# Predator: A Predictable SDRAM Memory Controller

Benny Akesson
Technische Universiteit Eindhoven
The Netherlands
k.b.akesson@tue.nl

Kees Goossens
NXP Semiconductors Research &
Delft University of Technology
The Netherlands
kees.goossens@nxp.com

Markus Ringhofer
Graz University of Technology
Austria
markus.ringhofer@tugraz.at

## ABSTRACT

Memory requirements of intellectual property components (IP) in contemporary multi-processor systems-on-chip are increasing. Large high-speed external memories, such as DDR2 SDRAMs, are shared between a multitude of IPs to satisfy these requirements at a low cost per bit. However, SDRAMs have highly variable access times that depend on previous requests. This makes it difficult to accurately and analytically determine latencies and the useful bandwidth at design time, and hence to guarantee that hard real-time requirements are met.

The main contribution of this paper is a memory controller design that provides a guaranteed minimum bandwidth and a maximum latency bound to the IPs. This is accomplished using a novel two-step approach to predictable SDRAM sharing. First, we define memory access groups, corresponding to precomputed sequences of SDRAM commands, with known efficiency and latency. Second, a predictable arbiter is used to schedule these groups dynamically at run-time, such that an allocated bandwidth and a maximum latency bound is guaranteed to the IPs. The approach is general and covers all generations of SDRAM. We present a modular implementation of our memory controller that is efficiently integrated into the network interface of a network-on-chip. The area of the implementation is cheap, and scales linearly with the number of IPs. An instance with six ports runs at 200 MHz and requires 0.042 $mm^2$ in 0.13$\mu$m CMOS technology.

**Categories and Subject Descriptors:** B.8.2 [Performance and reliability]: Performance Analysis and Design Aids

**General Terms:** Design, Reliability, Verification

**Keywords:** System-on-Chip, Memory Controller, SDRAM, Predictability

## 1. INTRODUCTION

Chip design is getting increasingly complex, as technological advances allow highly integrated systems on a single piece of silicon. A contemporary multi-processor system-on-chip (SoC) features a large number of intellectual property components (IP), such as streaming hardware accelerators and processors with caches, which communicate through shared memory. The resulting memory traffic is dynamic and the arrivals of requests at the memory controller are not fully known at design time. Some of the IPs have hard real-time requirements that must be satisfied to ensure the functional correctness of the SoC [5]. High-speed external memories, such as DDR2 SDRAMs [10], are used, as the memory capacity requirements of these systems cannot be satisfied in a cost-effective way by on-chip SRAM. These memories must be efficiently utilized, as they are one of the main SoC performance bottle-necks [5]. A difficulty when sharing SDRAM is that they have a highly variable access time that depends on previous requests, resulting in interference between the IPs sharing the resource, hereafter referred to as *requestors*. As a consequence, the amount of available bandwidth to/from external memory that is useful to the IP, referred to as *net bandwidth*, also depends on traffic [19]. These effects complicate analytical design-time verification, which is required to guarantee that hard real-time requirements are satisfied.

Existing external memory controllers are either not sufficiently flexible to handle the increasing complexity of SoCs, or do not support analytical design-time verification of hard real-time requirements. Statically scheduled memory controllers execute precomputed schedules, which makes them predictable, but also unable to adapt to changes in traffic and distinguish latency requirements of critical requestors without over-allocating. Other controllers use dynamic scheduling [13] that is flexible and maximize the offered net bandwidth, but where it is difficult to bound the latency of a request analytically. As a result, the offered net bandwidth can only be estimated by simulation, making bandwidth allocation a difficult task that must be re-evaluated every time a requestor is added, removed or changes configuration.

The main contribution of this paper is a memory controller design that provides a guaranteed minimum net bandwidth and a maximum latency bound to the requestors. This is accomplished using a novel two-step approach to predictable shared SDRAM access that combines elements of statically and dynamically scheduled memory controllers. First, we define read and write groups, corresponding to static sequences of SDRAM commands, with known efficiency and latency. This allows a lower bound on the offered net bandwidth to be determined. Second, requestors are scheduled dynamically at run-time by a Credit-Controlled Static-Priority (CCSP) [1] arbiter that is composed of a rate regulator and a scheduler. The rate regulator isolates requestors and guarantees their allocated net bandwidths independently of each other's behavior. The static-priority scheduler provides a maximum latency bound that is decoupled from the allocated bandwidth. The implementation of the design is modular and the area scales linearly with the number of requestors. An instance with six ports runs at 200 MHz and requires 0.042 $mm^2$ in 0.13$\mu$m CMOS technology.

