

Chapter 2

Reconfigurable Real-Time Memory Controller Architecture

The purpose of this chapter is to set the stage on which the rest of this book plays out. We describe the technology that we work with, in the form of the SDRAM chips that external companies produce for us (and the rest of the world) in Sect. 2.1. Because, the same SDRAM chips are used by everyone, it is not surprising that most memory controllers, i.e., the interfaces that interact with these chips, have at least the same high-level structure, as introduced earlier in Sect. 1.2. For the sake of efficiency, the proverbial wheel tends to be invented only a few times before the interested community settles for a design that works in most cases. Further improvements are driven by the needs of specific application areas and the gradual evolution of the surrounding actors and requirements. This book focuses on the area of mixed time-criticality systems, and uses an existing SDRAM controller template for real-time systems, the *pattern-based controller* [1], as its starting point. The properties of this controller are introduced in Sect. 2.2.

The story continues with a detailed description of our novel reconfigurable memory controller architecture in Sect. 2.3. It partially concerns the introduction of concepts and structures used in the controller, and touches upon some of the real-time aspects that are influenced by its structure and implementation. This controller is the framework on which the other contributions in this book are pinned. The memory patterns we generate in Chap. 3 are stored *within* this controller. The analysis from Chap. 4 and the trade-offs we describe in Chap. 5 *apply* to memory controllers that follows the architecture template we describe here, and the conservative open-page policy in Chap. 6 is *implemented on* a slightly modified version of the same template. The embedded reconfiguration hardware *enables* the controller to adapt to different use-cases as we describe in Chap. 7.

In Sect. 2.4, we derive a worst-case performance model for this memory controller architecture, based on a *Latency-rate* (\mathcal{LR}) server abstraction. This performance model applies to many well known arbiters and can be used in frameworks that enable system-level analysis. We then continue with a discussion on the implementation of a hardware instance on *Field Programmable Gate Array* (FPGA) in Sect. 2.5, which demonstrates that this memory controller is not only conceptually sound, but really works when it is connected to a real SDRAM module and integrated

in the *Composable System-on-Chip (CompSOC)* platform [2]. Its cost in terms of resource usage are evaluated and contrasted with a comparable FPGA controller implementation in Sect. 2.6.

2.1 SDRAM

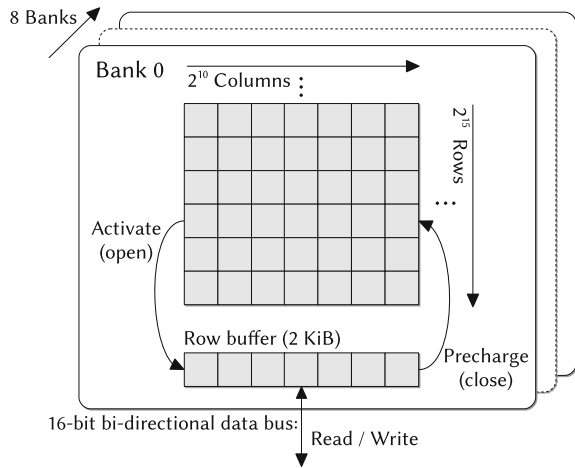
SDRAM is an extremely popular type of memory. DRAmExchange (a market analyst) reports that in February 2015 alone, 2.4 billion 2 gibibit (2^{30}) equivalent units were produced worldwide [3], for a total *capacity* of 5.16 exabits. This amounts to a production rate of 267 GB/s,¹ a relatively modest “bandwidth” that about 100 combined contemporary *SDRAM devices* (single chips) could easily deliver in the worst case, as we later show in Chap. 5.

SDRAM is volatile and used as temporary data storage, similarly to caches or *Static Random-Access Memory (SRAM)* memories. It only stores data as long as power is provided to it. In terms of area and power consumption, it is cheaper than SRAM, since it requires only a single transistor-capacitor pair to store a bit. This efficiency makes it feasible to store gigabytes of data in SDRAM, while SRAM and caches are limited to capacities in the order of megabytes.

