasingly inefficient as memories become faster, making them unsuitable for DDR3 SDRAM. A single burst to each bank only requires $BL/2 \cdot n_{banks}$ clock cycles to complete, which may be less than the minimum number of cycles required between two activates to a bank. It is also possible that there have not been enough cycles for the first bank to finish precharging at this time. Both of these cases prevent an access pattern from being repeated after itself, which is required by the scheduling rules. It hence becomes necessary to extend the access pattern with NOPs until the constraints are satisfied, reducing efficiency and increasing latency. This problem becomes increasingly severe for faster memories, since the number of cycles with data transfer is constant, while the number of overhead cycles increases, as explained in Section III-B. This causes the memory efficiency of a DDR2-1600 to be 35% lower than for a DDR2-400, as we will see in Section VIII.

We propose to address this problem by adding a pattern parameter that determines the number of consecutive bursts to each bank in a pattern. This parameter is referred to as *burst count* and is defined in Definition 6. The key idea is that increasing burst count increases the number of cycles between successive activates to a bank, removing the effects of the activate and precharge constraints. It furthermore increases the number of cycles with data transfer in a pattern, thus increasing efficiency by amortizing overhead due to read/write

switches. However, it also increases the access granularity of the memory, defined in Definition 7. More data hence has to be read or written on every access and requests smaller than the access granularity of the pattern are masked or padded to fit. This has a negative impact on data efficiency, as we will see when bounding memory efficiency in Section VI and when computing net bandwidth for a range of example memories in Section VIII.

Definition 6 (Burst count): The burst count of an access pattern is denoted by BC, and is defined as the number of consecutive SDRAM bursts per bank in the pattern.

Definition 7 (Access granularity): The access granularity in bytes of an access pattern is defined as $g = BC \cdot BL \cdot n_{banks} \cdot w$.

B. Pattern dominance

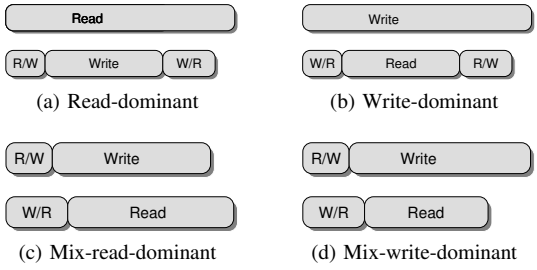
Before bounding the net bandwidth or latency of a pattern set, it is essential to determine which sequence of patterns produces the worst results. There are four different possibilities, depending on the relations between the lengths of the patterns in the set. We hence introduce a classification with four *pattern dominance classes*. This improves on the work in [7] that only supports one type of pattern sets. Although this is a common class, it is not always the most efficient, as we will see in Section VIII. We proceed by defining the dominance classes and show how to determine the classification of a pattern set.

A pattern set is classified as *read-dominant* when the read pattern is longer than the write pattern and both the switching patterns put together. This is formally defined in Definition 8 and illustrated in Figure 4a. In this case, the lowest bandwidth and longest latencies occur when all interfering requests are reads, i.e. when only read patterns and an occasional refresh pattern are issued. Conversely, a pattern is considered *write-dominant* if the write pattern is longer than the combined lengths of the read pattern and both the switching patterns. This case is defined in Definition 9, and an example is shown in Figure 4b. It follows by the earlier reasoning that the worst-case bandwidth and latency for a write-dominant pattern set occurs when all interfering requests are writes, resulting in that only write patterns and refresh patterns are issued. Pattern sets that are neither read-dominant nor write-dominant are referred to as *mix dominant* sets, defined in Definition 10. For these sets, the worst-case bandwidth and latency is provided when interfering requests alternate between reads and writes, causing as many switches as possible. The definitions of the dominance classes are all expressed in terms of the read pattern to clearly show that the classes are mutually exclusive and jointly exhaustive.

Definition 8 (Read-dominant pattern set): A pattern set is defined as read-dominant iff $t_{read} > t_{write} + t_{wtr} + t_{rtw}$.

Definition 9 (Write-dominant pattern set): A pattern set is defined as write-dominant iff $t_{write} > t_{read} + t_{wtr} + t_{rtw}$, which is equivalent to $t_{read} < t_{write} - t_{wtr} - t_{rtw}$.

Definition 10 (Mix-dominant pattern set): A pattern set is defined as mix-dominant iff $t_{write} - t_{wtr} - t_{rtw} \leq t_{read} \leq t_{write} + t_{wtr} + t_{rtw}$.



