# An analytical model for a memory controller offering hard-real-time guarantees

Benny Åkesson

Department of Information Technology
Lund University

Advisor:
Lambert Spaanenburg (Lund University)
Kees Goossens (Philips Research)

May 31, 2005

Printed in Sweden
E-huset, Lund, 2005

# Abstract

In this thesis we present an analytical memory controller model for embedded systems that bridges the gap between current static and dynamic solutions. The model delivers hard real-time net bandwidth guarantees and provides better flexibility than existing static memory controllers. A fixed back-end schedule, similar to that of a static memory controller design, defines how memory is accessed, and is used to do a gross to net bandwidth translation. The net bandwidth in the schedule is allocated to the requestors as credits by an allocation scheme offering hard real-time guarantees on net bandwidth. Access to the memory is provided by a dynamic front-end scheduler that increases the flexibility of the design yet provides a theoretical worst-case latency bound. The front-end scheduler is pluggable so that algorithms with different trade-offs in fairness, jitter bounds and buffering can be used. An example system is analytically verified and successfully simulated. The system runs with a maximum load of 89.3% of the peak memory bandwidth when optimized for maximum throughput. When optimizing for latency a theoretical worst-case latency bound of 550 ns for low latency requestors is provided.

# Table of Contents

Chapter 1

# Problem statement

This chapter introduces a problem present in the memory service of contemporary embedded systems. Section 1.1 describes the problem of satisfying the diverse communication requirements of memory clients. In Section 1.2 we present how two different directions of current memory controllers fail to address the problem, before we present our contribution in Section 1.3.

## 1.1 Communication requirements

Embedded systems of today can have a large number of memory clients, hereafter referred to as requestors, with diverse, and possibly conflicting, requirements. Some of them can have real-time requirements while others do not. In this section we identify and characterize different kinds of traffic and discuss requirements with respect to three axes: bandwidth, latency and jitter. For a more comprehensive overview one is referred to [1, 7, 23].

Non real-time traffic, such as memory requests from a cache miss by a CPU or a DSP, is irregular since it is bursty in nature and can occur at virtually any time. The processor is stalled while waiting for the cache line to be returned from the memory and thus the lowest possible latency is required to prevent wasting processing power. This kind of traffic requires *good average throughput* and a *low average latency* but cares little about the worst-case as long as it occurs infrequently.

There are two types of real-time applications: soft and hard. In a soft real-time application, data still has some value if delivered too

late whereas it becomes worthless in a hard real-time application. It follows that the soft real-time guarantees occasionally can be violated and hence can be statistical in nature. Embedded systems are more concerned with hard real-time, or absolute, guarantees as they are more application specific and need to be tailored to always meet their specification [7].

Consider a set-top box doing audio/video decoding. The requests and responses have predictable sizes and repeat periodically. This type of traffic requires a *guaranteed minimum bandwidth*. Low latency is good in this kind of system but it is more important that the latency is constant. Variations in latency, commonly referred to as jitter, cause problems since buffers are required in the receiver to prevent underflows causing stuttering playback. For this reason this kind of system requires *low bounded jitter*.

Embedded systems are often used to monitor and control potentially critical systems. Consider a control system in a nuclear power plant. Sensor input has to be delivered to the regulator before it is too late in order to prevent a potentially hazardous situation. This traffic requires guaranteed minimum bandwidth, *low worst-case latency* but is *jitter tolerant* as long as deadlines are never missed.

The CPU, set-top box and control system described above show the span of requirements. Good memory solutions can be designed for any of these systems. Difficulties arise in complex contemporary systems were all traffic types are present simultaneously. Such a system requires a flexible memory solution to address the diversity.

Bandwidth comes in two different flavors, *gross* and *net*, which further complicates the requirements. Gross bandwidth is a peak, or maximum theoretical, bandwidth measure that does not take memory efficiency into account. A gross bandwidth guarantee translates into guaranteeing the requestors a number of memory clock cycles, which is what most memory controllers do. Net bandwidth, on the other hand, is what the applications request in their specifications and corresponds to the useful bytes of data transferred per second. The translation from gross to net bandwidth is determined by the memory efficiency, which depends on the offered traffic. Unknown traffic may cause low memory efficiency resulting in wasted bandwidth. The difficulty in providing a net bandwidth guarantee is that details of how the traffic accesses

memory has to be well-known.

## 1.2 State of the art

There are two different types memory controller designs, static and dynamic. These types of controllers have very different properties, as we show next.

### 1.2.1 Static memory controllers

A static memory controller [18, 20] follows a hard-wired schedule to allocate memory bandwidth to requestors. The major benefit of static memory controllers is predictability; they offer guaranteed minimal bandwidth, maximal latency and bounds on jitter. Static memory controllers do not scale very well since the schedule has to be recomputed if more requestors are added to the system. The difficulties of calculating a schedule grows with an increasing number of requestors and may not be possible to do in run-time. A static memory controller is suitable in a system with predictable requestors but can not provide low latency to intermittent requestors. Due to the lacking flexibility a dynamic work-load is not handled well and results in low memory efficiency. It is well-known how the schedule accesses memory since it is pre-calculated. This makes it possible for static memory controllers to offer net bandwidth guarantees.

### 1.2.2 Dynamic memory controllers

Dynamic memory controllers [8, 9, 12, 14, 21] make scheduling decisions at run-time and adapt their behavior to the nature of the traffic. This makes them very flexible and allows them to achieve low average latency and high average throughput, even with a dynamic work-load.

The offered requests are buffered and one or more levels of arbitration decide which one to serve. The arbitration can be simple with static priorities or involve complex credit-based schemes with multiple traffic classes. While these arbiters can be made memory efficient, this comes at a price, as complex arbiters are required, which are slow,

require large chip area, and are difficult to predict. The lacking predictability of dynamic memory controllers makes it very difficult to offer hard real-time guarantees and calculate useful worst-case latency bounds. Dynamic memory controllers are often general purpose and do not make assumptions about how memory is accessed by its requestors. As a result, these controllers do not offer guarantees on net bandwidth. A way to derive such a guarantee is to perform extensive simulations in an attempt to cover the worst-case traffic, and then over-allocate resources to get a safety margin. This method is time-consuming, may result in low memory efficiency, and the resulting guarantees are statistical and not absolute.

## 1.3 Our contribution

The diverse communication requirements call for a flexible memory solution. It must provide hard real-time guarantees on net bandwidth, worst-case latency and jitter bounds while still providing good average performance. Current solutions do not simultaneously offer dynamic flexibility and hard real-time guarantees on net bandwidth, worst-case latency and jitter.

In this thesis we present a memory controller for embedded systems that bridges the gap between current static and dynamic solutions. Bandwidth is translated from gross to net on the bottom-level by fixing the sequence of commands sent to the memory. The net bandwidth is allocated to the requestors such that hard net bandwidth guarantees are provided. A top-level scheduler distributes the net bandwidth dynamically in run-time to increase flexibility.

Chapter 2

# Memories and controllers

This chapter provides information on memories and memory controllers to introduce the necessary terminology. Section 2.1 presents a brief overview of memories before we introduce our system model in Section 2.2. We describe the multi-bank layout of modern DRAMs in Section 2.3 and proceed to discuss memory efficiency in Section 2.4. A general memory controller architecture is described in Section 2.5 after which we present a brief outline of the proposed solution in Section 2.7.

## 2.1 Background

Random access memory, RAM, is a fundamental component in computer systems and has been for the past decades. It is used as intermediate storage for the processing units in the system, such as processors. There are several types of RAM targeting different requirements on bandwidth, power consumption, and manufacturing cost. There are two common types of RAMs: SRAM and DRAM. Static RAM, SRAM, was introduced in 1970 and offers high bandwidth and low access time. SRAM is often used for caches in the higher levels of the memory hierarchy to boost performance. The drawback of SRAM is cost since six transistors are needed for every bit in the memory array. The dynamic RAM, DRAM, was patented by Dennard in 1968. DRAM is considerably cheaper than SRAM, as it needs only one transistor and a capacitor per bit, but has a lower speed. The capacitor is charged with a high or low voltage to indicate a one or zero respectively. The term dynamic stems from that the capacitor is leaking current and needs to

be refreshed several hundred times per second. DRAM is used as main memory in most modern systems.

In the past ten years, there has been a number of improvements of the DRAM design. A clock signal has been added to the previously asynchronous DRAM interface to reduce synchronization overhead with the memory controller during burst transfers. This kind of memory is called synchronous DRAM, or SDRAM for short. In 2001 a new generation of SDRAM was introduced featuring significantly higher bandwidth. These memories transfer data on both the rising and the falling edge of the clock effectively doubling the bandwidth, hence the name double-data rate (DDR) SDRAM. The second generation of these DDR memories, called DDR2, is very similar in design but scales to higher clock frequencies and peak bandwidth.

## 2.2 System model

The system considered throughout this thesis consists of one or more requestors. The requestors are connected to the memory sub-system through an interconnect, such as direct wires, a bus or a network-on-chip. This is illustrated in Figure 2.1.

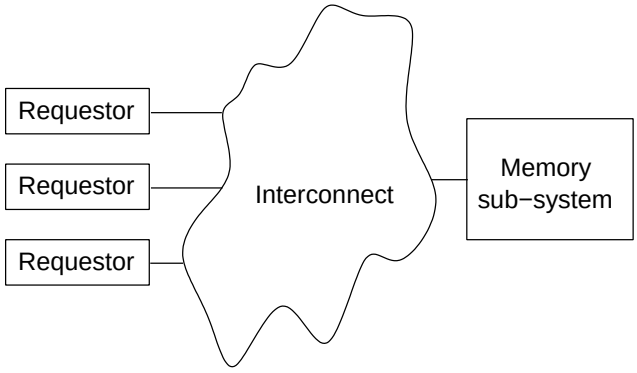


**Figure 2.1:** Three requestors are connected to the memory sub-system through an interconnect.

The memory sub-system consists of the memory controller and the memory, as shown in Figure 2.2. We use a channel buffer model to abstract the memory controller design from a particular interconnect.

Every requestor in the channel buffer model is associated with a request and a response buffer in the memory controller. These buffers provide a clock domain crossing so that the memory controller may operate at a different frequency than the interconnect.

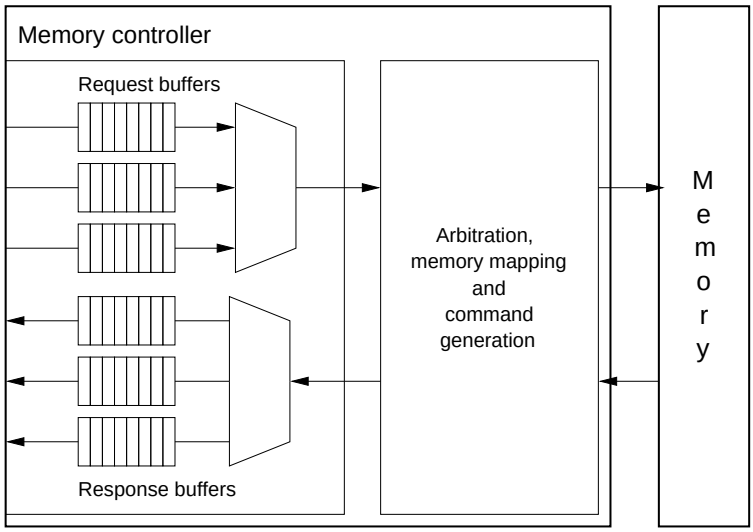


**Figure 2.2:** Every requestor is mapped to a request and a response buffer in the memory controller.

A requestor communicates with the memory sub-system through a connection. A connection is a bi-directional stream with request and response channels that connect a requestor to the corresponding buffers in the memory controller. The traffic characteristics and the desired priority level of a requestor are specified in use-case specifications. The admission controller of the memory controller accepts the service contract of a requestor provided that there are enough resources available to honor it. It then guarantees that the requirements are fulfilled as long as the requestor behaves according to the specification. We now formally define requestors and their associated properties.

**Definition 2.2.1.** *A direction, $d$, is defined to belong to the set $D = \{read, write\}$.*

**Definition 2.2.2.** *A requestor, $r$, is defined as a unit communicating with the memory controller through a connection and is formally ex-*

*pressed as a five-tuple ($d(r)$, $w(r)$, $\sigma(r)$, $t_{max}(r)$, $c(r)$). A requestor has a direction, $d(r) \in D$, determining if it is reading or writing. Every requestor is represented by a minimum requested data bandwidth, $w(r)$, a maximum request size in bytes, $\sigma(r)$, and a maximum latency, $t_{max}(r)$. A requestor is associated with a priority level, $c(r)$, from a totally ordered set of priority levels. All requestors belong to the set of requestors, $R$.*

The requestor is allowed to read or write, but not both. Separating reads and writes is required for us in order to provide bandwidth guarantees. We elaborate further on this decision in Chapter 4. Requestors communicate with the memory by sending requests. A request is sent on the request channel and is stored in the designated request buffer until it is serviced by the memory controller. In case of a read request the response data is stored in the response buffer until it is returned on the response channel.

The request buffer contains fields for command (read or write request), memory address, length of the request and, in case of a write command, write data. Requests in the request buffer are served in a first-come-first-served (FCFS) order by the memory controller, which thus provides sequential consistency within every connection, assuming that this is supported by the interconnect. No synchronization or dependency tracking is provided between different connections and must be supplied elsewhere.

Lin *et al.* presents a similar architecture in [12] but allows many requestors to aggregate their requests on a single connection. A benefit of this approach is that requestors that share a connection are assured that consistency between their requests is maintained. This, however, makes it impossible to offer net bandwidth guarantees to the requestors on a per-connection basis with reasonable traffic constraints and has thus not been considered further here.

With the architecture of the channel buffer model in mind it is possible to make a useful definition of latency. The total latency of a request in the memory sub-system is the sum of four components:

- Request queue latency
- Memory controller latency

- Memory latency
- Response queue latency

The request queue latency is the time from the arrival of the first word of a request in the request queue until it is dequeued by the memory controller. The memory controller latency is the time from the first word of the request has been dequeued until the corresponding read or write command has been posted on the command bus. The memory latency depends on the direction of a request and the state of the memory and is divided into read latency and write latency. Write latency is the time from the write command is issued until the first word is stored in the row buffer. It is specified in [11] to be at least $\mathrm{CL} - 1$ cycles, where CL is the CAS (column access strobe) latency. The read latency is the time from the read command is issued until the first word is returned on the data path. The read latency is at least CL cycles. The response queue latency is the time from when the first word arrives in the response queue until it is dequeued.

The request and response queue latencies are depending on the arrival and departure processes of the interconnect making them inappropriate metrics for the channel buffer model. The same goes for the memory controller latency, which depends on the implementation of the memory controller model and the memory latency that depends on the timings of the particular memory device. For this reason we choose to use another latency metric, service latency, defined in Definition 2.2.3. Service latency is measured from the moment a request is at the head of the request queue until the last word has left the queue. The service latency reflects how the memory controller schedules the memory but is independent of the interconnect and the timings of a particular memory device.

**Definition 2.2.3.** *The worst-case latency of a request of a requestor, $t_{lat}(r)$, refers to the service latency. Service latency is measured from the moment a request is in the head of the request queue until the last word has left the queue. It follows that the latency requirements of a system is satisfied if $t_{lat}(r) \leq t_{max}(r); \forall r \in R$.*

## 2.3 Modern memory layout

Modern DRAMs have a three dimensional layout, the three dimensions being banks, rows and columns. A bank is similar to a matrix in the sense that it stores a number of word sized elements in rows and columns. The described memory layout is depicted in Figure 2.3.

On a DRAM access, the address is decoded into bank, row and column addresses. A bank has two states, *idle* and *active*. A simplified DDR state diagram is shown in Figure 2.4. The bank is activated from the idle state by an *activate* command that loads the requested row onto the sense amplifiers, also known as the row buffer. Every bank has a row buffer, which is used to store the most recently activated row. Once the bank has been activated, column accesses such as *read* and *write* commands can be issued to the columns in the row buffer. A *precharge* command is issued to return the bank to the idle state. This stores the row in the buffer back into the memory array. A row is also referred to as a page that can be opened or closed depending on whether it is present in the row buffer or not. A memory access to a closed page is referred to as a *page fault* and results in lost cycles as the current page is closed and the requested one is opened.