Many *generations* of SDRAM have been developed since it was invented by Robert Dennard in 1967 [5], but most of their characteristics are similar. SDRAM devices contain a hierarchically structured storage array [6]. A schematic view on a generic SDRAM architecture is shown in Fig. 2.1. Each device consists of typically 8 or 16 *banks* that can work in parallel, but share a *command, address, and data bus*. Therefore, only one command can be sent to *one* bank at a time, but commands can take multiple cycles to complete, and the execution of commands on different banks can happen in a parallel (pipelined) fashion. A bank consists of a memory array, divided into *rows*, each row containing a certain number of *columns*. A column is as wide as the number of pins on the memory device’s data bus, and hence only one bank may drive the data pins at a time. Typically, there are 2^{10} or 2^{11} columns per row, and about 2^{14} – 2^{16} rows per column, depending on the capacity of the device and its data bus width. SDRAMs with 4, 8, 16, and 32-bit data buses exist. The data bus is bidirectional, i.e. the same pins are used for both reading and writing. Some SDRAMs are *Single Data Rate (SDR)*, transporting valid data on the rising clock edges only. However, all memory generations we consider in this book use a *Double Data Rate (DDR)*, i.e., they transfer one data word (which is as wide as the data bus) on both the rising and the falling edge of the clock.

¹ $\frac{2402 \times 10^6 \cdot 2 \times 2^{30} / 8 \text{ bytes}}{2.419 \cdot 10^6 \text{ s}}$. Incidentally, this is only 0.15 % less than the traffic flowing into the Amsterdam Internet Exchange (AMS-IX) in the same month [4] (645772 TB). The (live) construction of an SDRAM cache of a significant portion the Internet traffic was hence possible, although it might have been the last month this was feasible, given the growth trend of AMS-IX traffic. The power footprint of this Internet cache might be problematic though.

Fig. 2.1 Schematic view on the architecture of an SDRAM device with the dimensions of a 512 MiB DDR3-1600 chip (see Appendix B)



The name of an SDRAM device starts with its generation name, followed by its data rate in MHz, so for example DDR3-1600 refers to a DDR3 memory with a 800MHz command clock frequency. In this book, we refer to the generation name as the *SDRAM type*. The width of the data bus is often indicated by a postfixed ‘x’ followed by the width in bits, e.g., an LPDDR2-1066x32 has a 32-bit data bus. The capacity of SDRAM devices is usually expressed in multiples of Mib (2^{20} bits) or Gib (2^{30} bits), although the ‘i’ is commonly dropped in datasheets. Bandwidths in this book use SI prefixes. For example, fully reading a 512 MiB SDRAM with a bandwidth of 512 MB/s takes about 1.049 s (Gi is 7.3 % larger than G).

2.1.1 SDRAM Commands

An SDRAM can be instructed to perform certain actions by giving it *commands*. There are six main SDRAM commands: (1) *Activate (ACT)*, (2) *Read (RD)*, (3) *Write (WR)*, (4) *Precharge (PRE)*, (5) *Refresh (REF)* and (6) *No operation (NOP)*. The *command bus* of a DDR3 SDRAM consists of 4 wires: *row address strobe (RAS)*, *column address strobe (CAS)*, *chip select (CS)* and *write enable (WE)*. The combination of these wires forms a (4-bit) command, which is clocked into the SDRAM. The other generations use a similar interface, although some reuse parts of the address bus as command wires. The commands work as follows:

- An ACT command opens a row in a bank, and makes it available for subsequent RD and WR commands by moving its content to the *row buffer* of the bank. An activate command is accompanied by the *address of the row* that should be opened.
- Each RD or WR command results in a *burst* of data, consisting of a range of columns from the active row. One burst occupies the data bus for multiple

consecutive cycles. The number of words per RD or WR is called the *Burst Length* (*BL*). Across contemporary memory generations the commonly supported value for BL is 8 [7–12]. The memory generations we consider all have a DDR, transporting data on both the rising and falling clock edge. Therefore, it takes only $BL/2$ clock cycles to transfer a burst. A RD or WR command is accompanied by the *address of the first column of the burst*, which generally must start at a multiple of the burst length. Data is available on the data bus after the associated *read or write latency* after the RD or WR has been issued. The latencies for RD and WR commands may be different, but tend to be of the same order of magnitude (see Table 2.1).