This paper is organized as follows. In Section 2, we review related work. We proceed in Section 3 by briefly explaining the basic


*CODES+ISSS'07*, September 30–October 3, 2007, Salzburg, Austria.

operation of an SDRAM and defining the concept of memory efficiency. In Section 4, we propose a solution to the problem that allows us to verify at design time that bandwidth and latency requirements are satisfied. We show the implementation of our memory controller in Section 5, before discussing experimental results in Section 6. Finally, we present conclusions and future work in Section 7.

## 2. RELATED WORK

External memory controllers can be statically or dynamically scheduled. The level of predictability is high for statically scheduled controllers, as the latency of a request and the offered net bandwidth can be computed at design time. For this reason, statically scheduled memory controllers are used in some embedded systems with very strict real-time requirements, such as TV picture improvement ICs [15]. However, the precomputed schedule makes these controllers unable to adapt to any changes in traffic. Requestors have to wait for their reserved slots in the schedule before they receive service, which makes latency inversely proportional to the allocated bandwidth. This coupling between latency and allocated bandwidth makes these controllers unable to distinguish requestors with low latency requirements without reserving more slots in the schedule, resulting in low memory utilization. Finally, a large number of schedules have to be computed and stored in memory, as convergence of application domains in new SoCs causes a combinatorial explosion with respect to the number of use-cases [8]. These properties prevent statically scheduled controllers from scaling to larger systems with more requestors and more dynamic applications.

Dynamically scheduled memory controllers are more prevalent in areas where the traffic is not known up front, and target high flexibility and efficiency, at the expense of predictability. They offer sophisticated features, such as support for preemption and reordering [13], to optimize the offered net bandwidth and average latency. These features complicate design-time analysis and make it very difficult to provide analytical latency bounds.

The dynamically scheduled SDRAM controllers in [9, 11, 18] all use priorities to decouple latency and rate, and provide rate regulation to isolate requestors and prevent starvation. The arbitration in [11] supports preemption for high-priority requestors to reduce latency. Like [12, 18], it furthermore considers the memory state when scheduling to increase the amount of net bandwidth. No analytical bounds are presented on latency or on net bandwidth for any of these controllers thus preventing analytical design-time verification of hard real-time requirements. A streaming memory controller is presented in [2] that uses an access granularity of a full row (1 KB) to minimize power. Although this granularity allows the amount of net bandwidth to be determined at design time, it results in high latencies and large buffering. The scheduler is similar to weighted round-robin, which provides isolation of requestors through rate regulation, but does not decouple latency and rate. No general latency bounds are presented for this controller.

## 3. SHARING SDRAM

This section addresses the difficulties with sharing SDRAM in a predictable manner. First, Section 3.1 briefly presents the basics of SDRAM operation. We proceed in Section 3.2 by defining memory efficiency, and giving some insight into the complexities of determining the net bandwidth provided by an SDRAM at design time.

### 3.1 SDRAM operation

SDRAMs have a three dimensional layout. The three dimensions are banks, rows and columns. A bank stores a number of word-sized elements in rows and columns, as shown in Figure 1.

| *Description* | *Parameter* | *Min. cycles* |
|---|---|---|
| ACT to ACT cmd delay (same bank) | $tRC$ | 11 |
| ACT to ACT cmd delay (diff. bank) | $tRRD$ | 2 |
| ACT to RD or WR cmd delay | $tRCD$ | 3 |
| PRE to ACT cmd delay | $tRP$ | 3 |
| Average REF to REF cmd delay | $tREFI$ | 1560 |
| REF to ACT cmd delay | $tRFC$ | 15 |
| CAS latency | $CL$ | 3 |
| Write recovery time | $tWR$ | 3 |
| Write-to-read turn-around time | $tWTR$ | 2 |
| Read-to-write turn-around time | $tRTW$ | 4 / 6[a] |

**Table 1: Relevant timing parameters for a 32Mb x16 DDR2-400B memory device.**

[a] 4 cycles if $BL = 4$ and 6 cycles if $BL = 8$

On an SDRAM access, the address of the request is decoded into bank, row and column addresses 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 onto a row buffer, which stores the most recently activated row. Once the bank has been activated, column accesses such as *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 4 or 8 words (for DDR2 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 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.