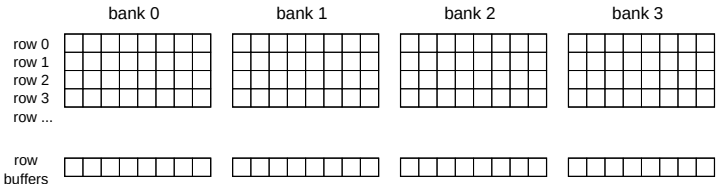


**Figure 2.3:** Layout of a multi-bank memory architecture.

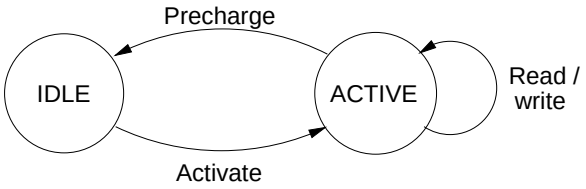


**Figure 2.4:** Simplified DDR state diagram per bank.

Reads and writes are done in bursts of 4 or 8 words. An open page

is divided into uniquely addressable boundary segments of the burst size that is programmed in the memory on initialization. This limits the set of possible start addresses for a burst and has consequences for efficiency, as we show later. Many systems experience spatial locality in memory accesses meaning that subsequent memory accesses often target memory addresses in close proximity of each other. It is therefore common for several read and write commands to target an already activated row since a typical row size on a DDR2 memory device is 1 KB.

In order not to loose data as a result of the previously described leakage, all the rows in the DRAM have to be refreshed regularly. This is done by issuing a *refresh* command. The average time between refresh commands varies between different memory generations but are the same for all DDR2 devices. The refresh command needs more time on larger devices causing them to spend more time refreshing than smaller ones, although the refresh time per bit gets lower. Before the refresh command is issued all banks have to be precharged. The SDRAM commands discussed are summarized in Table 2.1.

| *SDRAM command* | *Description* |
|---|---|
| No operation (NOP) | Ignores all inputs. |
| Activate (ACT) | Activate a row in a particular bank |
| Read (RD) | Initiate a read burst to an active row |
| Write (WR) | Initiate a write burst to an active row |
| Precharge (PRE) | Close an active row in a particular bank |
| Refresh (REF) | Start a refresh operation |

**Table 2.1:** Some SDRAM commands.

There is a major benefit with a multi-bank architecture since 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 is a process called *bank preparation* that can save a lot of time and sometimes completely hides the precharge and activate delays.

To put the number of banks, rows and columns into perspective we take a brief look into the DDR2 reference specification [11] and see

what a 256 Mb[1] (32Mx8) DDR2-400 SDRAM device looks like. It has a total number of 4 banks with 8192 rows each and 1024 columns per row. This means that 2 bits in the physical address are required for the bank number, 13 for the row number and 10 for the columns. The page size is 1 KB. These devices have a word width of 8 bits but several of them are usually combined to create memories with larger word width. How the chips are combined on a memory module is referred to as the *memory configuration*. For instance, when four of these devices are put in parallel the memory module has a capacity of $256\text{Mb} \cdot 4 = 128\text{MB}$ and a word width of 32 bits. This particular memory configuration, which is the one used throughout this thesis, runs at a clock frequency of 200 MHz, which results in a peak bandwidth of $200 \cdot 2 \cdot 32/8 = 1600$ MB/s.

A command is always issued during one clock cycle but the memories have very tight timing constraints defining the required delay between successive commands. This delay depends on the particular combination of commands. They are found in the specification but we summarize the most important ones in Table 2.2. Throughout this thesis timings with subscripts in capital letters refer to timings from the DDR2 SDRAM Specification [11]. All equations refer to timing values in clock cycles unless otherwise is explicitly noted.

| *Parameter* | *Min. time [ns]* | *Min. time [cycles]* | *Description* |
|---|---|---|---|
| $t_{CK}$ | 5 | 1 | Clock cycle time |
| $t_{RAS}$ | 45 | 9 | Activate to precharge delay |
| $t_{RC}$ | 60 | 12 | Activate to activate delay (same bank) |
| $t_{RCD}$ | 15 | 3 | Activate to read or delay |
| $t_{RFC}$ | 75 | 15 | Refresh to activate delay |
| $t_{RP}$ | 15 | 3 | Precharge to activate delay |
| $t_{RRD}$ | 7.5 | 2 | Activate to activate delay (diff. banks) |
| $t_{REFI}$ | 7800 | 1560 | Average refresh to refresh delay |
| $CL$ | 15 | 3 | CAS (column-access-strobe) latency |
| $t_{WTR}$ | 10 | 2 | Write-to-read turn-around time |
| $t_{WR}$ | 15 | 3 | Write recovery time |

**Table 2.2:** Some timing parameters for a DDR2-400 256 Mb device

[1]Bit is abbreviated as b and byte as B.

We conclude this section by making a formal definition of a memory in Definition 2.3.1

**Definition 2.3.1.** *A memory, $M$, is defined as a nine-tuple ($t_{REFI}(M)$, $t_{RFC}(M)$, $t_{CK}(M)$, $t_{WTR}(M)$, $CL(M)$, $t_{p\_all}(M)$, $s_{burst}(M)$, $s_{word}(M)$, $n_{banks}(M)$) where $t_{p_all}(M)$ is the worst-case time needed to precharge all banks, $s_{burst}(M)$ the burst size programmed in the memory, $s_{word}(M)$ the word width (the width of the data path) and $n_{banks}(M)$ the number of banks. The other timing parameters are explained in Table 2.2 and specified in [11].*

## 2.4 Memory efficiency

Embedded systems of today have high requirements when it comes to memory efficiency. This is natural since inefficient memory use means that faster or wider memories have to be used and these are more expensive and consume more power. In this section we briefly cover memory efficiency, a more complete overview is found in [22]. We start with a definition of memory efficiency.

**Definition 2.4.1.** *Memory efficiency, $e(M)$, is defined as the fraction between the amount of clock cycles when data is transferred, $S'(M)$, and the total number of clock cycles, $S(M)$.*

$$e(M) = \frac{S'(M)}{S(M)}; S'(M) \leq S(M)$$

There are a lot of factors causing data not to be transferred during every cycle. We refer to these as sources of inefficiency. We now look into the most important ones and discuss their impact:

- Refresh efficiency
- Data efficiency
- Bank conflict efficiency
- Read/write efficiency
- Command conflict efficiency

### 2.4.1 Refresh efficiency

As mentioned in Section 2.3 the memory needs to be refreshed regularly. The time needed for refreshing depends on the state of the memory since all the banks have to be precharged before the refresh command is issued. The specification states that this has to be done on average once every $t_{REFI}$, which is 7.8 $\mu s$ for all DDR2 devices. The average refresh interval allows the refresh command to be postponed but not left out. Refresh can be postponed up to a maximum of $9 \cdot t_{REFI}$ when eight successive refresh commands must be issued. Postponing refresh commands is useful when scheduling DRAM commands and helps amortizing the cost of precharging all banks.

Refresh efficiency is relatively easy to quantify since the average refresh interval, clock cycle time, refresh time and worst-case time needed to precharge all banks are derived from the specification of the memory device. Furthermore the refresh efficiency is traffic independent.

The refresh efficiency, $e_{refresh}(M, n)$, of a memory device is calculated as shown in Equation (2.1), where $n$ is the number of consecutive refresh commands. The worst-case time needed to precharge all banks, $t_{p\_all}(M)$, on DDR2-400 is ten cycles. This happens in the event that a bank is activated a cycle before the decision to refresh was taken.

$$e_{\text{refresh}}(M, n) = 1 - \frac{1}{t_{REFI}(M) \cdot n} \cdot (t_{RFC}(M) \cdot n + t_{p\_all}(M)); n \in [1..8] \tag{2.1}$$

For the DDR2-400 described above the effect of refresh efficiency is almost negligible, around 98.4% for a single refresh command and increases as more commands are issued consecutively. The refresh efficiency becomes more significant with larger and faster devices. There is, however, not much to do to reduce the impact of refreshes except trying to schedule them when the memory is idle.

### 2.4.2 Data efficiency

As explained in Section 2.3, the bursts can not start on an arbitrary word since memory is divided into segments of the programmed burst size. This introduces a problem with data alignment. Consider a request for eight words. If those eight words do not start at the boundary

of an eight word segment, two eight word bursts are needed to read or write the data: one for each segment that the data occupies. This is shown in Figure 2.5.

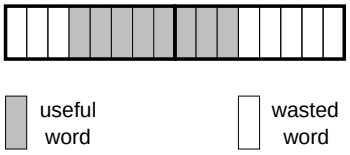


**Figure 2.5:** Half of the data is wasted in these two bursts due to poor data alignment.

The efficiency loss grows with smaller requests and bigger burst sizes. This problem is usually not solved by memory controllers since the minimum burst size is inherent to the memory device and the data alignment is a software issue.

### 2.4.3 Bank conflict efficiency

When a burst targets a column that is not in an open page, the bank has to be precharged and activated to open the requested page. As shown in Table 2.2 there is a minimum delay between consecutive activate commands to a bank resulting in a potentially severe penalty if consecutive read or write commands try to access different pages in the same bank. The impact of this is dependent on the traffic, timings of the target memory and of the memory mapping used.

This problem can be solved by reordering bursts or requests. Intelligent general-purpose memory controllers are fitted with a look-ahead or reorder buffer providing information about bursts that will be serviced in the near future. By looking in the buffer, the controller can detect and possibly prevent bank conflicts through reordering of requests [2, 9, 12, 17, 21] or within requests [10]. This mechanism works well enough to totally hide the extra latency introduced, provided that there are bursts to different banks in the buffer. This solution is very effective but increases latency for the requests. Reordering is not without difficulties. If the bursts within a request are reordered they must be reassembled, which requires extra buffering. If reordering is done between requests then read-after-write, write-after-read and write-after-

read hazards can occur unless dependencies are closely monitored. This requires additional logic.

### 2.4.4 Read/write efficiency

SDRAM suffers from costs when switching directions, i.e going from write to read or read to write. When the bi-directional data bus is being reversed, NOP commands have to be issued resulting in lost cycles. The number of lost cycles depends on the read and write latencies of the target memory, which are specified [11] in terms of the CAS latency, $CL$ of the target memory. The read latency, expressed in Equation (2.2), is the number of cycles from a read command is issued until the first word is returned on the data bus. Write latency, Equation (2.3), is the delay between issuing a write command and putting the first word on the data bus.

$$t_{RL}(M) = CL(M) \tag{2.2}$$

$$t_{WL}(M) = CL(M) - 1 \tag{2.3}$$

We define the switching costs using the terminology and equations from [22].

**Definition 2.4.2.** *$t_{rtw}$ is the number of lost cycles when switching directions from read to write.*

$$t_{rtw}(M) = 2 + t_{WL}(M) - t_{RL}(M) = 2 + (t_{RL}(M) - 1) - t_{RL}(M) = 1$$

Definition 2.4.2 states that one clock cycle is lost when switching from read to write. This is independent of which DDR2 memory that is used since the expression reduces algebraically to a constant value. The write to read time, $t_{wtr}$ is defined in Definition 2.4.3 where $t_{WTR}$ is the write-to-read turn-around-time.

**Definition 2.4.3.** *$t_{wtr}$ is the number of lost cycles when switching directions from write to read.*

$$t_{wtr}(M) = CL + t_{WTR}$$

As shown in Definition 2.4.3 the number of cycles lost when switching from write to read depends on the memory used. For convenience we define the switching cost of a memory in Definition 2.4.4.

**Definition 2.4.4.** *$t_{switch}(M)$ is the number of cycles needed to switch from read to write and back again.*

$$t_{switch}(M) = t_{rtw}(M) + t_{wtr}(M)$$

The read/write efficiency is dependent on the switching frequency, and thus, on traffic. This makes it difficult to estimate the impact on a running application without simulation or measurements during execution. Woltjer [22] states that the theoretical read/write efficiency of a CPU with 70% probability of reads and 30% probability of writes and a request size of 32 words is 93.8%. The worst-case with alternating reads and writes and a request size of four words results in a read/write efficiency of 40%. A solution to this problem is to amortize the cost by preferring reads after reads and writes after writes [2, 9], which again results in higher latency.

### 2.4.5 Command conflict 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. As a result, there may not be enough room on the command bus to issue the activate and precharge commands needed when consecutive read or write bursts are transferred. This results in lost cycles when a read or write burst has to be postponed due to a page fault. With a burst size of eight words, a new read or write command has to be issued every fourth clock cycle leaving the command bus free for other commands 75% of the time. With a burst size of four words read and write commands are issued every second cycle. First generation DDR modules supported a burst size of two. As no other commands can be issued with this burst size, it is impossible to sustain consecutive bursts for a longer period of time.

Read and write commands can be issued with an auto-precharge flag resulting in that the bank is precharged at the earliest possible moment after the transfer is completed. This saves space on the command bus

and is useful when the next burst targets a closed page. Woltjer [22] estimates the command conflict efficiency to 95 - 100%.

## 2.5 Memory controllers

The memory controller is the interface between the system and the memory. A general memory controller consists of four functional blocks:

- Memory mapping
- Arbiter
- Command generator
- Data path

Figure 2.6 shows the structure of a memory controller conforming to the channel buffer model.

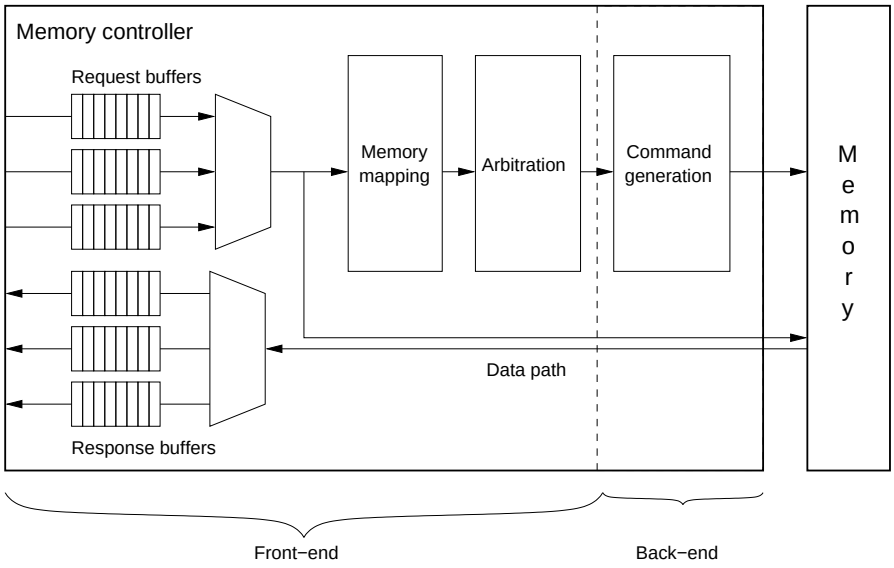


**Figure 2.6:** Layout of memory controller conforming to the channel buffer model.

The memory mapping does the translation from the logical address space used by the requestors to the physical address space (bank, row, column) used by the memory. We address this topic in detail in Section 2.6. The arbiter, or scheduler, decides what request (or burst, depending on the level of granularity) that will next access the memory. This choice can depend on the age of the requests, the amount of traffic that has already been served for that requestor and many other things. Arbitration is more thoroughly covered in Chapter 5. After the arbiter has chosen the request to serve, the actual memory commands need to be generated. The command generator is designed to target a particular memory architecture, such as SDRAM, and is programmed with the timings for a particular memory device, such as DDR2-400. This modularity helps adapting the memory controller for other memories. The command generator needs to keep track of the state of the memory to ensure that no timings are violated. The bi-directional data path is where the actual data is transferred to and from the memory. Our interest is limited to the fact that reversing the direction of this data path, i.e. switching from reads to writes, results in lost cycles. The memory controller is divided into two logical blocks called front-end and back-end. The memory mapping and arbiter is considered a part of the front-end while the command generator is a part of the back-end.

## 2.6 Memory mapping

On a memory access, the logical address, used by the requestor, is decoded into a physical address (bank, row, column) in the memory. This is done by consulting a memory map, which is a function that maps logical to physical addresses. The bank, row and column numbers are determined by closer examining the bits in the address.