(a) Read-dominant (b) Write-dominant

(c) Mix-read-dominant (d) Mix-write-dominant

Fig. 4. Example pattern sets illustrating the four different dominance classes.

The division into three dominance classes is sufficient to bound net bandwidth. However, to accurately determine worst-case latency, mix-dominant pattern sets are further subdivided into two categories: *mix-read-dominant* and *mix-write-dominant* sets. The reason is that we need to know if an odd number of interfering requests result in more read patterns or write patterns in the worst case. A mix-read-dominant pattern set corresponds to a mix-dominant set in which the lengths of the write to read switching pattern and the read pattern is greater than or equal to that of the read to write switching pattern and the write pattern. Otherwise, the pattern set is mix-write-dominant. Mix-read-dominant and mix-write-dominant pattern sets are formally defined in Definition 11 and Definition 12, respectively. The corresponding example pattern sets are illustrated in Figure 4c and Figure 4d.

Definition 11 (Mix-read-dominant pattern set): A mix-dominant-pattern set is defined as mix-read-dominant iff $t_{wtr} + t_{read} \geq t_{rtw} + t_{write}$, which is equivalent to $t_{read} \geq t_{write} - t_{wtr} + t_{rtw}$.

Definition 12 (Mix-write-dominant pattern set): A mix-dominant pattern set is defined as mix-write-dominant iff $t_{rtw} + t_{write} > t_{wtr} + t_{read}$, which is equivalent to $t_{read} < t_{write} - t_{wtr} + t_{rtw}$.

VI. Memory efficiency bounds

We have now extended the concept of memory patterns with a burst count parameter that enables fast memories to be efficiently accessed and showed how to categorize pattern sets into different dominance classes, based on the situation that triggers the worst-case bandwidth and latency. This section derives lower bounds on memory efficiency that holds for all burst counts and classes of pattern sets. This is an improvement over [7] that only covers the special case of mix-read-dominant pattern sets with $BC = 1$. We proceed by walking through each of the efficiency categories presented in Section III-B and describe how to bound them.

A. Refresh efficiency

We explained in Section III-B that the refresh efficiency depends on the time to precharge all banks and execute a refresh command, and the refresh period. The difficulty with accurately bounding refresh efficiency is to know how long it takes to precharge all banks, since this depends on the state of the memory. Memory patterns solve this problem, since the precharge cycles for all banks are known at design time by looking at the generated patterns. The length of the refresh

pattern, t_{ref}, hence accounts for all time lost due to refresh operations.

The refresh period is controlled by a timer that triggers every $tREFI$ clock cycles (corresponding to 7.8 μs for all DDR2 and DDR3 memories). At this point, the memory controller prepares to schedule a refresh pattern. However, the scheduling rules state that a refresh pattern can only be issued after a read or write pattern has finished. The longest blocking time, t_{block}, before a refresh pattern can be issued is hence determined by the largest sum of a write/read switching pattern and a read pattern, and a read/write switching pattern and a write pattern. This is expressed in Equation (1), which is independent of the dominance class of the pattern set. A refresh pattern is hence scheduled every 7.8 μs on average, but with some occasional jitter due to blocking. This jitter does not jeopardize the integrity of the data in the memory array unless it is greater than $8 \cdot tREFI$ [19], [22], which is a very long time in comparison to the time it takes to execute any reasonable pattern. We now bound refresh efficiency according to Equation (2).

$$t_{block} = \max(t_{wtr} + t_{read}, t_{rtw} + t_{write}) \tag{1}$$

$$e^{ref} = 1 - \frac{t_{ref}}{tREFI} \tag{2}$$

B. Read/write efficiency

The read/write efficiency accounts for the time lost to switching direction of the data bus. Using memory patterns, we know that the read/write efficiency corresponds to the maximum fraction of time spent executing read/write and write/read switching patterns. This can be determined at design time, since the length of the patterns and the scheduling rules are known. The read/write efficiency is straight-forward to determine for read-dominant and write-dominant pattern sets, since these issue only read or write patterns in the worst case. Since the worst case does not contain any switches, it follows that the read/write efficiency is 100% for these sets. However, if the set is mix-dominant, there is a switch after every read and write pattern in the worst case. The read/write efficiency is hence determined by the time required to execute a read and write pattern divided by the time to execute the patterns and their corresponding switches, as shown in Equation (3).