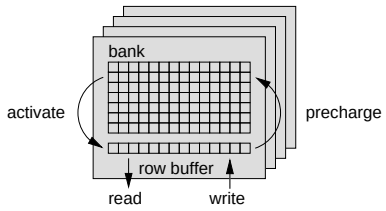


**Figure 1: The multi-banked architecture of SDRAM.**

SDRAMs have timing constraints defining the required minimum delay between different commands. Table 1 summarizes the most relevant constraints for a 64 MB DDR2-400B [10] device, which serves as example memory throughout this paper. The core of this memory runs at 200 MHz and the data bus at 400 MHz, as it transfers data on both the rising and falling edges of the core clock.

A benefit of a multi-bank architecture is that commands to different banks can be pipelined. While data is being transferred to or from a bank, the other banks can be precharged and activated with another row for a later request. This process, called *bank preparation*, sometimes completely hides the precharge and activate delays.

### 3.2 Memory efficiency

The bandwidth available from a memory ideally corresponds to the product of the word width and the clock frequency, referred to as the *peak bandwidth* (800 MB/s for our device). SDRAMs cannot, however, be fully utilized due to stall cycles caused by the timing constraints of the memory or memory controller policies. This is captured by the concept of *memory efficiency*. Memory efficiency is defined as the fraction of the amount of clock cycles in

which requested data is transferred, and the total number of clock cycles. Net bandwidth is hence the product of the peak bandwidth and the memory efficiency.

A useful classification of memory efficiency into five categories is presented in [19]. The categories are: 1) refresh efficiency, 2) read/write efficiency, 3) bank efficiency, 4) command efficiency, and 5) data efficiency. Memory efficiency can be expressed as the product of these efficiencies. All of these categories, except refresh efficiency, are traffic dependent, and hence difficult to determine at design time. We proceed by briefly looking into the different categories.

### *Refresh efficiency*

An SDRAM needs to be refreshed regularly in order to retain data integrity. A refresh command that requires $tRFC$ cycles to complete must be issued every $tREFI$ cycles on average. Before the refresh command is issued, all banks have to be precharged. Refresh efficiency, $e_{ref}$, is independent of traffic, and can be calculated at design time. Refresh efficiency depends on the memory size, and is typically between 95-99% for DDR2 memories.

### *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. In order to use the data bus of a DDR SDRAM at maximum efficiency, a read or write command must be issued every $BL/2$ cycles. We quantify the cost of switching directions as the number of extra cycles before the read or write command can be issued, which can be derived from [10]. The cost of a read/write switch is $t_{rtw} = tRTW - BL/2$ and for a write/read switch $t_{wtr} = CL-1+tWTR$, corresponding to 2 cycles and 4 cycles, respectively, for our example memory. Read/write efficiency, $e_{rw}$, depends on the number of read/write switches, and cannot be determined at design time, unless the read and write mix is known. The worst-case read/write efficiency for traffic consisting of 70% reads and 30% writes equals 93.8%, according to a formula presented in [19].

### *Bank efficiency*

The access time of an SDRAM is highly variable. A read or write command can be issued immediately to an active row. If a command, however, targets an inactive row (row miss), it first requires a precharge followed by an activate command. This requires at least an additional $tRP+tRCD$ cycles (6 cycles for our memory) before the read or write command can be issued. The penalty can be even larger, as $tRC$ cycles must separate one activate command from another within the same bank, according to Table 1. This overhead is captured by bank efficiency, $e_{bank}$, which is highly dependent on the target addresses of requests, and how they are mapped to the different banks of the memory by the memory map.

### *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, where smaller bursts result in lower efficiency. Command efficiency, $e_{cmd}$, is traffic dependent and can generally not be calculated at design time, but is estimated in [19] to be between 95-100%.