It is useful to gain a perspective on what the memory map does by observing what happens for a number of sequential addresses. A brief example follows that uses five bit addresses. The first memory map, observed in Figure 2.7, brings sequential addresses to the same bank. By decoding the two most significant bits into the bank number ensures that iteration is done over columns and rows before switching

bank. A sequential memory map is useful when mapping requestors to particular banks since all traffic in predefined address intervals are guaranteed to hit the same bank. The downside of this mapping is that a large request may hit the end of the page resulting in a page fault.

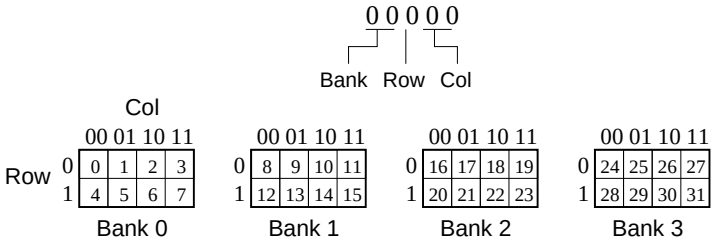


**Figure 2.7:** Sequential memory map.

The memory map in Figure 2.8 interleaves sequential addresses in pairs of two over all four banks. The interleaving memory map has benefits since a large request interleaves over all banks eliminating the risk of page faults all together. The downside of this map is that a minimum burst length is required to hide the latencies of activation and precharging. This particular map is useful when interleaving over banks under the assumption that a burst size of two is enough for a bank to precharge and activate if needed between consecutive accesses.

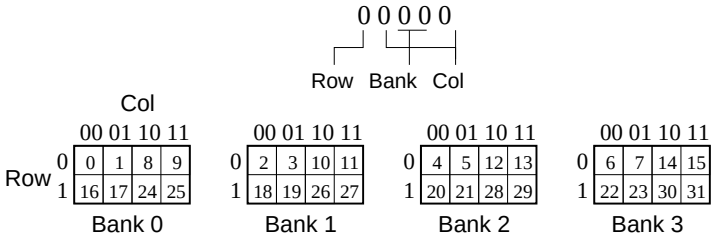


**Figure 2.8:** Interleaved memory map.

In conclusion the memory map is a powerful tool that determines how the requestors access the banks, rows and columns of the memory device. We show in Section 4.4 how this tool is used to provide a hard real-time guarantee on net bandwidth.

## 2.7 Proposed solution

In this section, we present a brief outline of the proposed solution to the problems described in Chapter 1. The proposed model can be viewed as a hybrid combining properties of static and dynamic memory controllers. It is presented as a three-step solution:

- Statically computing an efficient fixed back-end schedule
- Statically computing bandwidth allocation
- Dynamically schedule requestors in run-time

A fixed back-end schedule is computed that makes memory access predictable and provides an efficient gross to net bandwidth translation. The schedule is composed from read, write and refresh groups as shown in Figure 2.9. A read and a write group contains a memory access of maximum burst size for every bank in the memory. These accesses are interleaved over the banks in order to achieve efficient pipelining and thus high memory efficiency. The memory needs to be refreshed at times and thus a refresh group is scheduled after a number of basic groups.

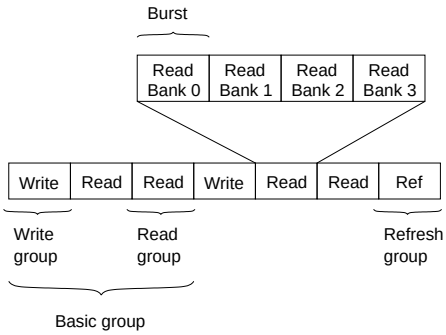


**Figure 2.9:** A back-end schedule composed of read, write and refresh groups.

The back-end schedule yields a good memory efficiency, since some of the sources of inefficiency described in Section 2.4 have been eliminated or bounded. For instance, bank conflicts can not occur by construction since the read and write groups interleave over the banks

providing enough time for bank preparation. Read/write efficiency is addressed by grouping read and write bursts together in the back-end schedule. This puts a bound on the number of switches.

The appropriate back-end schedule has to be computed for a given specification of traffic consisting of minimal net bandwidth requirements and a maximum latency. This requires determining the number and layout of read, write and refresh groups in the back-end schedule. The generated ordering of the groups must offer enough net bandwidth in the read and write directions and for the banks specified by the requestors. The computation of the back-end schedule is explained in Chapter 3.

Step two, allocating bandwidth to requestors is discussed in Chapter 4. This is done in such a way that hard real-time guarantees on net bandwidth and worst-case latency is provided.

Finally, the bursts in the back-end schedule are scheduled to the different requestors in the system taking their bandwidth allocation and quality-of-service requirements into account. This is done dynamically to increase flexibility. The dynamic front-end scheduler can be implemented in several ways to optimize for low worst-case or average latency, high level of fairness or good jitter bounds. It must, however, be sophisticated enough to deliver the guarantees while still being simple enough to be analyzed analytically. This topic is covered in Chapter 5.

Chapter 3

# Back-end schedule

The back-end schedule is the generated sequence of commands sent from the back-end of the memory controller to the memory. Fixing the back-end schedule makes memory access predictable allowing for a deterministic gross to net bandwidth translation. Our back-end schedule is a generalized version of the schedule used in [18, 20], already introduced in Section 2.7. In this chapter we work out a back-end schedule, suitable for a specified set of requestors, composed by a sequence of basic groups, as shown in Figure 2.9. A back-end schedule is formally defined in Definition 3.0.1.

**Definition 3.0.1.** *A back-end schedule, $\theta$, is defined by a three-tuple ($n(\theta)$, $c_{read}(\theta)$, $c_{write}(\theta)$) where $n(\theta)$ is the number of consecutive refresh commands in the refresh group and $c_{read}(\theta)$ and $c_{write}(\theta)$ the number of consecutive read and write groups respectively in the basic group.*

In Section 3.1 we show how to construct the basic building blocks, the read, write and refresh groups. We learn about the flexibility involved when scheduling refreshes in Section 3.2 before discussing schedule layout and the difficulties involved in determining the number of read and write groups in Section 3.3. The efficiency of a back-end schedule is finally defined in Section 3.4.

## 3.1 Basic groups

In order to compose the back-end schedule we must create the low-level building blocks. There are three different kinds of groups: the read

group, the write group and the refresh group. The groups consist of a number of memory commands and looks slightly different depending on the targeted memory generation.

We have chosen to focus our efforts on DDR2 SDRAM [11] to provide high bandwidth. The basic principle applies to previous SDRAM generations, such as SDR SDRAM and DDR SDRAM, although differences in timing requires some design decisions to be re-evaluated. Read and write groups have been manually created for DDR2-400 SDRAM and differs somewhat even for other memories of the same generation due to differences in timing. The groups, illustrated in Figures 3.1, 3.2 and 3.3, consist of a number of consecutive SDRAM-commands familiar from Table 2.1. The only way to make them 100% efficient with consecutive reads and writes potentially targeting different pages is to interleave memory access sequentially over all four banks and use a burst size of eight elements. The larger burst size provides enough time between successive accesses to the same bank to precharge and activate another row. The drawbacks are related to data efficiency. Data that is not aligned on a boundary of eight and requests smaller than the selected burst size results in significant waste. All read and write commands are issued with auto-precharge to make sure that the banks are precharged at the earliest possible moment. This avoids contention on the command bus and makes the groups easier to schedule.

| ACT | NOP | NOP | RD | ACT | NOP | NOP | RD | ACT | NOP | NOP | RD | ACT | NOP | NOP | RD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | | 0 | 1 | | | 1 | 2 | | | 2 | 3 | | | 3 |

**Figure 3.1:** Basic read group

The basic read group is shown in Figure 3.1. We show four pipelined executions to illustrate what a schedule of consecutive reads looks like once the initial start-up has been done. The read group consists of 16 cycles and data is transferred during all of them making the group 100% efficient.

Figure 3.2 shows the basic write group. The group spans 16 cycles and transfers data during all of them, just like the basic read group, and are thus also 100% efficient.

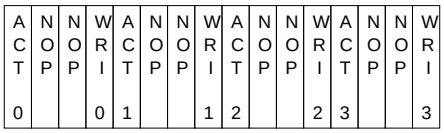


**Figure 3.2:** Basic write group

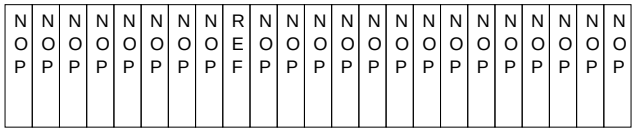


**Figure 3.3:** Basic refresh group

All of the banks must be precharged before a refresh command is issued. The refresh group shown in Figure 3.3 is assumed to follow a read group to more efficiently pipeline the precharging of the banks. Once the refresh command has been issued there is a number of NOP commands during what is called a refresh-to-activate delay ($t_{RFC}$), which have to pass before a new basic group is issued. This particular refresh group is one of several possibilities for a 256 Mb DDR2-400 device, a larger and faster device needs more cycles for refresh.

The back-end schedule is composed by putting these blocks in sequence. As explained in Section 2.4.4, there is a cost associated with switching directions from read to write and vice versa. This has the implication that we have to add two NOP instructions between a read and a write group and four NOP instructions between a write and a read group. This is shown in Figure 3.4.

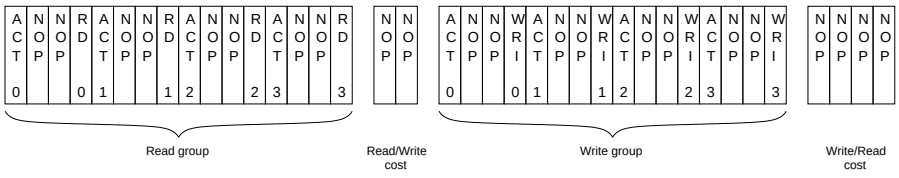


**Figure 3.4:** A read/write switch with extra NOP instructions.

The formal model requires expressions of the group lengths, which

are memory dependent. The read and write groups both require 16 cycles, which corresponds to one burst for every bank in the memory as shown in Equation 3.2.

$$t_{burst}(M) = s_{burst}(M)/2 \tag{3.1}$$

$$t_{group}(M) = t_{burst}(M) \cdot n_{banks}(M) \tag{3.2}$$

The length of the refresh group is discussed in Section 3.2.

## 3.2 Scheduling refreshes

Every row in a DRAM needs to be regularly refreshed in order not to loose data. This has to be taken into account to make the memory accesses predictable, and for this reason a refresh group is created at the end of the schedule.

The refresh group has to start by precharging all banks and then issue between one to eight consecutive refresh commands. If the refresh group succeeds a predefined read or write group the precharging commands of that group can be used to make the refresh group shorter. We have chosen to assume that the refresh group succeeds a read group. This assumption shaves two clock cycles off the refresh group.

**Assumption 3.2.1.** A refresh group always succeeds a read group.

The benefits of postponing refresh is that the overhead involved in precharging all banks is amortized over a large group, as concluded in Section 2.4.1. Postponing refresh is not without disadvantages though, since this makes the refresh group longer. This affects the worst-case latency as a refresh group is always included in the worst-case latency of a request. The number of cycles needed for the refresh group, $t_{ref}(M)$, depending on the number of consecutive refreshes, $n(\theta)$, and timings of the target memory is calculated in Equation (3.3). It considers the time required to precharge all banks, pipelined from a previous read group (8 cycles on DDR2-400), and the time required for the refresh commands as defined in the specification.

$$t_{ref}(M, \theta) = 8 + t_{RFC}(M) \cdot n(\theta); n(\theta) \in [1..8] \tag{3.3}$$

Knowing the refresh group length and the average refresh interval the maximum available number of cycles for read and write groups between two refreshes, $t_{avail}(M, \theta)$ is determined, as shown in Equation (3.4). This effectively determines the length of the back-end schedule.

$$t_{avail}(M, \theta) = n(\theta) \cdot t_{REFI}(M) - t_{ref}(M, \theta); n(\theta) \in [1..8] \quad (3.4)$$

## 3.3 Determining the read/write mix

The back-end schedule is composed of read, write and refresh groups. In this section we focus on determining how many read and write groups there must be and how these should be placed in the schedule. This is a generalization of what is found in [9] where only the equivalent of a single group is allowed before switching direction. This approach works well for older memories but the increasing cost of switching direction has made this generalization necessary.

The number of read and write groups is clearly related to the read and write requirements of the requestors in the system. We have chosen to sum the total bandwidth requested for reads and for writes and let the requested proportions between read and write groups in the schedule be determined by the fraction, $\alpha(R)$, determined by these numbers. This fraction is calculated in Equation (3.5) where $w(r, d)$ is a request function returning the bandwidth requested by requestor $r$ in direction $d$. The special case $\sum_{\forall r \in R} w(r, write) = 0$ is not considered as a read-only SDRAM is useless.

$$\alpha(R) = \frac{\sum_{\forall r \in R} w(r, read)}{\sum_{\forall r \in R} w(r, write)} \quad (3.5)$$

We are looking for the number of consecutive read and write groups, $c_{read}(\theta)$ and $c_{write}(\theta) \in \mathbb{N}$ to approximate this ratio and to constitute the basic group. The chosen values of $c_{read}(\theta)$ and $c_{write}(\theta)$ defines the provided read/write ratio, $\beta(\theta)$.

$$\beta(\theta) = \frac{c_{read}(\theta)}{c_{write}(\theta)} \quad (3.6)$$

The basic group is defined by the write groups followed by the read groups and padded with the extra NOP instructions needed for switching. The basic group is repeated, $k(M, \theta)$ times, until no more can be fitted before refresh. The number of repeated basic groups is calculated in Equation (3.7).

$$k(M, \theta) = \left\lfloor \frac{t_{avail}(M, n(\theta))}{(c_{read}(\theta) + c_{write}(\theta)) \cdot t_{group}(M) + t_{switch}(M)} \right\rfloor \quad (3.7)$$

We generally want to place groups going in the same direction in sequence for efficiency reasons. This saves us from issuing the extra NOP instructions needed to make the groups fit together. In the back-end schedule this heuristic is, however, only valid to a certain extent since large basic groups may not repeat well with respect to refresh due to the non-linearity of Equation (3.7). This means that a large group may be put in sequence a number of times, but leaves a large number of cycles before the end of the average refresh interval unused because it does not fit an additional time. This causes the refresh group to be scheduled prematurely yielding an inefficient schedule although the basic group, as such, is very efficient. This is seen in Equation (3.7) as the value being just slightly smaller than the closest integer value before being floored and the impact of this becomes larger as do $c_{read}(\theta)$ and $c_{write}(\theta)$.

A problem with putting all groups in the same direction in sequence is that the worst-case latency for a request increases significantly since there may be a large amount of bursts scheduled in the opposite direction before scheduling of a particular request can be considered. The maximum latency of the requestors constrains the number of read and write groups that can be put in sequence without violating the guarantees. The worst-case latency for a request depends, as we will show in Section 4.5, on both the number of consecutive read and write groups. It is, for this reason, not possible to solve for a single upper bound on the possible number of groups in a particular direction.

Yet another difficulty arises since many fractions cannot be accurately represented without a very large numerator or denominator. As the latency constrains the number of consecutive groups in a particular direction it is apparent that memory efficiency will, for some read/write

ratios, have to be traded for a lower latency. We look further into this in Section 3.4.

With these difficulties in mind we have decided to use an exhaustive search to find the most efficient solution. This algorithm is the topic of Section 4.6.

## 3.4 Calculating efficiency

The total efficiency of a schedule depends on two components. The first component, schedule efficiency, is due to the regular sources of inefficiency, discussed in Section 2.4, such as read/write switches and refresh resulting in lost cycles. The second component relates to how well the provided read/write mix, $\beta(\theta)$, corresponds to the requested, $\alpha(R)$, introduced in Section 3.3. The first component is, in some regard, significant in all memory controlling schemes but the second one is inherent to this approach. We next look into these one at a time.

First we look into the schedule efficiency. Data is transferred in all the cycles of the read and write groups. The only time when data cannot be transferred is during the refresh cycle and when switching directions. The efficiency of a back-end schedule targeting a specific memory is defined in Definition 3.4.1.