- The PRE command closes a row, i.e., it stores the contents of the row buffer in the memory array, allowing for another row to be subsequently opened. Only one row per bank can be open at a time. An optional *auto-precharge* flag can be added to RD and WR commands, such that the associated row is closed as soon as the read or write is completed. A RD or WR with auto-precharge can be regarded as a regular RD or WR, followed by a PRE command from a timing perspective. The difference is that the precharge does not require the command bus. This frees a slot in the command schedule, which may be used for other commands. Another command that precharges banks is called *Precharge All* (*PREA*). As the name suggests, it precharges all banks that are currently open.
- SDRAM is volatile, because the transistor–capacitor pairs it uses to store bits lose their charge over time. To avoid data loss, the memory must be *refreshed* periodically by issuing a REF command. The required refresh command interval depends on the operating temperature and the memory size, and ranges between approximately 1 and 10 μs [7–12]. In this book, we assume the SDRAM always works within a fixed temperature range, and that the refresh interval is set to an appropriate (fixed) value.
- Finally, the NOP command does nothing. It is used to fill the time, e.g., while waiting for timing constraints (see Sect. 2.1.2) to be resolved. Some standards also support a *deselect* (*DES*) command that behaves similarly to a NOP, while others

Table 2.1 Approximate values of SDRAM timings relative to RC

Timing	Related constraint	Approximate value
RC	ACT-to-ACT, same bank	45–60 ns
RAS	ACT-to-PRE, same bank	70 % of RC, 35 ns
RCD	ACT-to-RD/WR in the same bank	30 % of RC, 15 ns
RP	PRE-to-ACT, same bank	30 % of RC, 15 ns
RRD	ACT-to-ACT, same device	25–30 % of RC, 12.5 ns
RFC	REF-to-ACT, same device	1–5 times RC, depends on capacity
RL, WL or CL	RD/WR-to-data	30 % of RC, 15 ns
FAW	Four Activate Window	85 % of RC (50 % for DDR4)

only have DES commands. We do not require a distinction between NOP and DES commands in this book, and always refer to unused command bus cycles as NOPs.

Four relevant command relate to the entry and exit of various power-down modes. They are called *Self Refresh Entry (SRE)*, *Self Refresh Exit (SRX)*, *Power-down Entry (PDE)*, *Power-down Exit (PDX)*. Section 4.3 explains what these commands do exactly when it introduces the SDRAM power state machine.

The scheduling of PRE and ACT commands is determined by the memory controllers’ *page policy*. Memory controllers that leave a row open after a request is completed use an *open-page policy*, while those that close (precharge) it as soon as possible use a *close-page policy* [13]. A request that does not require an activate command, because the row it accesses is still open, is called a *row hit* or *page hit*. Requests that target a closed row are called *row misses* or *page misses*. We return to discuss page policies in Chap. 6.

The relation between the command, address and data bus is shown in Fig. 2.2. In figures, we often show traces of commands as a series of rectangular blocks, like at the top of Fig. 2.2 for example. Each block in this series represents a command. A block may contain a letter representing the command type, and a number, representing the

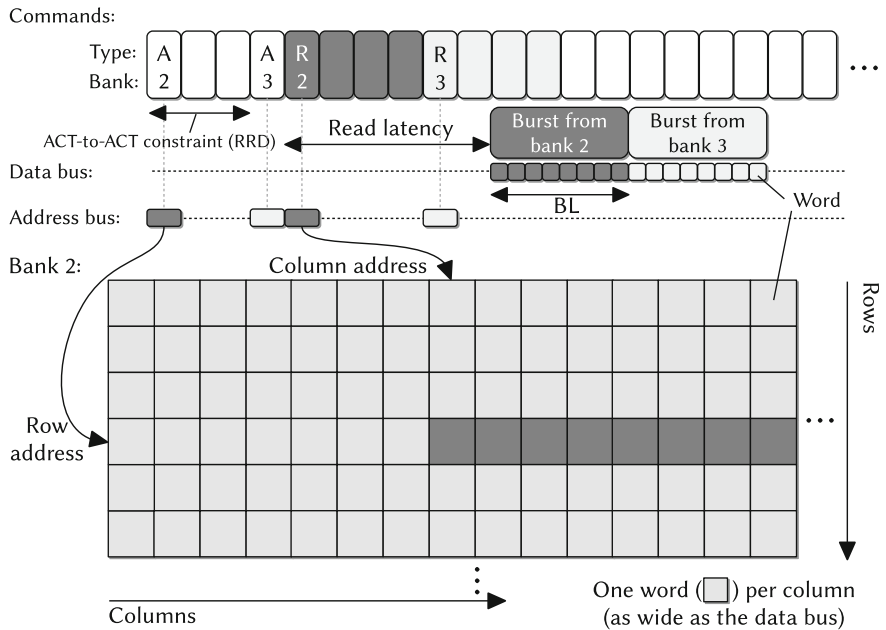


Fig. 2.2 High-level SDRAM operation. The activation of bank 3 happens in parallel with the read command to bank 2. Data bursts of different banks are serialized, since the data bus is shared across banks. The two cycles between A2 and A3 are the result of the ACT-to-ACT timing constraint (RRD)