### *Data efficiency*

Data efficiency, $e_{data}$, is defined as the fraction of a memory access that actually contains requested data. This can be less than 100% since external memories are accessed with a minimum burst length; four words for a DDR2 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. Data efficiency is traffic dependent, but can be computed at design time if the minimum access granularity of the memory, and the size and alignment of requests are known. For example, a large aligned cache line from an L2 cache typically has a data efficiency of 100%. On the other hand, [19] computes a data efficiency of 75% for an MPEG2 stream. For the purpose of this paper, we assume aligned requests with a size that is a multiple of the fixed access granularity, unless otherwise noted, which results in a data efficiency of 100%.

We conclude from this section that it is, in general, difficult to determine memory efficiency analytically at design time. Figure 2 shows the worst-case memory efficiency (excluding data efficiency), resulting from simple worst-case analysis, for burst lengths of four and eight words. The calculated efficiency assumes that every burst targets a different row in the same bank, and that there is a read/write switch between every burst. This results in unacceptably low efficiency, less than 40% for all memories in the figure, due to bank conflicts (that hide the read/write switching cost). We conclude from the figure that we must be able to guarantee at design time that fewer bank conflicts occur to improve efficiency and provide a tighter bound than that of simple worst-case analysis. We furthermore see that the worst-case efficiency drops as the memories become faster, indicating that the problem is becoming more severe. The reason is that the timings in the memory core are more or less the same for all DDR2 memories. Increasing clock frequencies hence results in longer timings measured in clock cycles. This trend is expected to continue into the DDR3 generation of SDRAM.

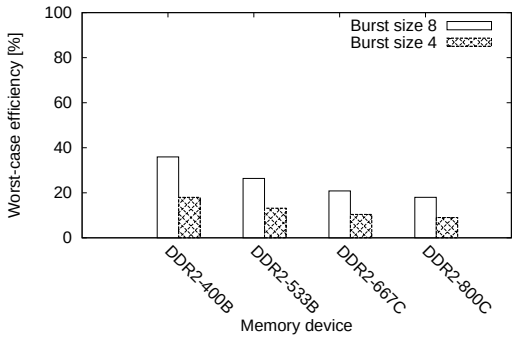


**Figure 2: Worst-case memory efficiency for a number of DDR2 memories.**

## 4. PROPOSED SOLUTION

In this section we present a novel approach to memory controller design that allows us to provide a net bandwidth guarantee and a maximum bound on latency. We present three memory access groups in Section 4.1 and derive a lower bound on memory efficiency based on the classification in Section 3.2. We proceed in Section 4.2 by explaining how a predictable arbiter schedules these groups dynamically at run-time to provide a guaranteed net bandwidth and maximum latency bound, while keeping requestors isolated.

## 4.1 Memory access groups

Memory access in our memory controller design is based on three memory access groups, a read group, a write group, and a refresh group. In this section, we show how to construct these groups to account for the different categories of memory efficiency, such that the efficiency of an arbitrary combination of groups can be bounded at design time. The groups used in this paper have been composed manually for our example memory, although the method is general and applies to all SDRAMs.

The read and write groups are composed of one read or write burst to all banks in sequence. This is a highly efficient way to access memory that makes maximum use of bank preparation. A limitation of this approach is that memory is accessed with a fixed granularity of $BL \times n_{banks} \times w_{mem}$ B, where $n_{banks}$ correspond to the number of banks and $w_{mem}$ to the width of the memory interface. This equals 64 B for our example memory with a burst length of eight words and four banks. Some SoCs, such as [4], are designed with the efficiency of SDRAM in mind, and choose the request sizes of the IPs as a multiple of this granularity where possible. This access granularity also fits well with the size of a typical L2 cache line, and even with the L1 cache line size of certain processors, such as the TriMedia 3270 [17]. Smaller grained accesses are supported by applying a read or write mask and throwing away unwanted data, which impacts the data efficiency of the solution.

The read and write groups are designed to eliminate interference between requestors. This is accomplished by ensuring that an arbitrary row can be activated without delay cycles caused by previous requests. According to [10], two timing constraints must be satisfied for this to hold. First, there must be at least $tRC$ cycles between consecutive activate commands to the same bank. Second, there must be at least $tWR + tRP$ cycles from the completion of the last write burst to a bank, before a new activate command can be issued to that bank. All bank conflicts are accounted for once these constraints are met. One read or write burst to each bank with burst length 8 satisfies these requirements for our example memory and yield a bank efficiency of 100%.