**Definition 3.4.1.** *The schedule efficiency, $e_\theta(M, \theta)$, of a back-end schedule $\theta$ is the fraction between the number of cycles in the back-end schedule with data transfer, $t'_\theta(M, \theta)$, and the total number of cycles needed to revolve the back-end schedule, $t_\theta(M, \theta)$. This is equal to the fraction between the amount of net bandwidth provided by the schedule, $S'(M, \theta)$, and the peak bandwidth of the memory, $S(M)$.*

$$e_\theta(M, \theta) = \frac{S'(M, \theta)}{S(M)} = \frac{t'_\theta(M, \theta)}{t_\theta(M, \theta)}$$

$$t'_\theta(M, \theta) = (c_{read}(\theta) + c_{write}(\theta)) \cdot k(M, \theta) \cdot t_{group}(M)$$

$$t_\theta(M, \theta) = ((c_{read}(\theta) + c_{write}(\theta)) \cdot t_{group}(M) + t_{switch}(M)) \cdot k(M, \theta) + t_{ref}(M, n(\theta))$$

Schedule efficiency is a metric of how well gross bandwidth is translated into net bandwidth. Although this is a relevant number, the total efficiency needs to take into account that the groups in the schedule may not correspond completely to what was requested.

The condition $\alpha(R) \neq \beta(\theta)$ results in an over-allocation for either reads or writes. This is expressed in Definition 3.4.2.

**Definition 3.4.2.** *Mix efficiency, $e_{mix}(R, \theta)$, expresses how well the provided read/write mix correspond to the requested.*

$$e_{mix}(R, \theta) = 1 - |\alpha(R) - \beta(\theta)|$$

Another, perhaps more useful expression of mix efficiency, is shown in Equation (3.8). This relates the average requested read and write groups in a basic group, $\omega(r, d)$, to the provided amount determined by $c_{read}(\theta)$ and $c_{write}(\theta)$. Both expressions are valid metrics to determine how the provided number of groups fit with the requested, but Equation (3.8) causes the mix efficiency to reflect the fraction of the basic group that is useful to the requestors. The provided groups must be larger than the requested for bandwidth allocation to be successful.

$$e'_{mix}(R, \theta) = \frac{\sum_{\forall r in R} \omega(r, \text{read}) + \omega(r, \text{write})}{c_{read}(\theta) + c_{write}(\theta)} \tag{3.8}$$

$$c_{read}(\theta) \geq \omega(r, \text{read})$$
$$c_{write}(\theta) \geq \omega(r, \text{write})$$

We have decided to use total efficiency, $e_{total}(M, \theta, R)$ as defined in Definition 3.4.3, as our metric of efficiency. The mix efficiency corresponds to Definition 3.4.2 since Equation (3.8) was defined too late to be incorporated in this thesis.

**Definition 3.4.3.** *The total efficiency, $e_{total}(M, \theta, R)$, of a back-end schedule, $\theta$ for a particular memory, $M$, and a set of requestors, $R$, is defined as the product between the schedule efficiency and the mix efficiency.*

$$e_{total}(M, \theta, R) = e_{\theta}(M, \theta) \cdot e_{mix}(R, \theta)$$

Figure 3.5 shows the solution space for an example system. The requested read/write ratio is close to 1 and total efficiency grows as the read and write groups are equally increasing until failed latency requirements causes a cut-off. Deviations from the equal number of read and write groups causes efficiency to drop since the read/write mix no longer corresponds to the requested. Bandwidth allocation fails when the deviation from the requested mix becomes too large since the schedule no longer has enough bursts in the right direction to satisfy the bandwidth requirements.

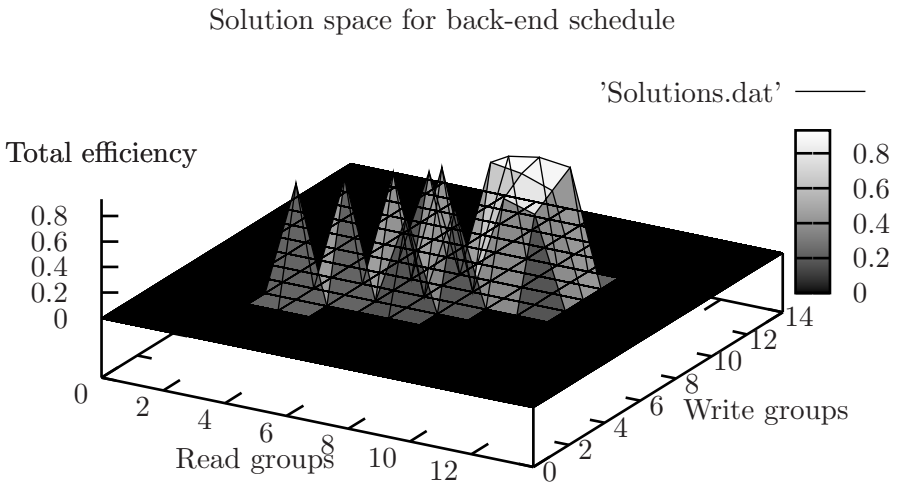


**Figure 3.5:** Solution space for an example system with $n(\theta) = 1$.

Chapter 4

# Bandwidth allocation

In Chapter 3 we calculated a fixed back-end schedule that fits with the specification of the requestors in the system and provides an efficient gross to net bandwidth translation. In this chapter we discuss how to allocate this bandwidth to provide hard real-time guarantees to individual requestors. We begin in Section 4.1 by discussing related work and introducing our bandwidth *allocation scheme*. Bandwidth is allocated to requestors in *service periods*, which are the topic of Section 4.2. In Section 4.3 we show how to compute the number of bursts to allocate to the requestors using an *allocation function*. Section 4.4 discusses how to constrain requestors in order to provide a hard real-time guarantee on net bandwidth. We derive worst-case latency bounds for a request in Section 4.5 before concluding in Section 4.6 by showing how to calculate a *scheduling solution*.

## 4.1 Allocation scheme

The allocation scheme determines how to distribute the bursts in the back-end schedule to guarantee the bandwidth requirements of the requestors in the system. This is done in such a way that it does not put unnecessary constraints on the dynamic front-end scheduling algorithm. This makes the design modular and provides a separation of concerns.

In order to provide a guaranteed service isolation must be provided for a requestor, so that it is protected from the behavior of others. This property, known as requestor *protection*, is important in real-time

systems to prevent a client from over-asking and use resources needed by another. Protection is often accomplished by using a currency, credits, representing how many cycles, bursts or requests that is maximally served before access is granted to another requestor.

Before we explain our choices we look briefly at allocation in related work. Lin *et al.* [12] allocate a programmable number of service cycles in a service period. This means allocating gross bandwidth but they do not disclose enough information to determine whether the bandwidth is guaranteed or not. In [21] a number of requests are allocated in a service period, which translates into a gross bandwidth guarantee as long as the size of the requests is fixed.

We have chosen to allocate a fixed number of bursts in the back-end schedule to every requestor in a service period. This is similar to [21] but with two important differences. Firstly, bursts in the back-end schedule correspond to net bandwidth and secondly the finer level of granularity enables a wider range of dynamic scheduling algorithms. We examine this closer in Chapter 5.

## 4.2 Service periods

As mentioned in Section 4.1, allocation is done on the basis of service periods. We define and discuss service periods in this section.

**Definition 4.2.1.** *A service period, p, has a length of $|p|$ bursts in the back-end schedule. A service period can be shorter than the back-end schedule, and consist of a number of basic groups, or span multiple revolutions of the schedule. All requestors in R have service periods of equal length.*

Definition 4.2.1 constrains how the length of the service period is picked. The back-end schedule is constructed to provide a read/write mix that fits with the requestors in $R$. It is imperative, since a fixed number of bursts are allocated to a requestor in a service period, that every service period maintains the same mix. If the read/write mix changes between service periods, as shown in Figure 4.1, it may not be possible to provide all of the allocated bursts to the requestors causing their bandwidth requirements to fail.

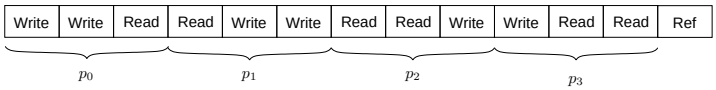


**Figure 4.1:** Four service periods with different read/write mixes.

Defining the service period to span a number of basic groups, as shown in Figure 4.2, prevents the offered read/write mix from changing between service periods. This can also be accomplished by defining the service period to be longer than the back-end schedule and span multiple revolutions. This situation is shown in Figure 4.3.

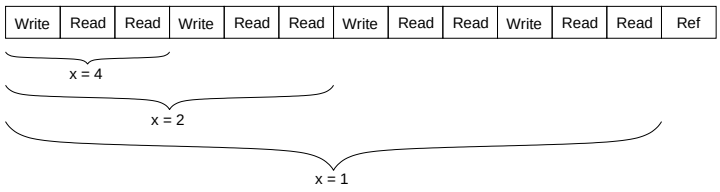


**Figure 4.2:** Possible service periods, shorter than the back-end schedule, that maintains the read/write mix.

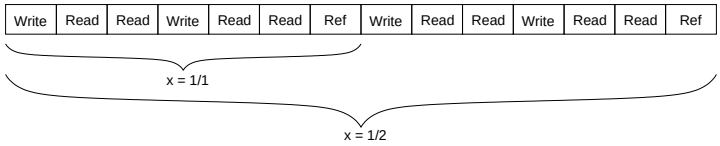


**Figure 4.3:** Possible service periods, longer than the back-end schedule, that maintains the read/write mix.

The formal model requires a relation between the length of the service period and the back-end schedule. Equation (4.1) expresses this relation where $x$ is a function returning the number of times $p$ repeats in one revolution of the back-end schedule. It follows by Definition 4.2.1 that $x(M, \theta, p)$ is a factor of $k(M, \theta)$ if the service period is shorter than the back-end schedule. If the service period spans multiple revolutions of the back-end schedule $x(M, \theta, p) = 1/i; i \in \mathbb{N}$ where $i$ equals the number of revolutions.

$$x(M, \theta, p) = \frac{k(M, \theta) \cdot (c_{read}(\theta) + c_{write}(\theta))}{|p|}; k(M, \theta), |p| \in \mathbb{N} \quad (4.1)$$

The length of the service periods is defined to be equal for all requestors. Varying service period lengths constrains the dynamic front-end scheduler to be schedule-based [23] and schedule memory access on basis of earliest-deadline-first. Greater flexibility is achieved if the scheduling algorithm is free to schedule any requestor whose request fits with the bank and direction of the current burst in the back-end schedule. This decision allows a wider range of scheduling algorithms to be defined, scheduling requestors to achieve anything from high throughput to low average latency or a high level of fairness. Properties of scheduling algorithms are further discussed in Section 5.1.

## 4.3 Allocation function

In [12, 21] the number of cycles and requests per service period is manually determined and programmed at device setup. We have chosen to automate this step by having an allocation function, $a(r, M, \theta, p)$, compute the allocation for the requestors.

**Definition 4.3.1.** *A requestor, $r$, is allocated at least $a(r, M, \theta, p)$ bursts out of every $|p|$ bursts. This corresponds to allocating an amount of net bandwidth, $A(r, M, \theta, p)$, that is a fraction of the available net bandwidth $S'(M, \theta)$. The bandwidth requirement of $r$ is satisfied if $A(r, M, \theta, p) \geq w(r)$.*

$$A(r, M, \theta, p) = \frac{a(r, M, \theta, p)}{|p|} \cdot S'(M, \theta)$$

For the allocated rates to make any sense we cannot allocate more bandwidth to the set of requestors than available, meaning that Equation (4.2) must hold.

$$\sum_{\forall r \in R} \frac{a(r, M, \theta, p)}{|p|} \leq 1 \quad (4.2)$$

The allocation function is simple and calculates the number of bursts required per service period. This is based on the average bandwidth requirement in the specification. The number of requested bursts in a service period needs to be calculated. To do this we need to determine the number of revolutions of the back-end schedule per second. This is done as shown in Equation (4.3), by calculating the number of available clock cycles in a second and dividing it by the number of cycles, $t_\theta(M, \theta)$, needed to revolve the back-end schedule once. The clock cycle time, $t_{CLK}(M)$, must be specified in seconds for Equation (4.3) to be accurate.

$$n_\theta(M, \theta) = \frac{\frac{1}{t_{CLK}(M)}}{t_\theta(M, \theta)} \tag{4.3}$$

We now translate the bandwidth requirement per second into a requirement per service period. Equation (4.4) shows how to calculate this burst requirement. We refer to this as the *real requirement*.

$$w_{real}(r, M, \theta, p) = \frac{\frac{w(r)}{s_{burst}(M) \cdot s_{word}(M)}}{n_\theta(M, \theta) \cdot x(M, \theta, p)}; w_{real}(r, M) \in \mathbb{R}^+ \tag{4.4}$$

The number of bursts allocated to the requestor, the *actual requirement*, must be a multiple of the request size of the requestor. This results in that a requestor is always capable of finishing the service of a request in one service period, which is good for the worst-case latency bound. This also increases the effect of discretization errors during allocation thus reducing memory efficiency. The actual requirement for a requestor is computed in Equation (4.6) where $\sigma_{burst}(r, M)$ is the request size in memory bursts. Note that $a(r, M, \theta, p) \geq w_{real}(r, M, \theta, p) \Leftrightarrow A(r, M, \theta, p) \geq w(r)$, which is required to satisfy the bandwidth requirement.

$$\sigma_{burst}(r, M) = \left\lceil \frac{\sigma(r)}{s_{burst}(M) \cdot s_{word}(M)} \right\rceil \tag{4.5}$$

$$a(r, M, \theta, p) = \left\lceil \frac{w_{real}(r, M, \theta, p)}{\sigma_{burst}(r)} \right\rceil \cdot \sigma_{burst}(r) \tag{4.6}$$

The over-allocation, $o(R, r, M, \theta, p)$, introduced by the discretization errors in Equation (4.6) is quantified in Equation (4.7).

$$o(R, r, M, \theta, p) = \frac{\sum_{\forall r \in R} a(r, M, \theta, p) - w_{real}(r, M, \theta, p)}{\sum_{\forall r \in R} a(r, M, \theta, p)} \tag{4.7}$$

The worst-case over-allocation, presented in Equation (4.8), occurs when $a(r, M, \theta, p) - w_{real}(r, M, \theta, p) = \sigma_{burst}(r) - \epsilon$ for all requestors where $\epsilon$ is an infinitesimal number. In this case every requestor is over-allocated one request per service period. The worst-case over-allocation grows with an increasing number of requestor, larger request size and shorter service periods. This is further examined in Section 6.4

$$o_{wc}(R, r, M, \theta, p) = \frac{\sum_{\forall r \in R} \sigma_{burst}(r)}{\sum_{\forall r \in R} w_{real}(r, M, \theta, p)} \tag{4.8}$$

We define allocation efficiency, $e_{alloc}(r, M, \theta, p)$, in Definition 4.3.2 as a metric of the efficiency of an allocation.

**Definition 4.3.2.** *Allocation efficiency, $e_{alloc}(r, M, \theta, p)$, is a measure of how well a particular allocation fits with the requested number of bursts. Allocation efficiency is defined as the ratio between the actual and the real number of requested bursts.*

$$e_{alloc}(r, M, \theta, p) = \frac{\sum_{\forall r \in R} w_{real}(r, M, \theta, p)}{\sum_{\forall r \in R} w_{real}(r, M, \theta, p)}$$

Total efficiency (Definition 3.4.3) is a metric of how much net bandwidth a schedule provides and how well it fits with the directions of the requestors. By including the allocation efficiency in this metric we state that we also want the requested bandwidth to fit as close as possible with the allocated bandwidth. This means that an efficient solution has more slack bandwidth available in a service period than it otherwise would, as shown in Figure 4.4. The slack bandwidth is not useful when adding more requestors since both the back-end schedule and the allocation must be recomputed. The allocation efficiency will also, if aggregated into the total efficiency metric, interact with the schedule and mix efficiency and promote solutions with less net bandwidth and slack bandwidth in non useful direction. We have thus decided not to include allocation efficiency in our total efficiency metric.

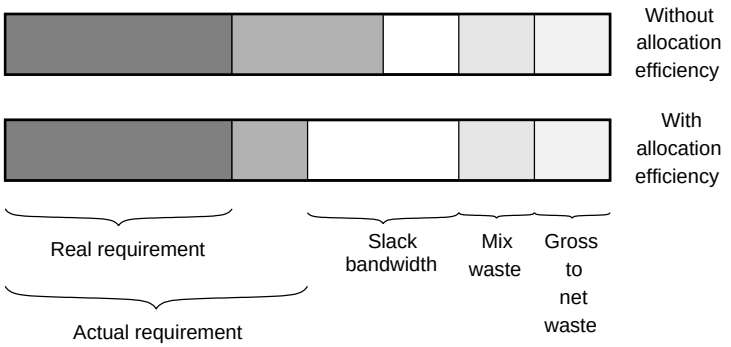


**Figure 4.4:** Preferred solutions with and without allocation efficiency in total efficiency metric. Blocks shown as fractions of peak bandwidth.