bank to which the command is directed. We abbreviate ACT, PRE, RD and WR by A, P, R, and W, respectively, and encode NOPs as empty boxes (Fig. 2.2).

2.1.2 *Timings and Timing Constraints*

Vendors of SDRAM devices characterize their memory chips by specifying their *timings*. Timings define the maximum time between internal operations in the memory, usually relating to the (analog) propagation delay between distinct components in the SDRAM. *Timing constraints* are built as mathematical expressions from these timings, and they define the minimum time between pairs of commands based on the state of the memory, which in turn is a consequence of earlier executed commands. An SDRAM controller has to satisfy all timing constraints to operate correctly. A detailed explanation of what each timing represents for a specific memory generation is found in the standards [7–12]. For the purpose of this book, these details are less important, since we mostly consider the SDRAM as a black box that we merely have to use according to its interface specification. Appendix B shows the numerical values associated with the timings of a range of SDRAM devices, while Chap. 3 provides a detailed view on the relation between timings and timing constraints. However, we will sometimes refer to timings before Chap. 3 to point out trends, and hence provide some early intuition on their relative length in Table 2.1. All numbers in this table are approximates, because timings vary across SDRAM devices and generations.

Some constraints only restrict commands for a single bank, like RC, and RCD for example, while others like RRD and RFC, are device-level constraints. The *Four Activate Window* (FAW) is different from other constraints. Instead of specifying a minimum distance between two commands, it defines a rolling time window in which at most four activate commands may be executed. In this book, we typeset timings in SMALL CAPS.

2.1.3 *Memory Generations*

SDRAM technology has evolved over the years. JEDEC creates the standards that ensure compatibility between devices of the same memory generation from different vendors. We consider six generations in this book. Chronologically ordered by the date of their introduction, they are: DDR2 [12], DDR3 [8], LPDDR [7], LPDDR2 [14], LPDDR3 [11], and DDR4 [10]. Newer standards evolve by defining timings for higher clock frequencies and modifications of the physical interface. The optional *LP*-part in a generation name stands for *Low Power*, and the respective standards are more suited for power/energy constrained systems, for example, by operating at a lower supply voltage, or by the introduction of more efficient low-power modes. LPDDR devices have a maximum of 4 banks, while DDR2s can have 4 or 8 banks, and DDR4 may have 8 or 16 banks. The remaining generations

always have 8 banks. Occasionally standards are augmented with new features, like a reduced supply voltage, for example, as in the case of DDR3L [15].

2.1.3.1 DDR4 Bank Groups

DDR4 introduces *bank groups*: banks are clustered into (at least two) bank groups per device. Banks in a bank group share power-supply lines. To limit the peak power per group, sending successive commands to the same group makes certain timings larger. These timings are postfixed with *_L* (long) or *_S* (short) for commands for the same or a different bank group, respectively. Successive RD or WR commands to the same group need to be separated by at least CCD_L cycles. Because CCD_L is larger than the number of cycles per data burst (BL/2), performance is impacted by CCD_L unless bursts are interleaved across bank groups.

2.1.4 Memory Hierarchies

SDRAM devices can be used as standalone chips, as generally done in embedded SoCs [16–19] for example (Fig. 2.3). The *Interface Width (IW)*, which we define as the width of the data bus between the memory controller and the SDRAM, is then equal to the data bus width of this chip, and typically ranges from 8 up to 32 bits.