$$e^{rw} = \begin{cases} 1 & \text{if read-dominant or write-dominant} \\ \frac{t_{read}+t_{write}}{t_{read}+t_{write}+t_{wtr}+t_{rtw}} & \text{if mix-dominant} \end{cases} \tag{3}$$

C. Bank and command efficiency

The bank efficiency accounts for the overhead associated with activating and precharging banks. The memory patterns allow us to tightly bound this efficiency, since the timings of all activates and precharges are known at design time. We compute the bank efficiency by determining the fraction of cycles of a read and a write pattern where data is actually transferred. However, this also accounts for any overhead due to command conflicts that may delay activate or precharge commands and result in a longer read or write pattern. Although it may be possible to distinguish this loss, we conveniently choose to compute bank efficiency and command efficiency as an aggregate. The aggregate bank and command efficiency is computed by first determining the number of cycles that data is transferred during a read pattern or a write pattern, denoted by $t_{transfer}$. This is calculated by considering that there are BC bursts of BL words to each of the n_{banks}, and that d words are transferred every clock cycle (2 for all DDR memories). This is expressed in Equation (4). For read-dominant pattern sets, we simply divide the data transfer cycles with the length of the read pattern. Conversely for write-dominant sets, we divide the transfer cycles with the length of the write pattern. Lastly for mix-dominant sets, we consider the fraction of transfer cycles during one read and one write pattern. This is expressed formally in Equation (5). Increasing burst count increases $t_{transfer}$, but also t_{read}, and t_{write}. However, the bank and command efficiency increases with burst count, since the impact of timing constraints is mitigated in longer patterns. This is experimentally demonstrated in Section VIII.

$$t_{transfer} = \frac{BC \cdot BL \cdot n_{banks}}{d} \tag{4}$$

$$e^{bank} \cdot e^{cmd} = \begin{cases} \frac{t_{transfer}}{t_{read}} & \text{if read-dominant} \\ \frac{t_{transfer}}{t_{write}} & \text{if write-dominant} \\ \frac{2 \cdot t_{transfer}}{t_{read}+t_{write}} & \text{if mix-dominant} \end{cases} \tag{5}$$

D. Data efficiency

Data efficiency corresponds to the amount of data that is transferred over the data bus that is useful to the requestors. The data efficiency of a requestor is determined by how the size and alignment of its requests fits with the minimum access granularity of the memory, as previously discussed in Section III-B. The minimum access granularity in our approach is equal to the granularity of an access pattern, computed according to Definition 7. This is a drawback of the memory pattern approach, since the access granularity of a pattern is larger than that of a single SDRAM burst, which is the minimum access granularity of the memory device itself. Increasing burst count exacerbates this problem, resulting in that the gains in bank and command efficiency may be lost in data efficiency in the presence of small requests. This indicates that faster memories become increasingly dependent on large requests. The data efficiency of a requestor r is computed according to Equation (6), where s_r denotes the request size of the requestor in bytes. For simplicity, we assume that all requests from a requestor are the same size and aligned with respect to the access granularity of the memory. The general case is covered in [21]. Equation (6) can be used to determine the total data efficiency of the memory in the special case where the request sizes and alignments of all requestors are the same. Otherwise, the data efficiency depends on how frequently the different requestors are scheduled, which is determined by the memory arbiter.

$$e_r^{data} = \frac{s_r}{g} \tag{6}$$

VII. Latency bounds

We have shown how to bound net bandwidth, based on how the patterns in a pattern set are dynamically combined in the worst case. We now proceed by showing how to derive the maximum latency of a request, given a maximum number of interfering requests with sizes equal or less than the access granularity of the access patterns. We choose this particular metric, since the hybrid controller cuts larger requests into smaller pieces of this size before arbitration [21]. The maximum latency of a request is defined as the total length of interfering memory patterns. This accounts for all switching patterns and access patterns related to different requests and to refresh patterns.