## 4.4 Requestor constraints

To provide a hard real-time net bandwidth guarantee, every requestor must be able to use its entire allocation every scheduling period. In this section we introduce constraints on the requestors to ensure that allocated bandwidth is not wasted.

**Definition 4.4.1.** *A requestor, r inR, is backlogged if the request queue for r is not empty. The requestors in R must be backlogged all the time for their bandwidth guarantee to be valid.*

Definition 4.4.1 is intuitive since bandwidth can not be delivered unless it is requested. Backlogging is required by most scheduling algorithms for their properties to be valid.

Transaction boundaries are another source of wasted allocated bandwidth. They can result in waste if a requestor changes directions, as shown in Figure 4.5.

A single requestor is allocated all of the bursts. A read request, interleaved over the banks by the memory map, is present in the request buffer, followed by a write request. The figure shows that once the read burst is finished a number of bursts cannot be used since they are in the wrong direction. The waste is less noticeable in systems with multiple requestors since one requestor can cover for the transaction boundaries of another by making use of the intermediate bursts. This potential waste of bandwidth is, however, not acceptable in a system that delivers

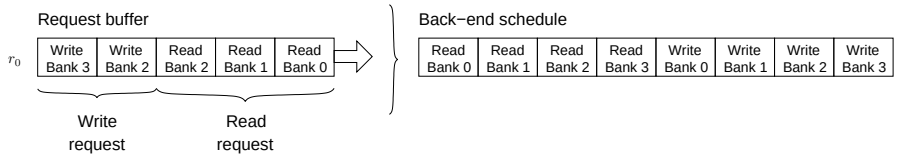


**Figure 4.5:** Bursts wasted when changing directions.

hard real-time guarantees, which is why Definition 2.2.2 states that a requestor is either reading or writing, but not both. Consider further the situation with an interleaved memory map depicted in Figure 4.6.

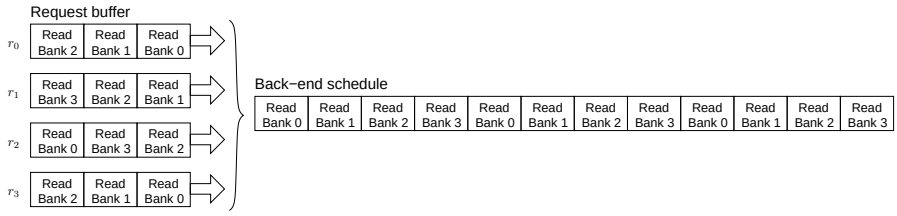


**Figure 4.6:** Arbitrary request patterns may result in wasted bursts.

The requestors do not have more bursts going in any direction than what is available, but still an arbitrary scheduling order may result in waste. If $r_0$ gets the first three bursts, then no one can make use of the fourth burst, which is wasted even though no requestor misbehaved. This may result in some requestor failing to meet its guarantees. It is very difficult to solve this problem through scheduling and this would seriously constrain the scheduler and hurt the flexibility of the model. The problem is relieved by making sure that it is known beforehand what bank that will be requested by the requestors. That can be done using one of two *bank access patterns*: partitioning and memory-aware IP design. Both approaches can be used to create a hard real-time guarantee on net bandwidth.

The memory map of a partitioned system maps the address space of a requestor to a single bank. All requests from a requestor will thus always target the same bank. A partitioned system guarantees that

a request can be scheduled and that a slot is only wasted if Definition 4.4.1 is violated. Partitioning requestors to particular memory banks is, however, a challenging problem. First of all nothing guarantees that a partitioning can be made that is balanced on both bandwidth and directions. Any unbalance in the partitioning results in wasted bandwidth and may cause allocation to fail. Secondly partitioning complicates reconfiguration. The partitioning has to be redone if requestors are added or if their requirements are modified and it may no longer be possible to find acceptable solutions. If a new solution can be found, some requestors may have been mapped to other banks and have to move their data before new memory accesses are allowed.

With memory-aware IP design we mean a system that is designed with the multi-bank architecture of the target memory in mind. This may involve making every memory access request all banks in sequence and thus have a system that is perfectly balanced over the banks by construction. This pattern avoids the balancing problem, but constrains the size of the requests to be a multiple of $s_{burst}(M) \cdot s_{word}(M)$ B.

## 4.5 Analysis of worst-case latency

In a hard real-time system the worst-case performance is of utmost importance and must be well-known if guarantees are to be provided. In this section we discuss how to compute the worst-case latency for a request.

We use a modular approach and calculate the worst-case latency as the sum of a number of components. We briefly list the components of the worst-case service latency and then discuss them in greater detail.

- Bursts needed in the direction of the request before it is finished
- Read/write switches and bursts going in the interfering direction.
- Interfering refresh groups
- Arrival/arbitration mismatch

We choose to keep the analysis general so that it is valid for all scheduling algorithms implementing the allocation scheme. A tighter

bound can be derived by examining a particular scheduling algorithm. We do not tailor the analysis to a particular quality-of-service scheme but require the existence of a total ordering between the priority levels used.

We assume that a request arrives at such a time that the interference from the other groups is maximized. The worst-case arrival for a request is to end up just in front of the last sequence of bursts going in the interfering direction. This does not only make sure that the maximum number of unusable bursts is up for scheduling, but also that every request has at least one refresh included in the worst-case latency. The worst-case positions in the back-end schedule for reads and writes are illustrated in Figure 4.7.

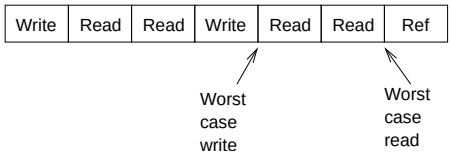


**Figure 4.7:** Initial position in schedule for worst-case read and worst-case write.

### 4.5.1 Remaining bursts

We need to calculate how many bursts in the direction of the requestor that are needed to guarantee that the request finishes. The request needs $\sigma_{burst}(r, M)$ bursts in the proper direction to finish. Since we do not make any assumptions about the scheduling order of the front-end scheduler these are assumed to be as late as possible. At this stage priorities come into play. A requestor can be forced to wait for all other requestors of equal or higher priority in the same direction. We do thus define the set $R'(R, r)$ to contain all such requestors.

**Definition 4.5.1.** *The set $R'(R, r)$ is defined to contain all requestors with the same direction as $r$ and with equal or higher priority.*

The request is thus finished after the number of bursts in the right direction computed by Equations (4.10) and (4.9). The equations calculates the combined allocation of all requestors in $R'(R, r)$ except for

$a(r, M, \theta, p)$ since only $\sigma_{burst}(r, M) \leq a(r, M, \theta, p)$ bursts are required by $r$ for the request to be finished. Equation (4.9) applies to systems using the memory-aware bank access pattern. Note that the computed value must be multiplied by the number of banks if the requestors are partitioned to specific banks, as shown in Equation (4.10), since only one out of $n_{banks}(M)$ bursts are useful to serve the request.

$$n_{left}^{aware}(R, r, M, \theta, p) = \sum_{\forall \rho \in R'(R,r)} a(\rho, M, \theta, p) - a(r, M, \theta, p) + \sigma_{burst}(r, M) \tag{4.9}$$

$$n_{left}^{partitioned}(R, r, M, \theta, p) = n_{left}^{aware}(R, r, M, \theta, p) \cdot n_{banks}(M) \tag{4.10}$$

### 4.5.2 Interfering bursts

The total number of bursts to wait for in order to get $n_{left}(R, r, M, \theta, p)$ bursts in the right direction may vary for reads and writes since the number of consecutive bursts, $c_{read}(\theta)$ and $c_{write}(\theta)$, can be different. Equations (4.11) and (4.12) calculates the time lost to bursts in the interfering direction for a read request, including the actual switches. Equations (4.13) and (4.14) applies to write requests.

$$n_{switches}^{read}(R, r, M, \theta, p) = \left\lceil \frac{n_{left}(R, r, M, \theta, p)}{c_{read}(\theta)} \right\rceil \tag{4.11}$$

$$t_{direction}^{read}(R, r, M, \theta, p) = n_{switches}(R, r, M, \theta, p) \cdot (t_{switch}(M) + c_{write}(\theta) \cdot t_{group}(M)) \tag{4.12}$$

$$n_{switches}^{write}(R, r, M, \theta, p) = \left\lceil \frac{n_{left}(R, r, M, \theta, p)}{c_{write}(\theta)} \right\rceil \tag{4.13}$$

$$t_{direction}^{write}(R, r, M, \theta, p) = n_{switches}(R, r, M, \theta, p) \cdot (t_{switch}(M) + c_{read}(\theta) \cdot t_{group}(M)) \tag{4.14}$$

### 4.5.3 Refresh interference

As previously stated the worst-case latency always contains at least one refresh group. For every revolution of the back-end schedule there is an additional refresh group. The number of interfering refresh groups are conveniently expressed in terms of the relation between the back-end schedule and the service period, provided by the function $x$, due to the constraints put on the service period in Definition 4.2.1. The total number of refreshes interfering with the transaction is calculated in Equation (4.15).

$$n_{ref}(M, \theta, p) = \left\lceil \frac{1}{x(M, \theta, p)} \right\rceil \tag{4.15}$$

### 4.5.4 Arrival/arbitration mismatch

The cycles until re-arbitration are lost if a request becomes eligible just after an arbitration decision is made. The number of cycles lost is one cycle less than the arbitration period. Re-arbitration occurs after every burst in the partitioned system but only after one burst is scheduled for every bank in the memory-aware system. This means that more cycles are lost in systems with the memory-aware bank access pattern, shown in Equation (4.16), than in partitioned systems (Equation (4.17)).

$$t^{aware}_{mismatch}(r, M) = \frac{s_{burst}(r, M)}{2} \cdot n_{banks}(M) - 1 \tag{4.16}$$

$$t^{partitioned}_{mismatch}(r, M) = \frac{s_{burst}(r, M)}{2} - 1 \tag{4.17}$$

### 4.5.5 Putting it together

We calculate the worst-case latency by combining the equations in this section. This is shown in Equation (4.18).

$$\begin{aligned} t_{lat}(R, r, M, \theta, p) = {} & n_{left}(R, r, M, \theta, p) \cdot t_{burst}(M) + \\ & t_{direction}(R, r, M, \theta, p) + n_{ref}(M, \theta, p) \cdot t_{ref}(M) + \\ & t_{mismatch}(r, M) \end{aligned} \tag{4.18}$$

This section shows that there are many factors affecting the worst-case latency for a request from a particular requestor. It depends on the requestor itself and all other requestors in the system. It also depends on the target memory, the back-end schedule and the scheduling period that is used.

Equation (4.18) reveals a number of ways to reduce the worst-case latency. We can reduce $n_{left}(R, r, M, \theta, p)$ by granting a latency sensitive requestor higher priority. This reduces the set $R'(R, r)$ and thus the impact of other requestors in the system. This has a significant impact, especially in partitioned systems with multiple requestors mapped to the same bank. The remaining number of bursts also shrinks with the request size since fewer bursts needed to transfer smaller requests. Groups in the interfering direction hurts worst-case latency. The impact is reduced by constraining the number of groups in that direction, which trades latency for memory efficiency and may cause allocation to fail for high loads. Refresh interference is not a major component in the worst-case latency but is minimized by keeping the service period shorter than the back-end schedule.

## 4.6 Computing a scheduling solution

In this section we show how to compute a scheduling solution, which consists of a back-end schedule and its relation to the service period. Scheduling solutions are defined in Definition 4.6.1.

**Definition 4.6.1.** *For a target memory a scheduling solution, $\gamma$, is defined by a tuple ($\theta$, $x(M, \theta, p)$), where $\theta$ is a back-end schedule and $x(M, \theta, p)$ the relation to the service period.*

As stated in Section 3.3 the nonlinear properties of the back-end schedule makes it difficult to directly compute an optimal solution. We have chosen to use an exhaustive search within a reduced search space to find the schedule. Since the algorithm computes the schedules for the different use-cases off-line, it has no real-time demands making an exhaustive search a feasible option. The search space is, however, bounded to make the run-time of the algorithm faster.

The algorithm consists of four nested loops iterating over the number of consecutive refreshes, read groups, write groups and the possible

service periods ($n(\theta)$, $c_{read}(\theta)$, $c_{write}(\theta)$ and $x(M, \theta, p)$ respectively). The refresh loop is bounded by the number of refresh commands in the refresh group, maximum eight for all DDR memories. The number of read and write groups are bounded by the total number of groups that fit before refresh. The number of groups that fit before refresh depends on the number of switches but corresponds roughly to 100 groups per refresh command in the refresh group. The possible service periods are bounded by the number of unique factors in $k(M, \theta)$ for every solution. The search space is limited by not adding further groups in one direction if there is a latency violation in the other unless more groups are added in that direction as well. If both read and write latencies are violated by a solution, then no better valid solution can be found with the present refresh settings. This means that the algorithm uses the latency calculations to limit the search space, compensating for the rather loose bounds on the number of read and write groups. The algorithm picks the optimal solution from the set of valid solutions. The *optimization criterion* can vary from memory efficiency, or lowest average latency to most efficient allocation.

**Algorithm 4.1** Algorithm for computing a scheduling solution.

```
for possible refresh intervals (1 to 8 consecutive refreshes)
    for 1 to READ_MAX read groups
        for 1 to WRITE_MAX write groups
            for possible service periods (unique factors in k)
                if allocation successful and
                   latency constraints satisfied
                    store solution
                else if read latency violation
                    break write loop
                else if read and write latency violation
                    break read loop
pick optimal solution
```

Chapter 5

# Dynamic front-end scheduler

The allocation scheme guarantees that a number of bursts, determined by an allocation function, is serviced to the requestors every service period. A dynamic front-end scheduler is introduced that bridges between the fixed back-end schedule and the allocation scheme by mapping bursts to requestors. Flexibility is increased by distributing the allocated bursts dynamically according to the quality-of-service levels of the requestors. We start by looking into some important properties of scheduling algorithms in Section 5.1 before introducing our scheduling algorithm, sliding QoS-aware FCFS scheduling, in Section 5.2.

## 5.1 Properties and terminology

The original work on real-time scheduling concerns job scheduling on a single processor but have been further developed in the context of packet-switched networks in the 90's. A large number of algorithms with different properties has been proposed over the years and in this section we learn about these properties and familiarize ourselves with the terminology of scheduling algorithms. We look into five general properties of scheduling algorithms: work conservation, fairness, protection, flexibility and simplicity.

### 5.1.1 Work-conservation

A scheduling algorithm can be classified as work-conserving or non-work-conserving. A work-conserving algorithm is never idle when there

is something to schedule. In a non-work-conserving environment requests get associated with an eligibility time and are not scheduled until this time, even though the memory may be idle. It is easy to realize that a work-conserving algorithm can yield a lower average latency than a non-work-conserving since it can achieve higher average throughput. The advantage of non-work-conserving scheduling algorithms is that they can reduce buffering by providing data just-in-time and put bounds on jitter. A number of work-conserving and non-work-conserving scheduling algorithms are overviewed in [23, 24]

### 5.1.2 Fairness

A fair scheduling algorithm is expected to serve the requestors in a balanced fashion according to their allocation. Perfect fairness is formally expressed in Equation 5.1 with $s_r(t_0, t_1)$ denoting the amount of service given to requestor $r$ in the half-open time interval $[t_0, t_1)$.

$$\left| \frac{s_k(t_0, t_1)}{a_k} - \frac{s_j(t_0, t_1)}{a_j} \right| = 0; \forall t_0, t_1, \forall k, j \in R \tag{5.1}$$

It follows from Equation 5.1 that a perfectly fair system can only be achieved in a system where work is infinitely divisible, a fluid system. A scheduling algorithm for this kind of system is proposed in [15]. The more general expression in Equation 5.2 is used if the system in question is not a fluid system. Several scheduling algorithms [3, 4, 15, 19] have been proposed that work with this kind of fairness bounds.

$$\left| \frac{s_k(t_0, t_1)}{a(k, M, \theta, p)} - \frac{s_j(t_0, t_1)}{a(j, M, \theta, p)} \right| < \kappa; \forall t_0, t_1, \forall k, j \in R \tag{5.2}$$

It should be clear from Equation 5.2 that the bound on fairness, $\kappa$, grows with the level of granularity in the system. It is thus possible to create an algorithm with higher degree of fairness in a system scheduling SDRAM bursts rather than requests since this is a closer approximation of a fluid system, which is why we chose to work on a finer level of granularity in Section 4.1.