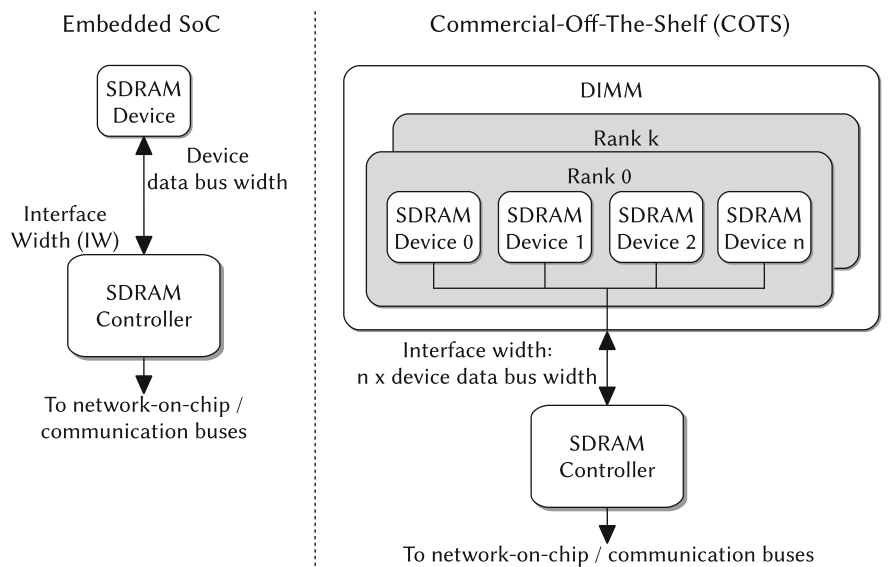


Fig. 2.3 Typical memory hierarchy for embedded SoCs and COTS systems

Bigger and wider memories can be built by having multiple chips work in lock-step in a *rank*, executing the same commands, producing or consuming data in parallel. The IW of the controller is then equal to the combined data bus width of all these chips. The request size divided by IW and BL determines how many data bursts, and thus RD or WR commands should be generated for a request, with a minimum of one burst.

Multi-device setups are typically used in *Commercial-Off-the-Shelf* (COTS) and high-performance computer systems. Memory chips are not bought individually for these systems, but instead come pre-combined on *Dual Inline Memory Modules* (DIMMs) [20] or *Small Outline DIMMs* (SO-DIMMs) [21] that contain one or more ranks with a combined data bus width of 64 bits. Ranks can share a command and data bus, as long as they do not drive the data bus simultaneously. Finally, a memory hierarchy may contain multiple independent groups of ranks called *channels*, each with an individual SDRAM controller.

In this book, we target embedded SoCs, and hence most of our examples are based on relatively narrow interfaces compared to DIMMs. The techniques that we propose are independent from how the memory hierarchy beneath the SDRAM controller is built, i.e., both single devices or DIMM modules can be supported. We do, however, rely on a custom controller architecture (Sect. 2.2), which by definition places this work outside of the COTS realm (assuming FPGA development kits are classified as non-COTS). We also focus on a single SDRAM controller, leaving multiple channels out of the equation. The interested reader can refer to [22] for more information on multi-channel real-time memory controller architectures and configuration.

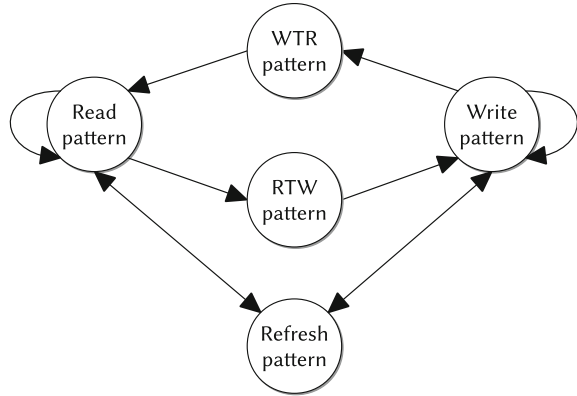
2.2 Pattern-Based SDRAM Controllers

To create a predictable SDRAM resource, useful bounds on the response time of memory requests have to be given. The underlying technique by which our memory controller bounds the response time of a request is the approach from [1], revolving around *memory patterns*. A *memory pattern* is a design-time constructed series of SDRAM commands with a known execution time (length) and a specific function.