Our first step towards bounding the worst-case latency of a request is to disregard the refresh patterns and compute the maximum latency caused by read, write and switching patterns in the presence of x interfering requests. The maximum latency in this case depends on the dominance class of the pattern set, as shown in Equation (7). If the set is read-dominant, then all interfering requests are assumed to be reads. In this case, the worst case contains an initial write/read switch, followed by x read patterns. By the same logic, all interfering requests are assumed to be writes for write-dominant patterns. The worst-case latency for mix-dominant patterns happens if the interfering requests alternate between reads and writes, resulting in the maximum number of interfering switching patterns. Which type of access pattern there are more of depends on whether the pattern set is mix-read-dominant or mix-write-dominant, as shown in Equation (7). Note that increasing burst count results in longer access patterns. This implies that while it increases the net bandwidth in the presence of large requests, it also increases the worst-case latency. We return to investigate this trade-off experimentally in Section VIII.

$$t_{aux}(x) = \begin{cases} t_{wtr} + t_{read} \cdot x & \text{if read-dominant} \\ t_{rtw} + t_{write} \cdot x & \text{if write-dominant} \\ \left\lceil \frac{x}{2} \right\rceil \cdot (t_{wtr} + t_{read}) + & \\ \left\lfloor \frac{x}{2} \right\rfloor \cdot (t_{rtw} + t_{write}) & \text{if mix-read-dominant} \\ \left\lceil \frac{x}{2} \right\rceil \cdot (t_{rtw} + t_{write}) + & \\ \left\lfloor \frac{x}{2} \right\rfloor \cdot (t_{wtr} + t_{read}) & \text{if mix-write-dominant} \end{cases} \quad (7)$$

Next, we account for interference due to blocking and refresh, and compute the total worst-case latency, t_{tot}. Blocking occurs because the worst-case latency of a request may start counting from a moment just after a scheduling decision has been taken by the memory arbiter. This results in that maximally one additional request may interfere with the requestor due to the non-preemptive nature of memory patterns. We account for this by adding one extra interfering request to the bound, thus using $t_{aux}(x+1)$ to compute the maximum interference from x requests. To compute the maximum interference from refresh patterns, we must consider the minimum distance between two of these patterns. This distance occurs if one refresh pattern is maximally blocked (t_{block}) by other patterns, and the following refresh pattern encounters no blocking. In this case, the time between two consecutive refresh patterns

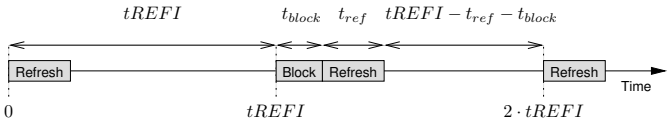


Fig. 5. The minimum distance between two refresh patterns.

is $tREFI - t_{ref} - t_{block}$, as illustrated in Figure 5. For every such interval, we add the time to execute a refresh pattern to the latency from other interfering patterns, as shown in Equation (8). This approach is somewhat pessimistic, since two such worst-case intervals cannot occur multiple times in a row. However, this pessimism only affects requestors with very long memory latencies that are in the range of several refresh periods. The equation rounds the number of interfering refresh patterns up, reflecting that a refresh can happen immediately in an arbitrary interval. Hence, all requestors always have at least one refresh pattern in their worst-case latency.

A key feature of Equation (8) is that it does not make *any assumptions* about the arbiter, since the number of interfering requests is left as a parameter. This *separates the analysis of the arbiter and the resource*, enabling the controller to be used in a predictable manner with a variety of arbiters. This is a differentiating feature with respect to the state of the art that enables the hybrid controller to satisfy a wider range of latency requirements.

$$t_{tot}(x) = \left\lceil \frac{t_{aux}(x+1)}{tREFI - t_{ref} - t_{block}} \right\rceil \cdot t_{ref} + t_{aux}(x+1) \quad (8)$$

VIII. Experimental results

This section evaluates the theory provided in this paper in four experiments. The first experiment bounds memory efficiency for a number of memories, assuming large requests. This demonstrates how memory efficiency fundamentally reduces for faster memories and how the problem is mitigated by increasing burst count. The second experiment takes data efficiency into account and shows that fast memories inherently requires large requests to efficiently bound net bandwidth. The third experiment evaluates the tightness of our bound on memory efficiency, and the last experiment investigates the bandwidth/latency trade-off when increasing burst count.

A. Experimental setup

The experiments in this section consider four different memories with different speeds: DDR2-400, DDR2-800, DDR3-800, DDR3-1600. These memories cover the span from the slowest specified memory in the DDR2 generation to the fastest specified DDR3 device. All memories have a capacity of 512 Mb and 16-bit interfaces. The DDR2 memories have four banks, and the DDR3 memories eight. The generalized theory has been implemented in the configuration flow of the memory controller [21]. The configuration flow supports three heuristic algorithms for memory pattern generation. Due to limited space, we cannot discuss these algorithms further in this paper. For all experiments, we generate patterns sets

TABLE I
LENGTH OF GENERATED PATTERNS FOR A DDR2-400 MEMORY.