Fairness impacts buffering. The channel buffers bridge between the arrival and the consumption processes. The consumption process is determined by the memory controller but the arrival process is assumed,

from Section 2.2, to be unknown. For this reason these processes must be assumed to have a maximum phase mismatch. A high level of fairness makes the consumption process less bursty, causing the buffers to drain more evenly. This brings the worst-case and average-case buffering closer together.

Fairness has a dualistic impact on latency. When interleaving requests of the same size, the worst-case latency remains the same but the average latency increases since the requests finishes later. The impact of this grows with finer granularity. If requests have different sizes, fairness prevents a small request from being blocked by a large one and receive high latency and an unreasonable wait/service ratio.

Our design has a two level fairness scheme. The allocation scheme provides fairness in the sense that the requestors get their allocated number of bursts in a service period. The smaller the period the larger the level of fairness. The second level of fairness is optionally provided by the scheduling algorithm.

### 5.1.3 Protection

It has been observed in packet-switched networks employing a FCFS algorithm that a host can claim an arbitrary percentage of the bandwidth by increasing its transmission rate. This enables malfunctioning or malicious hosts to affect the service given to well-behaving hosts. Nagle [13] addresses this problem by using multiple output queues and servicing them in a Round-Robin fashion. This provides isolation and protects a host from the behavior of others.

Protection is fundamental in a system providing guaranteed services and for that reason this property is built into the allocation scheme, as discussed in Section 4.1, and is provided regardless of the scheduling algorithm in use. Over-asking results in buffers filling up, which can cause data loss in a lossy system or flow control to halt the producer in a lossless one. Either way the service of the other requestors is not disrupted.

### 5.1.4 Flexibility

A scheduling algorithm must be flexible and cater to diverse traffic characteristics and performance requirements, as discussed in section 1.1.

These kinds of traffic and their requirements are well recognized. Many memory controllers deal with these differing demands by introducing traffic classes. Although the memory controllers are quite different the chosen traffic classes are very similar since they correspond to well-known traffic types. Three common traffic classes are identified:

- Low latency
- High bandwidth
- Best effort

The low latency traffic class targets requestors that are very latency sensitive. In most memory controllers the requestors in this class have the highest priority, at least as long as they stay within their allocation [12, 14, 19, 21]. In their attempts to minimize latency Lin *et al.* [12] enables requests within this traffic class to pre-empt other requests of lower priority. This reduces latency at the expense of memory efficiency and predictability.

The high bandwidth class is used for streaming requestors. In some systems these have loose bounds on latency allowing the requests in this traffic class to be reordered and thus sacrifice latency in favor of memory efficiency.

The best effort traffic class is found in [12, 19, 21] and these requests have the lowest priority in the system. They have no guaranteed bandwidth nor bounds on latency but are served whenever there is bandwidth left over from the higher priority requestors. It is important to keep in mind that if the left over bandwidth is lower, on average, than the requested rate from the requestors in this traffic class, requests will have to be dropped to prevent overflowing the request buffer or flow-control must be used to slow down the requestors.

### 5.1.5 Simplicity

There are limitations on the complexity of the scheduling. It must be feasible to implement in hardware and run at high speeds. The time available for arbitration depends on the size of the service unit used. We have chosen to work with bursts in the back-end schedule, which are

DDR bursts of eight words. This means that re-arbitration is needed every four clock cycles, corresponding to 20 ns for DDR2-400 and 12 ns for DDR2-667. This provides a lower bound on the speed of the arbiter.

## 5.2 Sliding QoS-aware FCFS scheduling

An objective of the bandwidth allocation scheme presented in Chapter 4 is to place as few constraints as possible on the scheduling algorithm. The allocation scheme states that the algorithm used must provide an allocated number of bursts to every requestor in a service period. No assumptions are made regarding the order in which the requestors get their allocated number of bursts. This provides great flexibility since the scheduling algorithm is free to pick any requestor whose bank and direction matches the current burst in the back-end schedule.

In this section we present our scheduling algorithm, called sliding QoS-aware FCFS scheduling. The term sliding stems from that the service periods may be unaligned and that a new period does not start unless there is a request available in the request buffer of the requestor. Figure 5.1 shows a number of aligned consecutive service periods and Figure 5.2 a situation with sliding service periods. A black slot indicates that there are no requests in the request buffer causing the periods to slide.

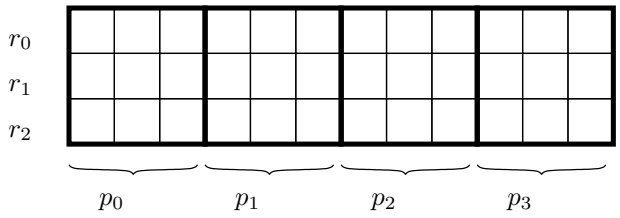


**Figure 5.1:** The service periods are aligned.

The benefits of the sliding service periods is that a requestor that has been idle benefits from bandwidth guarantees right away and does not have to wait for the service period to restart, a delay that is po-

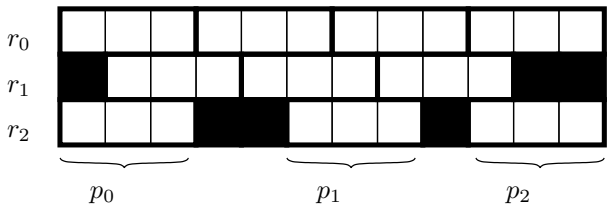


**Figure 5.2:** The service periods are unaligned and sliding. Service periods shown for $r_2$.

tentially very long depending on the granularity of the scheduling and the priority level of the requestor.

Sliding the service periods does not complicate the bandwidth guarantee. All requestors can still use their entire allocation regardless of the behavior of others as stated in Theorem 5.2.1.

**Theorem 5.2.1.** *A requestor, $r$, is guaranteed $a(r, M, \theta, p)$ bursts in a sliding service period.*

*Proof.* The proof is expressed in terms of delay. Consider the set of requestors, $R^-(R, r)$, consisting of all requestors but one, $r$. When the requestors are all backlogged and aligned like in Figure 5.3 the delay, $\delta(R, r)$, imposed on $r$ by $R^-(R, r)$ reaches its maximum value. This is realized by considering what happens in terms of delay for $r$ as the windows start to slide. This is shown in Figure 5.4. The maximum delay is bounded by the total allocation of the requestors in $R^-(R, r)$. This is stated in Equation (5.3)

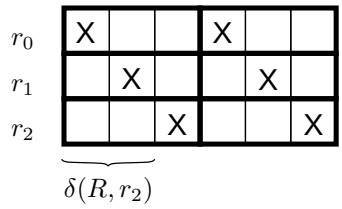


**Figure 5.3:** Worst-case for $r_2$. Maximally delayed by $r_0$ and $r_1$. $a(r, M, \theta, p) = 3; \forall r \in R$ and $|p| = 3$.

No matter in what direction or over what distance $r_0$ or $r_1$ slides, $\delta(R, r_2)$ decreases indicating that the case in Figure 5.3 was indeed the

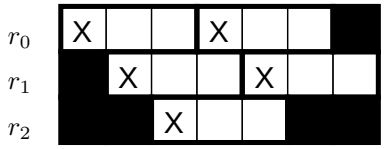


**Figure 5.4:** As the windows slide $r_2$ gets less delayed. In this figure $\delta(R, r_2) = 0$.

worst-case.

$$\delta(R, r) \leq \sum_{\forall \rho \in R^-(R,r)} a(r, M, \theta, p) \tag{5.3}$$

For the bandwidth guarantee to hold we must show that there is enough space in $p$ for $r$ to allocate $a(r, M, \theta, p)$ bursts even with the maximum delay, hence Equation (5.4) must hold.

$$\delta(R, r) \leq \sum_{\forall \rho \in R^-(R,r)} a(\rho, M, \theta, p) \leq |p| - a(r, M, \theta, p) \tag{5.4}$$

Equation (5.4) is, according to Equation (4.2) together with Definition 4.2.1 guaranteed to hold since

$$\sum_{\forall \rho \in R} a(\rho, M, \theta, p) \leq |p|$$

$$0 \leq |p| - \sum_{\forall \rho \in R} a(r, M, \theta, p) = |p| - \left( \sum_{\forall \rho \in R^-(R,r)} a(\rho, M, \theta, p) \right) - a(r, M, \theta, p)$$

$$\sum_{\forall \rho \in R} a(\rho, M, \theta, p) \leq |p| - a(r, M, \theta, p)$$

□

This shows that there are always enough bursts within $p$ to schedule $a(r, M, \theta, p)$ of them to $r$.

Our front-end scheduler is similar to that of the Deficit Round-Robin (DRR) scheduling algorithm. Two variations of this algorithm are introduced in [19] and our implementation is inspired by one of them, called DRR+. DRR+ is designed as a fast packet-switching algorithm with a high level of fairness. It operates on the level of packets with variable size, which is very similar to the requests considered in our model, and can easily be modified to work with bursts. We primarily employ two priority levels, low latency and high-bandwidth, from the well-known classes described in Section 5.1. One way to improve the average latency is to make the algorithm work-conserving and let requestors who have run out of credits degrade to the best effort priority level and be served only if no other backlogged and eligible requestor has remaining credits. We look into this option in Section 6.7.

**Definition 5.2.2.** *The set of valid priority levels is defined as* $C = \{LL, HB, BE\}$.

Since the back-end schedule has decided on the bank and direction of a particular burst only requests going in that direction can be considered and do thus constitute a subset of the requestors eligible for scheduling. Lists, similar to the active-lists of DRR+, are maintained in FCFS order for every quality-of-service level. A previously idle requestor is added to the corresponding list when a request arrives at an empty request buffer. These lists are maintained in one of two ways depending on which of two variations of the algorithm that is being used. The first variation does scheduling on the request level and does not pick another request from the eligible subset until the entire request is finished. The requestor is added to the bottom of the list when a request is finished, provided that there are more requests in the request queue. The second variation of the algorithm operates on the burst level and moves the requestor to the bottom of the list for every scheduled burst. Both variants allow only a single entry in the lists per requestor.

The first variation reduces the amount of interleaving and provides a lower average latency, although the worst-case latency remains the same. The amount of buffering required is proportional to the burstiness of the arrival and consumption processes and the worst-case latency. The arrival process and worst-case latency is unchanged for the

two variations but the first variation has more bursty consumption and has thus a larger worst-case buffer requirement.

The lists are examined in a FCFS order and the first eligible requestor is scheduled. To give low latency requestors the quality-of-service they require they are always served first. If there are no backlogged low latency requestors, or if they have run out of allocation credits, a high bandwidth requestor is picked. This situation is illustrated in Figure 5.5.

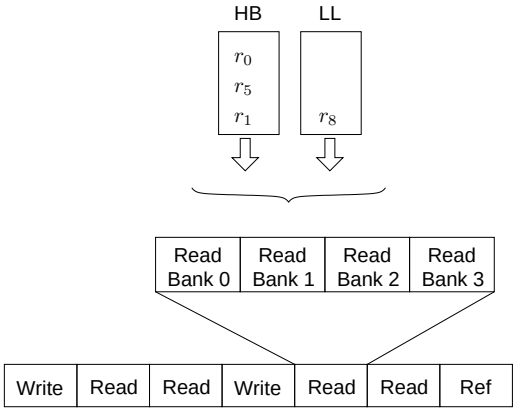


**Figure 5.5:** The scheduler picks from a set of eligible low-latency and high-bandwidth requestors.

The FCFS nature of the algorithms increases fairness beyond that of the allocation scheme, which means that tighter latency bounds than those of Section 4.5 can be derived. This analysis is, however, not included in this thesis.

# Chapter 6
# Experiments

We derived an analytical memory controller model in previous chapters. In this chapter we simulate an example application and compare analytical and simulated results. In Section 6.1 we present the simulation platform and the conditions of the simulation. The example application used in our simulations is introduced in Section 6.2 before scheduling solutions are computed for the partitioned and memory-aware access patterns in Section 6.3. We discuss over-allocation and the impact of request sizes in Section 6.4 as we present the allocation results and then proceed by showing bandwidth and latency results in Section 6.5 and Section 6.6 respectively. Finally, we optimize a system for low latency in Section 6.7.

## 6.1 Simulation setup

The memory controller model is implemented in SystemC and is simulated using the Æthereal network-on-chip simulator [5]. The requestors are specified using a spreadsheet and are simulated with traffic generators. The traffic generator for a requestor, $r$, sends requests strictly periodically with the period calculated in Equation (6.1). The minimum requested bandwidth, $w(r)$, is converted into requests of the specified size, $\sigma(r)$. These requests are transmitted equidistantly over time.

$$t_{gen}(r) = \frac{10^9 \cdot \sigma(r)}{w(r)} \tag{6.1}$$

A network fitting the specification is generated by an automated tool flow [6]. All requests are transmitted across the network as guaranteed service traffic [16] ensuring ordered non-lossy delivery with time-related performance guarantees. In order for the latency measurements to be comparable to the results from the analytical model we enforce that the service of a request does not stall while waiting for write data to arrive. This is accomplished by making write requests eligible for scheduling when all their data have arrived.

## 6.2 Example application

In Table 6.1[1] we present the example system that is used throughout this chapter. The system has 11 requestors, $r_0..r_{10} \in R$, and is based on the specification of a Philips video processing platform with two filters. We scaled the bandwidth requirements of the requestors to achieve a more suitable load for a 32-bit DDR2-400 memory device that has a peak bandwidth of 1600 MB/s. The specified net bandwidth requirements correspond to approximately 70% of the peak bandwidth. We also added a latency sensitive CPU with three requestors ($r_8$, $r_9$ and $r_{10}$) to the system.

| *Requestor* | *Direction* | *Request size [B]* | *Bandwidth [MB/s]* | *Max lat. [ns]* | *Traffic class* | *Partition* |
|---|---|---|---|---|---|---|
| $r_0$ | write | 128 | 144.0 | 6000 | HB | 0 |
| $r_1$ | write | 128 | 72.0 | 6000 | HB | 1 |
| $r_2$ | read | 128 | 144.0 | 6000 | HB | 0 |
| $r_3$ | read | 128 | 72.0 | 6000 | HB | 1 |
| $r_4$ | write | 128 | 144.0 | 6000 | HB | 2 |
| $r_5$ | write | 128 | 144.0 | 6000 | HB | 3 |
| $r_6$ | read | 128 | 144.0 | 6000 | HB | 2 |
| $r_7$ | read | 128 | 144.0 | 6000 | HB | 3 |
| $r_8$ | read | 128 | 50.0 | 1300 | LL | 1 |
| $r_9$ | read | 128 | 20.0 | 1300 | LL | 1 |
| $r_{10}$ | write | 128 | 50.0 | 1300 | LL | 1 |

**Table 6.1:** Specification of requestors for an example video processing system.

[1] MB is defined as $10^6$ bytes throughout the experiments.

The load and service latency requirements are not aggressively specified since we want to find solutions using both the partitioned and the memory-aware bank access patterns and compare the results. The limits for bandwidth requirements are examined in Section 6.5 and for latency in Section 6.6. The request size has been set to 128 B (4 bursts) for all requestors to be compatible with the memory-aware access pattern. This is not unreasonable for high bandwidth requestors communicating via shared memory or for cache misses in a level 2 cache. The optimization criterion is to find *the most efficient (Definition 3.4.3) solution satisfying the latency constraints of the requestors.*

## 6.3 Scheduling solutions

In this section scheduling solutions are created and compared for two systems implementing the example application. We discuss the generated solutions and calculate their total efficiency. The system presented in Section 6.3.1 uses the partitioned bank access pattern and Section 6.3.2 shows a system with the memory-aware access pattern.