Figure 3 shows a read and a write group for our example memory. The numbers in the figure correspond to the number of the corresponding bank. We have wrapped the elements on the data bus with respect to the length of the groups. This shows that the groups are pipelined and very efficient, as they transfer data during all cycles when groups of the same type follow each other. All read and write commands are issued with auto precharge, since we want to be able to activate an arbitrary row as fast as possible. There is no contention on the command bus in these groups, and the command efficiency is hence 100%.

Unlike [19], we do not use the average read/write mix to compute the read/write efficiency, as this average may not be representative for shorter intervals. This may result in less available net bandwidth than computed during these intervals, potentially violating the bandwidth guarantees of the requestors. Our bound on read/write efficiency is computed according to Equation (1), where $t_{group}$ corresponds to the number of cycles in the read and write groups. The equation determines the fraction of cycles during one read and one write group that data is transferred, assuming a maximum number of switches. This results in a lower bound on read/write efficiency of 84.2%.

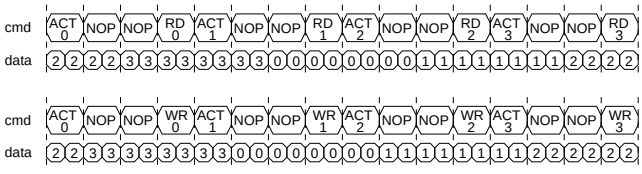


**Figure 3: Read group (above) and write group (below) with burst length 8 for DDR2-400.**

$$e_{rw} = \frac{2 \times BL/2 \times n_{banks}}{2 \times t_{group} + t_{rtw} + t_{wtr}} \quad (1)$$

The refresh group consists of a single refresh command issued after 10 NOP commands. This gives just enough time for the last auto precharge from a write group to finish. This is the worst case, as auto precharge after a read group finishes faster. The refresh requires $tRFC$ cycles to finish, giving the refresh group for our example memory a total length $t_{ref} = 25$ cycles. A refresh group is scheduled as soon as possible after $tREFI - t_{wtr} - t_{group}$ cycles. This avoids preempting a group in progress, while ensuring that a refresh command is issued at least every $tREFI$ cycles. The switching cost from write to read is used in the expression, as it is larger than its counterpart in the other direction for all DDR2 memories. It furthermore takes four cycles before the first read or write command is issued after the refresh group is finished, as shown in Figure 3. Refresh efficiency, for our example, hence equals 98.1%, according to Equation (2).

$$e_{ref} = 1 - \frac{10 + 4 + tRFC}{tREFI - t_{wtr} - t_{group}} \quad (2)$$

Having accounted for all categories of memory efficiency, we conclude that our solution provides a guaranteed efficiency of $e_{ref} \times e_{rw} \times e_{bank} \times e_{cmd} \times e_{data} = 0.981 \times 0.842 \times 1 \times 1 \times 1 = 82.6\%$ for aligned requests with a size that is a multiple of the fixed access granularity. This corresponds to 660.9 MB/s for our memory.

Read and write groups with burst length 4 have also been composed. These groups require 12 cycles to satisfy all timing requirements, and result in a memory efficiency of 60.3%. This choice may be beneficial for systems with smaller request sizes, as the access granularity is reduced to 32 B, which may increase data efficiency.

## 4.2 Arbitration

A predictable arbiter is required to schedule the memory access groups in order to bound latency. Our approach is valid for any arbiter providing a maximum latency bound, but we have chosen a Credit-Controlled Static-Priority arbiter [1] for our implementation. A CCSP arbiter consists of a rate regulator and a static-priority scheduler. This particular scheduler fits well with our requirements for five reasons: 1) it isolates requestors by means of rate regulation, 2) it guarantees an allocated bandwidth and a maximum latency bound, 3) it decouples latency and rate using priorities, 4) it has negligible over-allocation, and 5) it has a cheap RTL implementation and runs at high speed.

Requestors are characterized according to the $(\sigma, \rho)$ model [3], which uses a linear function to bound the amount of requested data in an interval. The bounding function is determined by two parameters, burstiness, $\sigma$, and average request rate, $\rho$. Burstiness can be interpreted as the maximum amount of data that can be requested at any instant in time. Since memory requests are served by the memory controller in a non-preemptive manner, the maximum request size, $\hat{s}$, for each requestor is also required to be specified. A graphical interpretation of this terminology is shown in Figure 4. The characterization of requestors is defined in Definition 1, where $R$ is the set of requestors sharing the resource. An abstract resource view is used in [1], where a service unit corresponds to the access granularity of the resource. We hence express $\sigma$ and $\hat{s}$ in units of the fixed granularity of the read and write groups. We furthermore express $\rho$ as the required fraction of the available net bandwidth.