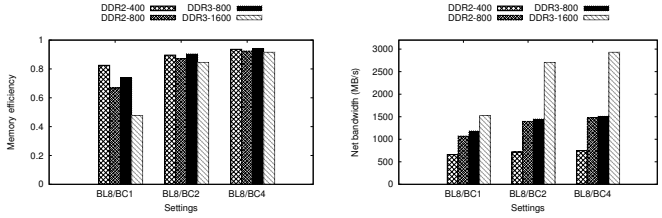
BL/BC Dominance	4/1 wr	8/1 mix rd	8/2 mix rd	8/4 mix rd
t_{read}	11	16	32	64
t_{write}	13	16	32	64
t_{rtw}	0	2	2	2
t_{wtr}	0	4	4	4
t_{ref}	27	32	32	32

using two of these algorithms (ASAP scheduling and bank scheduling) and choose the most efficient result. We disregard of a branch and bound algorithm, since this algorithm does not finish patterns with high burst count for fast memories within 10 days. Omitting this algorithm is not a severe restriction, as it has been shown to provide the same efficiency as bank scheduling for all finished patterns [21].

B. Bounding memory efficiency

This experiment demonstrates how burst count affects memory efficiency using the proposed analysis. This exercises our configuration flow, but does not use the implementation of the controller. We disregard data efficiency in this experiment by assuming that requests are equal to the access granularity of the memory. The effects of data efficiency are investigated in the following experiment. We use a DDR2-400 memory and generate sets of patterns with burst counts 1, 2, and 4, all with $BL = 8$. To provide a low-latency option, we also generate a pattern set with $BC = 1$ and $BL = 4$. Table I lists the lengths of the generated memory patterns. Note that the most efficient pattern set with $BL = 4$ is write dominant and hence not supported by earlier work. More details on the generated patterns are provided in [21].

Figure 6 shows the bounds on memory efficiency and net bandwidth for all considered memories. The generated pattern sets for these memories are all mix-read-dominant, except for the DDR2-800 with $BC = 2$ and $BC = 4$, which are both mix-write-dominant. Again, this shows that although mix-read-dominant patterns are common, they are not always most efficient. The extended search space provided by the proposed generalization increases the time required by the configuration flow, although it still finishes in a few seconds. Since the configuration flow runs at design time, *we conclude that the additional complexity of the new concepts is negligible.* Figure 6 allows us to draw two additional conclusions. 1) *Memory efficiency monotonically increases with burst count, assuming large requests*. This is seen for all tested memories in Figure 6a, and is the motivation for introducing the burst count parameter to the access patterns. 2) *Newer faster memories inherently offer higher peak bandwidths, but lower memory efficiency, due to increasingly severe timing constraints. However, the provided net bandwidth is still increasing with clock frequency if requests are large.* Figure 6a indicates that memory efficiency is reducing as memories get faster. The fact that net bandwidth is increasing despite the reducing memory efficiency is clearly shown in Figure 6b, where DDR3-1600 provides the highest net bandwidth with large requests.



(a) Memory efficiency comparison. (b) Net bandwidth comparison.

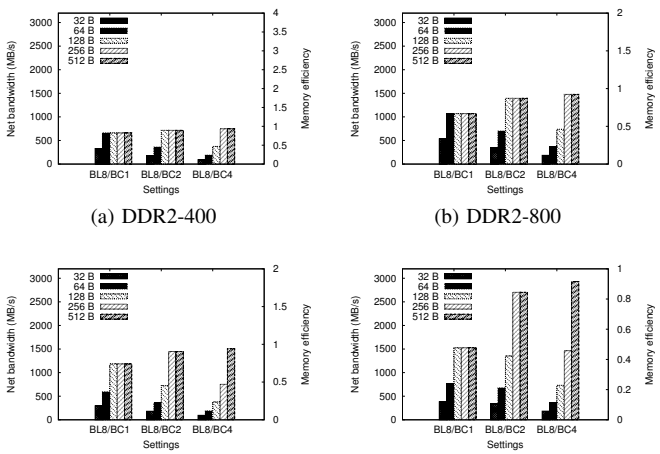
Fig. 6. Memory efficiency and net bandwidth comparisons between different DDR2 and DDR3 memories, assuming large requests.