### 6.3.1 Partitioned system

The example system is partitioned as shown in Table 6.1. Each of the two filters have four requestors for reading and writing luminance and chrominance values. One read and one write requestor is partitioned to every bank and the CPU is partitioned to the bank with the lowest load. This assumes that the data required by the CPU is located in that bank or that the CPU is independent of the filters. The computed scheduling solution for the partitioned system is shown in Equation 6.2.

$$\gamma_{partitioned} = ((1, 8, 6), 3) \tag{6.2}$$

The efficiency of the calculated schedule is 95.8%, meaning that the refresh group and read/write switches account for 4.2% of the available bandwidth. This is an efficient gross to net bandwidth translation. The basic group consists of eight read groups and six write groups. This is not a very good match for the specified read/write ratio, and results in a mix efficiency of 78.5%. Attempts for a closer approximation to

the requested ratio has unwanted effects. The fact that allocation is done in multiples of the request size may cause small changes in the schedule to significantly increase the allocation of the requestors. This causes latency requirements to fail if another write group is added.

The service period is determined to consist of two basic groups, resulting in three service periods for every revolution of the back-end schedule. Making the service period shorter than the schedule lowers the worst-case latency bounds due to the increased fairness in the system. It is no longer possible to maintain the shorter service period if a read group is removed since this causes the factors in $k(M, \theta)$ to change. This affects the set of service periods with maintained read/write mix and, again, causes latency requirements to fail.

The total efficiency of this system is computed in Equation (6.3) according to Definition 3.4.3.

$$e_{total} = e_\theta \cdot e_{mix} = 0.752 = 75.2\% \tag{6.3}$$

### 6.3.2 Memory-aware system

The scheduling solution for the memory-aware system looks different from that of the partitioned system, as shown in Equation 6.4.

$$\gamma_{aware} = ((2, 10, 10), 9) \tag{6.4}$$

The basic group is longer in this schedule and consists of ten read and ten write groups. This results in fewer read/write switches, which is good for memory efficiency. The memory-aware schedule ends up being slightly more effective, for this particular use-case, with a schedule efficiency of 96.9%. The mix efficiency of this system is 96.5% since equally many read and write groups comes fairly close to the requested ratio. The request group contains two refresh commands making this schedule approximately twice as long as for the partitioned system.

The total efficiency of this system is calculated in Equation (6.5). The equation shows that the efficiency is significantly higher for the memory-aware system, since latency and allocation constraints of the partitioned system caused the mix efficiency to plunge.

$$e_{total} = e_\theta \cdot e_{mix} = 0.935 = 93.5\% \tag{6.5}$$

## 6.4 Allocation results

In this section we examine the allocation results of the two systems. We compute the real and actual requirements and examine the resulting over-allocation.

Both the partitioned and the memory-aware system employ service periods shorter than the back-end schedule. The service period of the partitioned system holds 112 bursts while the memory-aware system has only 80 bursts. Short service periods are, as concluded in Section 4.5, good for the worst-case latency, especially for high bandwidth requestors, but may cause extensive over-allocation due to discretization errors. Table 6.2 shows the allocation results for the partitioned system. The table clearly shows that discretization errors causes massive over-allocation for requestors whose real requirement is small compared to their request size. This is especially apparent for $r_9$ with a real requirement of 1.5 bursts that is being rounded to the nearest multiple of the request size causing an over-allocation of 166.7%. A request from $r_9$, however, requires almost three service periods to finish without this over-allocation, which results in an unacceptable worst-case latency in the range of 4500 ns.

| *Requestor* | *Real requirement* | *Allocated bursts* | *Over-allocation* |
|---|---|---|---|
| $r_0$ | 10.5 | 12 | 14.3% |
| $r_1$ | 5.3 | 8 | 50.9% |
| $r_2$ | 10.5 | 12 | 14.3% |
| $r_3$ | 5.3 | 8 | 50.9% |
| $r_4$ | 10.5 | 12 | 14.3% |
| $r_5$ | 10.5 | 12 | 14.3% |
| $r_6$ | 10.5 | 12 | 14.3% |
| $r_7$ | 10.5 | 12 | 14.3% |
| $r_8$ | 3.7 | 4 | 8.1% |
| $r_9$ | 1.5 | 4 | 166.7% |
| $r_{10}$ | 3.7 | 4 | 8.1% |

**Table 6.2:** Allocated bursts per service period for the partitioned system. A service period contains 112 bursts.

The allocation results in 711.6 MB/s being allocated to cover the 574.0 MB/s requested for reads. 656.9 MB/s is allocated for writes requiring only 554.0 MB/s. This results in a total over-allocation due to discretization of 21.3%. The worst-case over-allocation for this scheduling solution and this set of requestors is, according to Equation (4.8) 53.4%.

The allocation in the memory-aware system is shown in Table 6.3. The real requirement of the high bandwidth requestors comes quite close to a multiple of the request size, causing a less severe over-allocation for these requestors in this system. The real requirement of the low latency requestors are still significantly smaller than the request size resulting in significant over-allocation.

| *Requestor* | *Real requirement* | *Allocated bursts* | *Over-allocation* |
|---|---|---|---|
| $r_0$ | 7.4 | 8 | 8.1% |
| $r_1$ | 3.7 | 4 | 8.1% |
| $r_2$ | 7.4 | 8 | 8.1% |
| $r_3$ | 3.7 | 4 | 8.1% |
| $r_4$ | 7.4 | 8 | 8.1% |
| $r_5$ | 7.4 | 8 | 8.1% |
| $r_6$ | 7.4 | 8 | 8.1% |
| $r_7$ | 7.4 | 8 | 8.1% |
| $r_8$ | 2.6 | 4 | 53.8% |
| $r_9$ | 1.0 | 4 | 300.0% |
| $r_{10}$ | 2.6 | 4 | 53.8% |

**Table 6.3:** Allocated bursts per service period for the memory-aware system. A service period contains 80 bursts.

This system has 697.7 MB/s allocated for read requests and 620.2 MB/s for write requests. This corresponds to a total over-allocation of 16.8%. This indicates that the over-allocation for the memory-aware system is smaller than for the partitioned system, even though the service period of the memory-aware scheduling solution is smaller. This is, however, only circumstantial as the worst-case over-allocation for this solution is 75.6%, which is higher than the 53.4% of the partitioned

system.

Table 6.4 examines how the total over-allocation is affected by the request size for the scheduling solution used by the memory-aware system. Over-allocation becomes a serious problem for this solution as the request size grows and causes allocation to fail for the example application for any request size over 128 B. The problem is solved by Algorithm 4.6 by picking a solution with a longer service period, if the latency requirements allows this.

| *Request size [B]* | *Over-allocation* |
|---|---|
| 32 | 10.0% |
| 64 | 13.4% |
| 128 | 16.8% |
| 256 | 51.2% |
| 512 | 202.4% |

**Table 6.4:** The total over-allocation increases with the size of the requests.

## 6.5 Bandwidth results

We claim that the analytical model guarantee net bandwidth to the requestors according to their specification. In this section we examine the net bandwidth provided to the requestors during simulation. The simulated time is $10^6$ ns, which corresponds to several thousand revolutions of the back-end schedule. There are some initial delays before requests arrive at the memory controller over the network, but the simulated time is considerably more than needed for the results to converge. Figure 6.1 shows the cumulative data passing the data path on behalf of the requestors in the memory-aware system. Results are not shown for the partitioned system since they are nearly identical, as they should be, since the analytical guarantee is independent of the access pattern. The results are a number of straight lines ending in the targeted levels, as shown in Table 6.5. The delivered amount of data corresponds nicely to the requested data, scaled to fit the simulation time. The discrepancy from the specified bandwidth is small, maximum 0.22%, and is believed to be attributed to the initial delay. This

means that net bandwidth is delivered to the requestors in real-time.

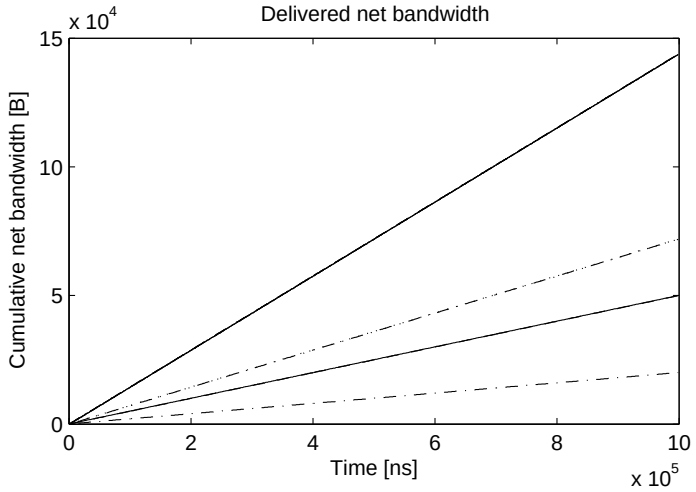


**Figure 6.1:** Cumulative data during $10^6$ ns of simulation.

| *Requestor* | *Cumulative data [B]* | *Avg. net bandwidth [MB/s]* | *Discrepancy* |
|---|---|---|---|
| $r_0$ | 143744 | 143.744 | -0.18% |
| $r_1$ | 71840 | 71.840 | -0.22% |
| $r_2$ | 143872 | 143.872 | -0.09% |
| $r_3$ | 71936 | 71.936 | -0.09% |
| $r_4$ | 143744 | 143.744 | -0.18% |
| $r_5$ | 143744 | 143.744 | -0.18% |
| $r_6$ | 143872 | 143.872 | -0.09% |
| $r_7$ | 143872 | 143.872 | -0.09% |
| $r_8$ | 50048 | 50.048 | +0.10% |
| $r_9$ | 20096 | 20.096 | +0.05% |
| $r_{10}$ | 49920 | 49.920 | -0.16% |

**Table 6.5:** Net bandwidth delivered to the requestors after $10^6$ ns.

To test the scalability of the systems we increase the bandwidth requirements of the high bandwidth requestors. A small increase causes the partitioned system to fail allocation due to unbalance in the partitions. The memory-aware system scales further as we increase the

load, although using different scheduling solutions. The system simulates properly with a gross load 89.3% using the solution shown in Equation (6.6), while the latency constraints are kept the same. Further increases of the load causes no valid solutions to be found for this application. This shows that the model is capable of dealing with a high load.

$$\gamma_{bandwidth} = ((1, 4, 4), 1) \tag{6.6}$$

## 6.6 Latency results

In this section we focus on comparing the latency results from the simulation of the two systems to the theoretical bounds computed by the analytical model. We look at the measured minimum, mean and maximum latencies when examining the latency of the simulation. The minimum value is primarily determined by the burst size and the bank access pattern. The maximum measured latency depends on the arrival process of the interconnect, the allocation scheme and the scheduling algorithm. It is interesting to compare this value to the worst-case theoretical bound since this tells something about the frequency of the worst-case. The mean latency should be kept low since it affects the performance of the system. This value also depends on the arrival process, allocation scheme and the scheduling algorithm.

A sliding QoS-aware FCFS request level scheduler is used in non-work-conserving mode and does thus not distribute slack bandwidth. We study the impact work-conservation Section 6.7.

### 6.6.1 Partitioned system latency results

Figure 6.2 shows the latency results for the partitioned system. As shown in Table 6.6 many requestors hit the theoretical minimum bound of 260 ns. The maximum measured values come close to their theoretical bounds since partitioning eliminates part of the competition in arbitration, thus reducing the complexity of the worst case. The banks in our particular system has only one reading and one writing requestor, except for bank 1 that also houses the three low latency requestors of the CPU. The figure clearly shows that the requestors partitioned to

this bank ($r_1$, $r_3$, $r_8$, $r_9$, $r_{10}$) have increased maximum measured latency, as indicated by the theoretical bounds. This is reflected in the increased mean latency. We also notice that the difference between the mean and maximum value is slightly bigger for these requestors. This reflects that it is not very common for the other requestors in the same direction with equal or higher priority mapped to the same bank, to have their requests available at the same time.

The average latency of the high bandwidth requestors is 699.9 ns and 651.9 ns for the latency sensitive ones, which is not a remarkable difference. The worst-case latency is, in fact, higher for the low latency requestors than for some of the high bandwidth requestors. It is apparent, as far as flexibility is concerned, that this system does not offer low latency to latency sensitive requestors. There are two reasons for this. Firstly, partitioning the requestors to different banks impacts latency since only one out of $n_{banks}(M)$ bursts is useful to a requestor, regardless of priority level. Secondly, partitioning limits priorities to be significant on a per-bank basis, which causes a high priority requestor to come second to low priority requestors partitioned to different banks. These are limitations inherent to this access pattern.

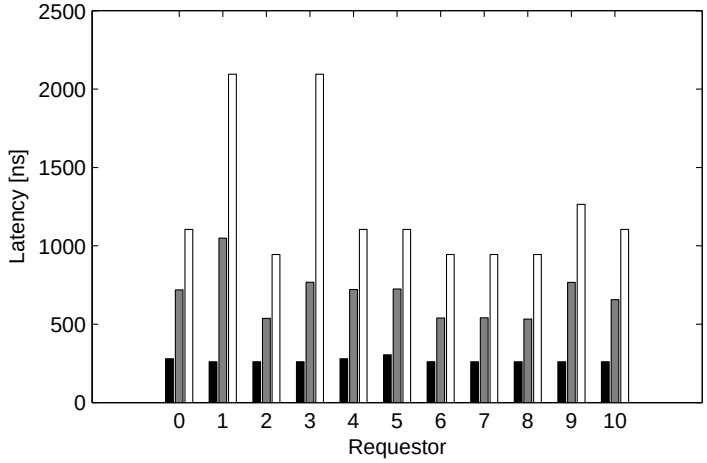


**Figure 6.2:** Simulated minimum, mean and maximum latencies using partitioning and request level scheduling

| *Requestor* | *Analytical min [ns]* | *Simulated min [ns]* | *Simulated mean [ns]* | *Simulated max [ns]* | *Analytical max [ns]* |
|---|---|---|---|---|---|
| $r_0$ | 260.0 | 280.0 | 718.8 | 1105.0 | 1105.0 |
| $r_1$ | 260.0 | 260.0 | 1048.7 | 2095.0 | 2095.0 |
| $r_2$ | 260.0 | 260.0 | 537.8 | 945.0 | 945.0 |
| $r_3$ | 260.0 | 260.0 | 767.8 | 2095.0 | 2095.0 |
| $r_4$ | 260.0 | 260.0 | 721.2 | 1105.0 | 1105.0 |
| $r_5$ | 260.0 | 3050.0 | 724.9 | 1105.0 | 1105.0 |
| $r_6$ | 260.0 | 260.0 | 539.3 | 945.0 | 945.0 |
| $r_7$ | 260.0 | 260.0 | 541.0 | 945.0 | 945.0 |
| $r_8$ | 260.0 | 260.0 | 533.0 | 945.0 | 1265.0 |
| $r_9$ | 260.0 | 260.0 | 766.1 | 1265.0 | 1265.0 |
| $r_{10}$ | 260.0 | 260.0 | 656.5 | 1105.0 | 1105.0 |

**Table 6.6:** Minimum, mean and maximum latencies using partitioning and request level scheduling.

Changing the scheduler to work on the burst level instead of the request level trades average latency for fairness. The effects of this trade are only visible in partition one, as that is the only partition with multiple requestors with the same direction and hence the only partition with requestors competing for a particular burst. The average latency of $r_8$ was increased by 12.4%, while it remained unchanged for the other requestors. This indicates that there no overlap in the presence of requests from the other requestors.

### 6.6.2 Memory-aware system results

Switching from partitioning to a memory-aware design changes the results considerably, as shown in Figure 6.3. A requestor is scheduled for four consecutive bursts at a time, which turns the burst and request level schedulers into the same algorithm for our particular request size. Since a requestor ideally starts right away and is granted four consecutive bursts it follows that the minimum measured latency is lower with this access pattern than in the partitioned system. The maximum measured latency is considerably lower than the theoretical bounds, as shown in Table 6.7, since the analytical worst-case latency is independent of the interconnect, whereas the measured worst-case is not. Every requestor in the system can not have their requests available at the same time on a shared interconnect and hence the worst-case

situation never occurs. The average latency is lower for all requestors compared with the partitioned system. The average latencies for low latency and high bandwidth requestors are 350.2 ns and 490.8 ns respectively showing that priority levels are useful to diversify the service with this pattern.