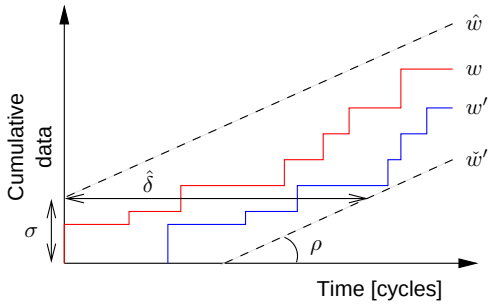


**Figure 4: Request curve and service curve along with their corresponding bounds.**

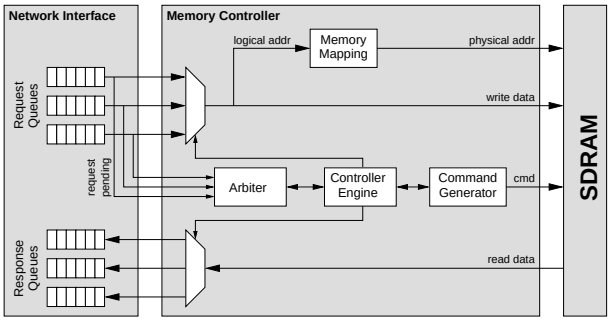


**Figure 5: Memory controller architecture.**

DEFINITION 1. *The amount of requested data, $w$, by a requestor $r \in R$ is characterized as ($\sigma$, $\rho$, $\hat{s}$). The amount of requested data is in any time interval $[\tau, t]$ upper bounded by $\hat{w}(t-\tau) = \sigma + \rho \times (t-\tau)$.*

The rate regulator isolates requestors from each other and protects requestors that behave according to their specification from those that do not, by enforcing the specified rate and burstiness. A requestor that does not behave according to its specification hence only invalidates its own latency guarantee. This is a key property in providing guaranteed service to requestors with timing constraints [20], and fits with the requirement that requestors must be isolated.

It is shown in [1] that a CCSP arbiter belongs to the class of latency-rate ($\mathcal{LR}$) servers [16] and guarantees a requestor its allocated bandwidth, $\rho$, and a maximum delay bound, $\hat{\delta}$. Delay is defined as the time a request spends from entering the request queue until it is scheduled. The maximum delay for a requestor with priority level $p$ (lower level indicates higher priority) is computed according to Equation (3), where $\max_{\forall r \in R} \hat{s}_r$ accounts for that requests in progress are not allowed to be preempted. This is a standard delay bound [3] for static-priority schedulers in combination with $(\sigma, \rho)$-constrained requestors that defines a guaranteed service curve, $\check{w}'$, as shown in Figure 4.

$$\hat{\delta}_p = \frac{\max_{\forall r \in R} \hat{s}_r + \sum_{j=0}^{p} \sigma_j}{1 - \sum_{j=0}^{p-1} \rho_j} \tag{3}$$

Due to the abstract resource view, the maximum delay in Equation (3) is expressed as the maximum number of read or write groups that are scheduled before the first group of the considered request. Equation (5) translates this into clock cycles. It is straightforward to compute the time before the first or last data word of the request is available in the response queue of the requestor, considering that the exact compositions of the groups are known. Equation (4) computes the time of the read and write groups, plus the maximum additional time required for read/write switches. This considers read requests, which is the worst-case, as their experienced switching cost is slightly higher than that of a write group. Equation (5) adds the maximum interference from refreshes during this interval.

$$t_{aux}(x) = x \times t_{group} + \left\lceil \frac{x+1}{2} \right\rceil \times t_{wtr} + \left\lfloor \frac{x+1}{2} \right\rfloor \times t_{rtw} \tag{4}$$

$$t_{tot}(x) = \left\lceil \frac{t_{aux}(x)}{tREFI - t_{wtr} - t_{group}} \right\rceil \times t_{ref} + t_{aux}(x) \tag{5}$$

# 5. CONTROLLER ARCHITECTURE

The proposed memory controller has been implemented in VHDL in the context of a multi-processor SoC that is interconnected using the Æthereal NoC [6]. The memory controller architecture, shown in Figure 5, is modular and consists of four functional blocks: 1) Controller Engine, 2) CCSP Arbiter, 3) Memory mapping, and 4) Command Generator.