C. The impact of small requests

For our second experiment, we study the impact of small requests when bounding the net bandwidth offered by the memories. This experiment uses the same memory patterns and analysis as the previous, except that data efficiency is taken into account. Figure 7 shows the bound on net bandwidth provided by the different memories and settings for different request sizes. For simplicity, we assume that the request sizes of all requestors are the same, taking the choice of memory arbiter out of the equation. The bars in the plot can be read from either y-axis, depending on if net bandwidth or memory efficiency is of interest. All graphs have the same scale, allowing the net bandwidths provided by the different memories to be compared. From this experiment, we learn that while increasing burst count consistently increases net bandwidth with large request, it may reduce if requests are small. The reason is that increasing burst count also increases the access granularity of the memory, resulting in more waste for small requests. Similarly, increasing the number of banks from 4 to 8 improves bank and command efficiencies, but can still reduce net bandwidth, due to the larger access granularity. A consequence of this behavior is that our DDR2-800 provides more net bandwidth for small requests than the DDR3-800. However, the tables turn as requests become big enough to benefit from the larger granularity. *We conclude from the figure that achieving high worst-case bandwidths with an interleaving memory map fundamentally requires large requests.* In fact, the DDR2 memories with 4 banks require requests of 64-128 B to provide a net memory efficiency of above 80%, while the DDR3 memories require requests of 256 B to accomplish the same. If the requests in the system are small, there is hence no benefit in using a faster SDRAM memory unless it is cheaper to buy. A good example of this is that the DDR2-800 with four banks provides the most net bandwidth for requests of 32 B.

D. Tightness of net bandwidth bound

In our third experiment, we evaluate the tightness of our lower bound on net bandwidth by simulation using a SystemC model of the hybrid controller. We measure the running average net bandwidth, which we expect to converge to a value greater than or equal to our derived bound during the simulation. The experiment is conducted by sending an equal mix of read and write requests to a DDR2-400 memory using the mix-read-dominant pattern set with $BL = 8$ and $BC = 1$,



(a) DDR2-400 (b) DDR2-800

(c) DDR3-800 (d) DDR3-1600

Fig. 7. Bound on net bandwidth for different memories and request sizes.

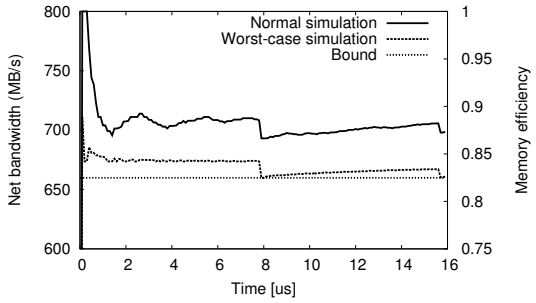


Fig. 8. Net bandwidth plotted over time for a DDR2-400 memory with and without worst-case switches.

previously shown in Table I. The sizes of the requests are 64 B, which is equal to the access granularity of the pattern, thus providing a data efficiency of 100%. The bound on net bandwidth with this setup is 660 MB/s. We simulate the memory controller twice. In the first simulation, we let read and write requests arrive in a random order. In the second simulation, arriving requests are alternating reads and writes to illustrate what happens during worst-case conditions. The simulation time in both cases is 100 ms. The results of this experiment are shown in Figure 8, where the provided net bandwidth is plotted over time. The figure shows the first 16 μs of the simulation, which is just enough to get two interfering refresh patterns. In both simulations, net bandwidth shoots towards 800 MB/s as the first request arrives. This is because the bank and command efficiency of the patterns is 100% and hence that data is transferred on every cycle of the pattern. The efficiency then gradually reduces as read/write switches cause lost cycles on the data bus. We note that the impact of these switches is considerably higher when the worst-case switching behavior is enforced. We see the effects of refresh at 7.8 μs and again at 15.6 μs, where the efficiency reduces due to the 32 idle cycles required to precharge all bank and refresh the memory. The measured bandwidth is very close to the bound at the end of the refresh pattern, indicating that this is the time at which the memory efficiency calculation "evens out". This is not surprising, considering that all events covered by the bound, such as read/write switches and refresh, have happened at this time. After 100 ms when the simulation ends, the worst-case simulation converges at a net bandwidth of 661.0 MB/s, which is less than 0.2% from the derived bound. This is not completely unexpected, since we have enforced exactly the behavior assumed by the bound. The normal simulation, on the other hand, converges at 694 MB/s, thus providing about 4% additional net bandwidth due to the reduced number of read/write switches. Similar results on tightness are observed when repeating the experiment for the other memories and burst counts with appropriate changes in requested bandwidth and request sizes. *We hence conclude that the bound on net bandwidth is tight.*