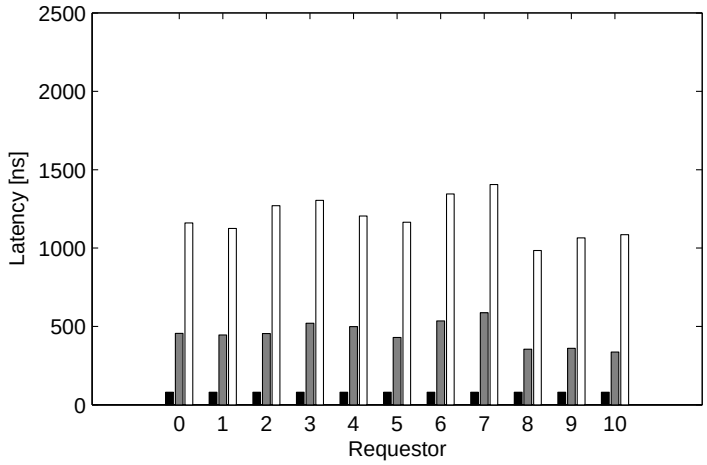


**Figure 6.3:** Simulated minimum, mean and maximum latencies using memory-aware IP design.

| *Requestor* | *Analytical min [ns]* | *Simulated min [ns]* | *Simulated mean [ns]* | *Simulated max [ns]* | *Analytical max [ns]* |
|---|---|---|---|---|---|
| $r_0$ | 80.0 | 80.0 | 455.5 | 1160.0 | 1655.0 |
| $r_1$ | 80.0 | 80.0 | 445.3 | 1125.0 | 1735.0 |
| $r_2$ | 80.0 | 80.0 | 454.6 | 1270.0 | 1735.0 |
| $r_3$ | 80.0 | 80.0 | 520.3 | 1305.0 | 1815.0 |
| $r_4$ | 80.0 | 80.0 | 499.2 | 1205.0 | 1655.0 |
| $r_5$ | 80.0 | 80.0 | 429.8 | 1165.0 | 1655.0 |
| $r_6$ | 80.0 | 80.0 | 534.9 | 1345.0 | 1735.0 |
| $r_7$ | 80.0 | 80.0 | 586.7 | 1405.0 | 1735.0 |
| $r_8$ | 80.0 | 80.0 | 354.1 | 985.0 | 1255.0 |
| $r_9$ | 80.0 | 80.0 | 360.4 | 1065.0 | 1255.0 |
| $r_{10}$ | 80.0 | 80.0 | 336.0 | 1085.0 | 1175.0 |

**Table 6.7:** Minimum, mean and maximum latencies using memory-aware IP design.

## 6.7 A latency-optimized system

The memory-aware system is clearly capable of delivering lower latency than the partitioned system. In fact, the partitioned system cannot come up with a solution with lower latency than the one presented in Section 6.3.1. The potential of the memory-aware system is shown if we change the optimization criterion to find the solution with *the lowest average worst-case latency for low latency requestors satisfying the bandwidth requirements*. The computed scheduling solution is shown in Equation 6.7.

$$\gamma_{latency} = ((1, 2, 2), 3) \tag{6.7}$$

This back-end schedule is shorter than the previous one since only one refresh command is included in the refresh group. The basic group is also shorter and consists of two read groups and two write groups, which helps worst-case latency at the expense of the scheduling efficiency dropping down to 90.0%. Since the number of read groups still equals the number of write groups the mix efficiency remains at 96.5%.

The service period consist of three basic groups, or 112 bursts, and results in an over-allocation of 14.0% (worst-case 50.1%). The latency results of this solution are shown in Figure 6.4.

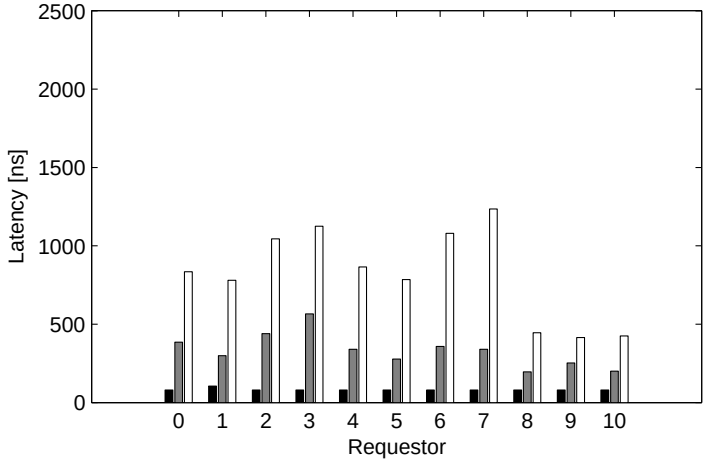


**Figure 6.4:** Minimum, mean and maximum latencies using memory-aware IP design in a latency-optimized system.

The measured and theoretical worst-case latency for the low latency requestors, $r_8$, $r_9$ and $r_{10}$, are approximately cut in half, as shown in Table 6.8, compared to the solution presented in Section 6.3.2. The tighter bounds of the new solution also affects the average measured latency of the requestors. The average latency for the low latency requestors is 216.0 ns, which is a reduction of 38.3%. The high bandwidth requestors are not considered by the new optimization criterion and have their theoretical worst-case latency bounds increased. The average latency for these requestors is 375.6 ns, which is an improvement despite the increased theoretical bounds.

| *Requestor* | *Analytical min [ns]* | *Simulated min [ns]* | *Simulated mean [ns]* | *Simulated max [ns]* | *Analytical max [ns]* |
|---|---|---|---|---|---|
| $r_0$ | 80.0 | 80.0 | 385.4 | 835.0 | 1940.0 |
| $r_1$ | 80.0 | 105.0 | 299.5 | 780.0 | 2210.0 |
| $r_2$ | 80.0 | 80.0 | 439.3 | 1045.0 | 2210.0 |
| $r_3$ | 80.0 | 80.0 | 565.8 | 1125.0 | 2290.0 |
| $r_4$ | 80.0 | 80.0 | 340.2 | 865.0 | 1940.0 |
| $r_5$ | 80.0 | 80.0 | 277.0 | 785.0 | 1940.0 |
| $r_6$ | 80.0 | 80.0 | 358.5 | 1080.0 | 2210.0 |
| $r_7$ | 80.0 | 80.0 | 339.5 | 1235.0 | 2210.0 |
| $r_8$ | 80.0 | 80.0 | 195.3 | 445.0 | 540.0 |
| $r_9$ | 80.0 | 80.0 | 252.4 | 415.0 | 540.0 |
| $r_{10}$ | 80.0 | 80.0 | 200.3 | 425.0 | 460.0 |

**Table 6.8:** Minimum, mean and maximum latencies using memory-aware IP design in a latency-optimized system.

The average-case is further improved by allowing requestors to degrade to best effort priority, as discussed in Section 5.2, and also distribute the slack bandwidth in the system. This improvement results in an mean reduction of the average measured latency of 2.6%.

Chapter 7

# Conclusions and future work

We present an analytical memory controller model for embedded system that is a hybrid between contemporary static and dynamic solutions. We show that it is possible to provide hard real-time guarantees on net bandwidth and worst-case latency by fixing the back-end schedule and guarantee an allocated number of memory bursts to every requestor in a service period. An example application is analytically verified and successfully simulated with a load of 89.3% of the peak memory bandwidth. When optimizing for latency a theoretical worst-case latency bound of 550 ns for low latency requestors is provided.

Future work involves further real-time analysis as the difference between the analytical and measured worst-case latency may be large in systems with many competing memory requestors. The analytical bounds assumes that the interconnect can deliver requests for all requestors simultaneously. This is the case for direct wires but not in systems with shared communications resources, such as a bus or a network-on-chip. To get tighter bounds for these systems the analysis has to be extended to cover the properties of the interconnect. Latency bounds can also be tightened further by examining the fairness properties of employed scheduling algorithm.

The requestors are assumed to conform to particular bank access patterns. This is accomplished by partitioning the requestors to particular banks or making sure that a request always access all banks in sequence. This constraint is, however, rather severe and it may be very difficult to get the requestors of a system to conform to this assumption. We show that the choice of bank access pattern has a tremendous impact on latency. Further work is required in this area to

ease application mapping.

The dynamic front-end scheduler is used to increase the flexibility of the model. We show that although flexibility is increased, it is limited by the bank access pattern and the fixed back-end schedule. The results in lower flexibility than a completely dynamic controller since every requestor, regardless of priority level, has bursts in the interfering direction included in the average and worst-case latency. This means that the model trades flexibility for the ability to offer hard real-time net bandwidth guarantees. This can be improved by making the gross to net bandwidth translation dynamic. It is possible to generate part of the back-end schedule dynamically in run-time if the number of switches is bounded. This increases flexibility at the potential cost of memory efficiency.

Discretization errors cause over-allocation that increase with larger request size, shorter service periods and the number of requestors. This inhibits scaling of the model and should be examined further.

# References


[1] Aras, C., Kurose, J., Reeves, D., and Schulzrinne, H. Real-time communication in packet-switched networks. In *Proceedings of the IEEE* (January 1994), vol. 82, pp. 122–139.

[2] ARM. *PrimeCell Dynamic Memory Controller (PL340)*, r0p0 ed., June 2004.

[3] Bensaou, B., Tsang, D. H. K., and Chan, K. T. Credit-based fair queueing (CBFQ): a simple service-scheduling algorithm for packet-switched networks. *IEEE/ACM Trans. Netw. 9*, 5 (2001), 591–604.

[4] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols* (1989), ACM Press, pp. 1–12.

[5] González Pestana, S., Rijpkema, E., Rădulescu, A., Goossens, K., and Gangwal, O. P. Cost-performance trade-offs in networks on chip: A simulation-based approach. In *DATE' 04: Proceedings of the conference on Design, Automation and Test in Europe* (Feb 2004), pp. 764–769.

[6] Goossens, K., Dielissen, J., Gangwal, O. P., González Pestana, S., Rădulescu, A., and Rijpkema, E. A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification. In *DATE' 05: Proceedings of the conference on Design,*

*Automation and Test in Europe* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 1182–1187.

[7] Goossens, K., Gangwal, O. P., Röver, J., and Niranjan, A. P. 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*, J. Nurmi, H. Tenhunen, J. Isoaho, and A. Jantsch, Eds. Kluwer, 2004, ch. 15, pp. 399–423.

[8] Harmsze, F., Timmer, A., and van Meerbergen, J. Memory arbitration and cache management in stream-based systems. In *DATE* (2000), pp. 257–262.

[9] Heithecker, S., do Carmo Lucas, A., and Ernst, R. A mixed QoS SDRAM controller for FPGA-based high-end image processing. In *IEEE Workshop on Signal Processing Systems* (Aug 2003), IEEE, pp. 322–327.

[10] Jaspers, E. G. Interfacing DDR SDRAM for video communication. Technical Note NL-TN 2000/346, Philips Electronics, September 2000.

[11] JEDEC Solid State Technology Association. *DDR2 SDRAM Specification*, jesd79-2a ed. JEDEC Solid State Technology Association 2004, 2500 Wilson Boulevard, Arlington, VA 22201-3834, Jan 2004.

[12] Lin, T.-C., Lee, K.-B., and Jen, C.-W. Quality-aware memory controller for multimedia platform soc. In *IEEE Workshop on Signal Processing Systems, SIPS 2003* (August 2003), pp. 328 – 333.

[13] Nagle, J. B. On packet switches with infinite storage. *IEEE Transactions on Communications COM-35*, 4 (April 1987), 435–438.

[14] Otero Pérez, C., Rutten, M., van Eijndhoven, J., Steffens, L., and Stravers, P. Resource reservations in shared-memory multiprocessor SOCs. In *Dynamic and Robust Stream-*

*ing In And Between Connected Consumer-Electronics Devices*, P. van der Stok, Ed. Kluwer, 2005.

[15] Parekh, A. K., and Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw. 1*, 3 (1993), 344–357.

[16] Rijpkema, E., Goossens, K., Rădulescu, A., Dielissen, J., van Meerbergen, J., Wielage, P., and Waterlander, E. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip. *IEEE Proceedings: Computers and Digital Technique 150*, 5 (Sep 2003), 294–302.

[17] Rixner, S., Dally, W. J., Kapasi, U. J., Mattson, P., and Owens, J. D. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture* (2000), ACM Press, pp. 128–138.

[18] Roest, J. Spider project: Detailed design description of the DDR SDRAM controller. Tech. Rep. 1.3, Philips Consumer Electronics, Mar 2004. Philips confidential.

[19] Shreedhar, M., and Varghese, G. Efficient fair queueing using deficit round robin. In *SIGCOMM* (1995), pp. 231–242.

[20] Steenhof, F. Columbus SDRAM interface. Tech. Rep. 0.8, Philips Consumer Electronics, Nov 2002. Philips confidential.

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

[22] Woltjer, L. Optimal DDR controller. Master's thesis, University of Twente, 2005. Philips Semiconductors Confidential.

[23] Zhang, H. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE 83*, 10 (Oct. 1995), 1374–96.

[24] Zhang, H., and Keshav, S. Comparison of rate-based service disciplines. In *SIGCOMM '91: Proceedings of the conference*

*on Communications architecture & protocols* (1991), ACM Press, pp. 113–121.

# List of symbols

**Bandwidth allocation**

| | | |
|---|---|---|
| $A(r, M, \theta, p)$ | net bandwidth allocated to a requestor | 36 |
| $a(r, M, \theta, p)$ | bursts allocated to a requestor | 36 |
| $e_{alloc}(r, M, \theta, p)$ | allocation efficiency | 38 |
| $o(R, r, M, \theta, p)$ | over-allocation | 38 |
| $o_{wc}(R, r, M, \theta, p)$ | worst-case over-allocation | 38 |
| $p$ | service period | 34 |
| $\|p\|$ | bursts in service period | 34 |
| $x(M, \theta, p)$ | service periods in back-end schedule | 36 |

**Back-end schedule**

| | | |
|---|---|---|
| $\alpha(R)$ | requested read/write ratio | 27 |
| $\beta(\theta)$ | provided read/write ratio | 27 |
| $c_{read}(\theta)$ | consecutive read groups in basic group | 23 |
| $c_{write}(\theta)$ | consecutive write groups in basic group | 23 |
| $e_{mix}(M, \theta)$ | mix efficiency | 30 |
| $e\prime_{mix}(M, \theta)$ | alternate mix efficiency | 30 |
| $e_{\theta}(M, \theta)$ | back-end schedule efficiency | 29 |
| $e_{total}(M, \theta, R)$ | total efficiency | 30 |
| $k(M, \theta)$ | basic groups in schedule | 28 |
| $n(\theta)$ | refresh commands in refresh group | 23 |
| $n_{\theta}(M, \theta)$ | revolutions per second of schedule | 37 |
| $S\prime(M, \theta)$ | net bandwidth provided by schedule | 29 |
| $t_{avail}(M, \theta)$ | maximum available cycles | 27 |

**Memory timings**



**Memory properties**



**Requestors**

| | | |
|---|---|---|
| $R\prime(R, r)$ | set of equal or higher priority requestors | 42 |
| $\sigma_{burst}(r, M)$ | request size in words | 37 |
| $\sigma(r)$ | maximum request size | 7 |
| $s_r(t_0, t_1)$ | service given to requestor in time interval | 48 |
| $t_{max}(r)$ | maximum latency requirement | 7 |
| $w(r)$ | bandwidth requirement (bytes) | 7 |
| $w_{real}(r, M, \theta, p)$ | real burst requirement | 37 |

**Worst-case latency**

| | | |
|---|---|---|
| $n_{left}(R, r, M, \theta, p)$ | remaining bursts | 43 |
| $n_{ref}(M, \theta, p)$ | interfering refresh groups | 44 |
| $n_{switches}(R, r, M, \theta, p)$ | number of switches | 43 |
| $t_{direction}(R, r, M, \theta, p)$ | direction related latency | 43 |
| $t_{lat}(R, r, M, \theta, p)$ | worst-case latency of a request | 44 |
| $t_{mismatch}(r, M)$ | Arrival/arbitration mismatch latency | 44 |

**Miscellaneous**

| | | |
|---|---|---|
| $D$ | set of directions | 7 |
| $d$ | direction | 7 |
| $\delta(R, r)$ | delay due to other requestors | 52 |
| $\gamma$ | scheduling solution | 45 |
| $\kappa$ | fairness bound | 48 |
| $t_{gen}(r)$ | periodicity of traffic generator | 57 |