The commands in a pattern are scheduled such that all timing constraints within the pattern itself are satisfied. Six different *patterns types* exist: (1) *Read*, (2) *Write*, (3) *Read-To-Write switch* (RTW), (4) *Write-To-Read switch* (WTR), (5) *Refresh* and (6) *Idle* patterns. The sequences of patterns that can be executed by the controller are summarized in Fig. 2.4. The function of each pattern type is the following:

- Read and write patterns are *access patterns* that transport data from and to the SDRAM, respectively. Multiple read patterns and multiple write patterns may be executed successively, indicated by their respective self-edges in Fig. 2.4. In their construction, that factor has to be taken into account, such that SDRAM timing constraints within and across these patterns are not violated. Typically, read and write patterns contain between 1 and 32 bursts (RD or WR commands). Read and write patterns implement a close-page policy. They activate the banks they will be accessing, and all banks are precharged at the end of the pattern.

Fig. 2.4 Allowed pattern sequences



- Switching patterns consist of only NOPs. They are inserted between a read and write pattern to resolve timing constraints across access patterns of opposing types. If there are no such constraints, or if no additional NOPs are required to satisfy them, then switching patterns may have a length of zero.
- A refresh pattern consists of a single refresh command preceded and succeeded by enough NOPs such that it can be scheduled after an access pattern without violating timing constraints. The switching patterns and the refresh pattern are called the *auxiliary patterns*.
- Finally, the idle time of the controller can be discretized explicitly into idle or power-down patterns [23]. We do not evaluate the use of power-down patterns in combination with the techniques proposed in this book, and hence we stick to idle patterns consisting only of NOPs. Idle patterns can be inserted on most edges in Fig. 2.4 (only not between WTR and read patterns and RTW and write patterns). Their minimum size is 1 cycle.

There is *one* pattern of each type available to the memory controller in what is called a *pattern set*. The SDRAM controller makes scheduling decisions at the granularity of patterns instead of individual commands, which simplifies bounding its performance. Some close-page real-time controllers use variations of memory patterns in their architecture [24–27], scheduling patterns from such a set instead of individual commands. This simplifies the logic of the controller, since there are fewer constraints it has to track. Others define patterns only in their worst-case analysis [28, 29], knowing the behavior of their architecture is bounded by them. In both cases, the analysis complexity is greatly reduced.

2.2.1 Burst Grouping

The smallest request size that a memory controller has to process is often larger than the size of one read or write burst to a memory device in the embedded SoCs

we focus on. This means that multiple bursts can be grouped together to form a single atomic access at a larger access granularity. The relative order of bursts within one such an atom is fixed, which gives it guaranteed properties that improve the worst-case performance. Intuitively, this effect can be understood as a sort of batch processing, in which groups of bursts that are relatively similar can be processed more quickly than those that are relatively different. Most memory controllers try to achieve some degree of burst grouping. The banks to which these grouped bursts are sent is determined by the low-level memory map. Depending on this memory map, grouping bursts can guarantee:

1. *Bank parallelism*: The atom is interleaved over multiple banks that work in parallel to produce or consume data. While one bank is precharged or activated, other banks are accessed with read and write commands.
2. *Consecutive bursts access the same row*: Multiple bursts are fetched from the same row in the same bank within an atom, in essence generating guaranteed row hits, and guaranteeing no read-write switching of the data bus across those bursts.

Timing constraints enforce a minimum amount of time between consecutive activations of the same bank, and they also separate bursts of different types (read/write). Atomically grouping bursts helps to reduce the overhead of these two effects, improving the memory efficiency, since more useful commands are executed in the same amount of time, as shown in Fig. 2.5.

A trade-off exists between these two effects: requests have a fixed size, and hence there is only a limited movement range within these two dimensions. The width of the SDRAM's data bus also plays an important role here. The wider it is, the more bits are transferred per burst, and the fewer bursts can be grouped to fill an atom with a given access granularity. DIMM-based (COTS) systems, which typically have a