E. Bandwidth/latency trade-off

The fourth and last experiment investigates the bandwidth and latency trade-off that follows from the introduction of the burst count parameter. For this purpose, we generate 5000 synthetic use-cases, each with six requestors. The memory used in this experiment is a DDR2-400. The requestors issue requests with sizes $64 \cdot i$ bytes, where i is a uniformly varying integer in the range 1-8. Together, the requestors require 660 MB/s in all use-cases. This corresponds to 82.6% of the peak bandwidth offered by the memory, which is very close to 100% of the net bandwidth provided by the memory with $BC = 1$. The generated service latency requirements are randomized according to $27 \cdot j$ clock cycles at 200 MHz, where j is a uniformly varying integer in the range 1-100. Some latency requirements may hence be quite tight, while others may be quite relaxed and up to 50% longer than the refresh interval of the memory. To illustrate the bandwidth and latency trade-off, we let our configuration flow [21] configure the use-cases in four different ways. First, using only memory patterns with $BC = 1$, then using only patterns with $BC = 2$, followed by only using $BC = 4$. Lastly, we use an iterating scheme normally used by our configuration flow. This scheme tries all of these burst counts and chooses the best result. All generated patterns use $BL = 8$. We consider three metrics: 1) the percentage of use-cases where bandwidth requirements are satisfied for all requestors, 2) the percentage where latency requirements are satisfied for all requestors, and 3) the percentage where both bandwidth and latency requirements are satisfied for all requestors. The memory is shared using a Credit-Controlled Static-Priority arbiter [23], [24], comprising a rate regulator and a static-priority scheduler. The rate regulator has a precision of six bits in the service allocation mechanism, resulting in small, but not negligible, over-allocation due to discretization. The configuration flow assigns priorities according to an optimal algorithm that runs in polynomial time [25]. The results of this experiment are shown in Figure 9. Note that using a large number of synthetic benchmarks enables us to clearly illustrate the bandwidth and latency trade-off, although the actual numbers vary depending on the generated requirements.

Bandwidth requirements are satisfied in 21% of the use-cases with $BC = 1$, due to the high load required by the requestors in combination with over-allocation in the service

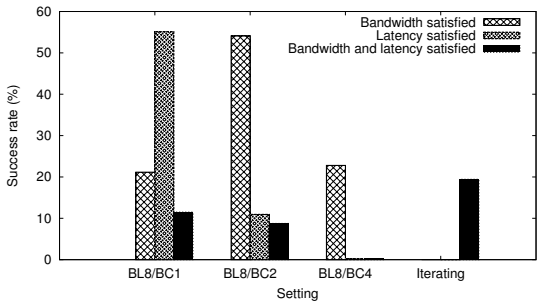


Fig. 9. The percentage of use-cases with bandwidth and latency requirements satisfied using pattern generators with fixed and iterating burst counts.

allocation mechanism in the rate regulator. The success rate increases to 54% with $BC = 2$, because of the extra 55 MB/s provided by the longer patterns. At this point, some requests may be larger than access granularity of the memory, being 128 B, although the reducing data efficiency does not yet eliminate the benefits of the increased bank efficiency. However, further increasing the burst count to $BC = 4$ reduces the percentage of satisfied bandwidth requirements to 23%, since the access granularity of 256 B is now too large compared to the sizes of the requests. This results in that more bandwidth is wasted than is added by the longer access patterns. While the percentage of use-cases with satisfied bandwidth requirements initially increases with burst count, the percentage of satisfied latency requirements monotonically decrease, starting at 55% with $BC = 1$, 11% for $BC = 2$, and ending at 0% with $BC = 4$. Looking at the percentage of use-cases with both bandwidth and latency requirements satisfied, we conclude that it is kept at approximately 10% for both $BC = 1$ and $BC = 2$, in the first case because of unsatisfied bandwidth requirements, and in the second case because of failing latency requirements. The total success rate with $BC = 4$ is 0%, due to the failing latency requirements. Lastly, we look at the results with the iterative approach that is normally used by our configuration flow. We ignore the separate results for bandwidth and latency requirements, since these depend on which pattern set is chosen for a use-case if either set of requirements fail. Instead, we focus on the percentage of use-cases where all requirements are satisfied. The iterative approach satisfies the requirements of almost twice as many use-cases as any of the fixed burst counts, clearly demonstrating the value of considering pattern sets with different burst counts.