Requests arrive at a network interface (NI) [14] on the edge of the network, where they are buffered in separate request queues per requestor. The requestors are mapped to request queues, according to their priorities. Priorities are changed between use-cases by using the reconfiguration abilities of Æthereal [7] to change the mapping of requestors to queues.

The NI signals to the arbiter, for each requestor, the size of the request at the head of the queue and if it is a read or a write. Once the memory controller is ready for a new group, the controller engine instructs the arbiter to schedule a requestor. The static-priority scheduler is implemented by a tree of multiplexors that grants access to the highest priority requestor that has a pending request that is considered eligible by the rate regulator. An index identifying the scheduled requestor is returned to the controller engine along with a read/write identifier. The controller engine uses this information to route the destination address to the memory mapping and data to and from the appropriate queue in the NI.

The memory mapping decodes the logical memory addresses of the requestors, into a physical SDRAM address consisting of bank, row and column. An interleaving memory map is used to map the bursts of a group to different banks, as mentioned in Section 4.1. For our example groups with burst length 8, this is implemented by letting bits 3 to 4 in the logical memory address index the bank, 12 to 24 index the row, and 0 to 2 and 5 to 11 index the column.

The command generator generates the SDRAM commands corresponding to the scheduled memory access group. This is the only part of the design that requires modification if the target memory is changed.

The design is small for two reasons. First, because there are no data buffers in the controller, as it re-uses buffering already present in the network interface. Second, unlike [9, 11], the command generator does not need to use registers to track the memory state, since the memory access groups are designed with the timing constraints of the memory in mind. Synthesis results in 0.13$\mu$m CMOS technology with six ports and a speed target of 200 MHz resulted in a total area of 0.042 mm$^2$. The arbiter is the dominant block of the synthesized instance, as it accounts for almost half the total area. Figure 6 shows a linear increase in the area of the arbiter as the number of ports increase. The sizes of the memory mapping, command generator and controller engine are constant and hence unaffected by the number of ports.

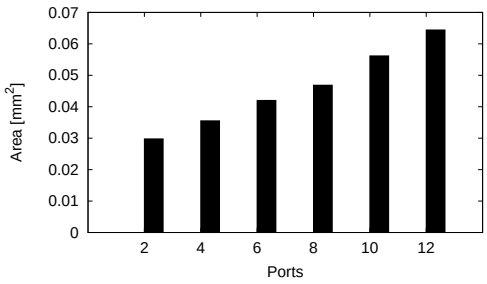


**Figure 6: Controller area depending on the number of ports.**

| *Requestor* | *Bandwidth [B]* | *Max [ns]* | *Bound [ns]* |
|---|---|---|---|
| $r_0$ | 16499968 | 204 | 340 |
| $r_1$ | 16500032 | 304 | 615 |
| $r_2$ | 16499968 | 463 | 1185 |
| $r_3$ | 16499968 | 732 | 2810 |

**Table 2: Results for use-case with four requestors with identical behavior.**

# 6. EXPERIMENTAL RESULTS

We demonstrate the analysis of the memory controller by observing simulation results from a SystemC model. The model is implemented on the level of basic groups and is timing accurate. For ease of understanding, we choose a simple use-case with four hard real-time requestors, mimicked by traffic generators. The requestors have identical behavior with net bandwidth requirements of 165 MB/s ($\rho = 0.249$) and burst sizes of 64 B ($\hat{s} = 1$). This corresponds to a total load of 660 MB, or 82.5% of the peak bandwidth. Priorities are assigned to the requestors in ascending order, such that $r_0$ have the highest priority, and $r_3$ the lowest. Requests are issued periodically, although jitter is introduced by the network. Setting $\sigma = 1.3$ accounts for this jitter and prevents the rate regulator from slowing down the requestors, as long as they behave according to their specification.