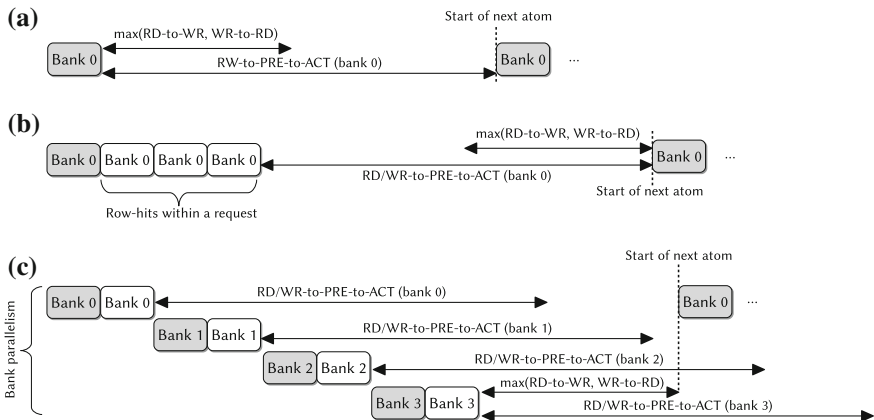


Fig. 2.5 Examples of the effects of grouping bursts. *Shaded* bursts are page misses. It shows how the number of bursts that can be executed within a fixed amount of time varies based on how they are grouped. **a** Using BI 1, BC 1. **b** BI 1, BC 4. **c** BI 4, BC 2

64-bit bus, are hence more limited in their ability to exploit burst grouping compared to embedded SoCs that typically use single SDRAM chips with a smaller (8–32 bit) interface.

We define two parameters to characterize where a controller operates within this configuration space:

1. *Bank Interleaving (BI)*: the number of banks that are accessed atomically, and
2. *Burst Count (BC)*: the number of bursts per bank.

Using these parameters, we can describe the *Access granularity (AG)* of a pattern-based controller, i.e., the number of bytes that are transported within a read or write pattern. It depends of the number of bursts in the pattern, given by $BI \cdot BC$, the length of a burst in words (BL), and the number of bytes per word, which is equal to the interface width in bytes (IW):

$$AG = BI \cdot BC \cdot BL \cdot IW \quad (2.1)$$

The worst-case or average-case behavior of an SDRAM controller's command scheduler can be characterized by a (BI, BC) combination, and this in turn determines its performance. Some real-time memory controllers interleave bursts belonging to one request over all available banks [25, 28, 29]. Others interleave consecutive bursts to different banks [24, 26], but the origin of each of these bursts may be a different request. Controllers using open-page policies generally assume each request maps to a single burst [30, 31]. Stiliadis and Varma [25] considers the number of bursts per bank as configuration parameter, but not the number of banks. Chapter 3 turns BI and BC into an integral part of the generation of patterns for our memory controller, and Chap. 5 shows the configuration trade-offs when both degrees of freedom are used.

2.3 Controller Architecture

The section describes the architecture template of a pattern-based SDRAM controller. Figure 2.6 shows the three main blocks that constitute its architecture. We make a distinction between the *resource front-end*, which deals with the preparation of requests from clients and the arbitration amongst them, and the *SDRAM back-end*, which schedules patterns and translates them into SDRAM commands. Finally, the *Physical interface (PHY)*, deals with the physical connection to the (off-chip) SDRAM. The following sections introduce the different components within these blocks, discuss their functionality, and their qualitative impact on the worst-case performance where relevant.

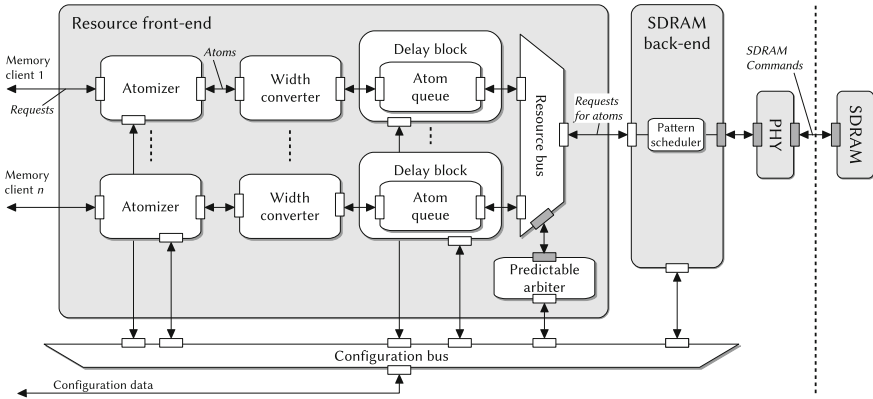


Fig. 2.6 SDRAM controller architecture. *Arrows* indicate the flow direction of data

2.3.1 Resource Front-End

The first block we discuss is the resource front-end. Its primary function is to enable sharing of the SDRAM amongst multiple clients. It implements and extends the general template from [32]. First we look at the interface that is exposed to the memory clients, and we discuss the contents of the front-end.