IX. Conclusions

This paper addresses efficient use of SDRAM memories with formal verification of real-time requirements. A predictable memory controller has been proposed that provides hard bounds on bandwidth and latency by dynamically scheduling memory patterns, which are statically computed sequences of SDRAM commands. This controller enables formal verification of real-time requirements, although the proposed memory patterns are inefficient for faster memories, such as DDR3 SDRAM.

The main contributions of this work are: 1) A pattern parameter called burst count that considers that efficient accesses to faster memories require larger requests with multiple bursts per bank. 2) A classification of memory pattern sets into four categories based on the combination of patterns that cause worst-case bandwidth and latency to be provided. 3) Bounds on bandwidth and latency that cover all pattern types and burst counts. 4) Memory efficiency trends for DDR2/DDR3 memories are demonstrated and discussed. We experimentally evaluate the tightness of the bound on bandwidth and show that simulation results with worst-case stimuli deviates from the bound with less than 0.2%. Experiments also show that burst count enables real-time requirements to be satisfied for more use-cases and that the most efficient pattern sets are not always covered by earlier theory.

Future work involves integrating this analysis into the dataflow based performance verification tool of our MPSoC design flow. This enables latency and throughput requirements of applications to be verified and sufficient buffer sizes in the memory controller to be derived.

References

[1] "International Technology Roadmap for Semiconductors (ITRS) - System Drivers," 2007, http://www.itrs.net/reports.html.

[2] K. Goossens *et al.*, "Interconnect and memory organization in SOCs for advanced set-top boxes and TV — Evolution, analysis, and trends," in *Interconnect-Centric Design for Advanced SoC and NoC*. Kluwer, 2004, ch. 15.

[3] A. Hansson *et al.*, "Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip," in *Proc. DATE*, 2007.

[4] R. Cruz, "A calculus for network delay. I. Network elements in isolation," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, 1991.

[5] S. Sriram and S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*. CRC, 2000.

[6] E. A. Lee, "Absolutely positively on time: what would it take?" *IEEE Trans. Comput.*, vol. 38, no. 7, 2005.

[7] B. Akesson *et al.*, "Predator: a predictable SDRAM memory controller," in *Proc. CODES+ISSS*, 2007.

[8] S. Bayliss and G. Constantinides, "Methodology for designing statically scheduled application-specific sdram controllers using constrained local search," in *Proc. FPT*, 2009.

[9] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers," *IEEE Micro*, vol. 29, no. 1, 2009.

[10] J. Shao and B. Davis, "A burst scheduling access reordering mechanism," in *Proc. HPCA*, 2007.

[11] C. Macian *et al.*, "Beyond performance: Secure and fair memory management for multiple systems on a chip," in *Proc. FPT*, 2003.

[12] W.-D. Weber, *Efficient Shared DRAM Subsystems for SOCs*, Sonics, Inc, 2001, white paper.

[13] K. Lee, T. Lin, and C. Jen, "An efficient quality-aware memory controller for multimedia platform SoC," *IEEE transactions on circuits and systems for video technology*, 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] M. Paolieri *et al.*, "An analyzable memory controller for hard real-time cmps," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, 2009.

[17] J. Lee and K. Asanovic, "METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors," in *Proc. RTAS*, 2006.

[18] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. RTSS*, 1999.

[19] *DDR2 SDRAM Specification*, JESD79-2E ed., JEDEC Solid State Technology Association, Apr. 2008.

[20] L. Woltjer, "Optimal DDR controller," Master's thesis, University of Twente, 2005.

[21] B. Akesson, "Predictable and Composable System-on-Chip Memory Controllers," Ph.D. dissertation, Eindhoven University of Technology, 2010.

[22] *DDR3 SDRAM Specification*, Jesd79-3d ed., JEDEC Solid State Technology Association, Sep. 2009.

[23] B. Akesson *et al.*, "Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration," in *Proc. RTCSA*, 2008.

[24] ——, "Efficient Service Allocation in Hardware Using Credit-Controlled Static-Priority Arbitration," in *Proc. RTCSA*, 2009.

[25] N. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," *Real-Time Systems*, 1991.