The use-case was simulated during $10^8$ ns. Table 2 shows that all requestors receive their allocated bandwidth, and that the maximum measured delay is less than the calculated bound for all requestors. We observe that the difference between the maximum measured delay and the bound gets larger for lower priority requestors. A reason for this is that the risk of maximum interference from higher priority requestors becomes increasingly unlikely for every requestor.

To illustrate the benefits of rate regulation, we doubled the requested bandwidth of $r_0$ without changing the requestor characterization in the rate regulator. Table 3 shows that this causes the maximum measured delay to explode for $r_0$, while the others still enjoy their guarantees. All requestors still receive their allocated bandwidth, and $r_0$ even receives some extra, since there is slightly more offered net bandwidth than indicated by the lower bound. This is because the worst-case amount of read/write switches did not occur. These results confirm that rate regulation allows us to give hard real-time guarantees to requestors that behave according to their specification in the presence of, for example, soft real-time requestors that cannot be accurately characterized.

# 7. CONCLUSIONS AND FUTURE WORK

In this paper, we present a memory controller design that provides a guaranteed minimum bandwidth and a maximum latency bound to the IPs. This is accomplished using a novel two-step approach to predictable SDRAM sharing. First, we define memory access groups, corresponding to precomputed sequences of SDRAM commands, with known efficiency and latency. Second, a predictable arbiter is used to schedule these groups dynamically at run-time, such that an allocated bandwidth and a maximum latency bound is guaranteed to the IPs. We present a modular implementation of our memory controller that is efficiently integrated into the network interface of a network-on-chip. The area of the implementation is cheap, and scales linearly with the number of IPs. An instance with six ports runs at 200 MHz and requires 0.042 mm$^2$ in 0.13$\mu$m CMOS technology.

| *Requestor* | *Bandwidth [B]* | *Max [ns]* | *Bound [ns]* |
|---|---|---|---|
| $r_0$ | 17327104 | 5316 | 340 |
| $r_1$ | 16500032 | 303 | 615 |
| $r_2$ | 16499968 | 364 | 1185 |
| $r_3$ | 16499968 | 737 | 2810 |

**Table 3: Results for use-case where $r_0$ is over-asking.**

Future work in this direction involves developing an algorithm for automatic generation of memory access groups, given a set of memory timings and a burst size. With this it is possible to generate the VHDL code for the corresponding command generator. This significantly reduces the effort to provide a low-cost predictable memory controller design for an arbitrary SDRAM memory.

# 8. REFERENCES


[1] B. Akesson *et al.* Real-Time Scheduling of Hybrid Systems using Credit-Controlled Static-Priority Arbitration . Technical report, NXP Semiconductors, 2007. http://www.es.ele.tue.nl/~kakesson/publications/pdf/NXP-TN-2007-00119.pdf.

[2] A. Burchard *et al.* A real-time streaming memory controller. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2005.

[3] R. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Trans. Inf. Theory*, 37(1), 1991.

[4] S. Dutta *et al.* Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design and Test of Computers*, 2001.

[5] 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*, chapter 15. Kluwer, 2004.

[6] K. Goossens *et al.* The Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test. Comput.*, 22(5), Sept. 2005.

[7] A. Hansson and K. Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. In *The 1st ACM/IEEE International Symposium on Networks-on-Chip*, 2007.

[8] A. Hansson *et al.* Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Apr. 2007.

[9] S. Heithecker and R. Ernst. Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements. In *Proc. DAC*, 2005.

[10] JEDEC Solid State Technology Association. *DDR2 SDRAM Specification*, JESD79-2C edition, May 2006.

[11] T.-C. Lin *et al.* Quality-aware memory controller for multimedia platform SoC. In *IEEE Workshop on Signal Processing Systems, SIPS 2003*, 2003.

[12] C. Macian *et al.* Beyond performance: Secure and fair memory management for multiple systems on a chip. In *IEEE International Conference on Field-Programmable Technology (FPT)*, 2003.

[13] S. Rixner *et al.* Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, 2000.

[14] A. Rădulescu *et al.* An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 24(1), Jan. 2005.

[15] F. Steenhof, *et al.* Networks on chips for high-end consumer-electronics TV system architectures. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2006.

[16] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Trans. Netw.*, 6(5), 1998.

[17] J.-W. van de Waerdt *et al.* The TM3270 Media-Processor. In *Proc. MICRO 38*, 2005.

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

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

[20] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10), Oct. 1995.