2.3.1.1 DTL Interfaces per Client

The controller has a *Device Transaction Layer (DTL)* interface [33] for each memory client, which is a handshake-based communication standard that is similar to AXI4 [34]. DTL has individual command, read-data and write-data channels, and supports multiple outstanding (pending) requests. Each DTL request consists of a *type*, which can be either read or write, a *size*, specifying the number of words to read or write, and an *address*. A multi-word request reads or writes its data from/to consecutive locations in the logical address space. Byte masking is supported for write-requests only, and addresses have to be byte-aligned. Requests are executed by the controller in order of arrival on a per-client basis, i.e., requests from the same client are never reordered, even though the DTL standard theoretically allows this. DTL interfaces are also used to connect components within the front-end; all white ports in Fig. 2.6 are DTL ports. The gray ports use non-DTL interfaces that are specific to the components they connect to. Commands and data passing through a pair of DTL ports experience a cycle of latency, so each pair represents a pipeline stage in the controller. Flow control is based on back-pressure by means of valid-accept flags in the DTL interfaces of the blocks.

2.3.1.2 Atomizer

The commands in a pattern are fixed at design time, and the controller hence always works at the same fixed *access granularity*, i.e., there is a specific number of bytes associated with a read or write pattern. When clients send requests into the memory controller, they are not necessarily of the same size as the access granularity. The *atomizer* resolves this inconsistency by splitting incoming requests into atomic service units called *atoms*. Access to the SDRAM is granted to clients by the arbiter on a per-atom basis. This allows clients to be preempted at the granularity of atoms, independently of the size of the requests they produce, which is a property we require to be able to bound the interference from each client without making assumptions about their behavior [2, 35]. The type of the atoms (read or write) is equal to the type of the request they are based on, but the amount of data that is associated with an atom is always equal to the access granularity of the memory controller, which typically ranges from 16 bytes up to 1 KiB, depending on its configuration. The atomizer concept was first shown in [32, 36], and we base our implementation on these works.

To make the *atomizer* suitable for use with an SDRAM, it enforces the address alignment of its outgoing requests to atom boundaries, and handles requests with sizes that are non-integer multiples of the atom size by padding and masking them where required. The atomizer is pipelined, such that the first stage acts as the *input buffer* for the front-end, quickly terminating logic paths leading from the clients into the controller, and allowing the overall design to run at a higher clock frequency. The configuration port on the atomizer allows its access granularity to be (re)configured at run time. The benefits and limitations of reconfiguration are explained in detail in Chap. 7. The atomizer uses the same data width as the client it is connected to.

2.3.1.3 Width Converter

The *width converter* accepts requests at the data width of the atomizer (generally 32-bits wide), and converts them to the width the back-end works at, which is typically larger. In essence, this is a common serial-to-parallel converter. Both the atomizer and width converter work on a streaming basis, i.e., they contain no data buffers apart from pipeline registers that break up the critical paths within the blocks. After width conversion, all clients use the same data width on their DTL interfaces.

2.3.1.4 Atom Queue and Delay Block

The *atom queue* holds incoming atoms until either all associated data is buffered (for write atoms), or enough space is available for the response (for read atoms). An atom is only eligible for scheduling once this buffering requirement is satisfied. Internal and individual buffering per client is necessary for two reasons

1. the SDRAM determines when data must be provided to and accepted from it on consecutive cycles, in accordance with the JEDEC specifications [7–12]. Clients are not guaranteed (or required) to produce or consume all data for an atom on consecutive cycles, and data must hence be buffered somewhere *internally* in the memory controller to ensure this requirement is always satisfied.
2. *Individual* queues per client are needed to avoid situations where clients occupy the shared resource before they are capable of reading/writing a complete atom. If a shared queue would be used, then a noncooperative (blocking) client could occupy the queue indefinitely and stall the resource as a result. This would break the isolation between clients, because preempting (and flushing out) an ongoing transaction is not supported. Using individual queues, a noncooperative client can only indefinitely occupy its own queue, which is not disruptive for others.

Delay blocks wrap the atom queues. Each delay block can be configured such that the data consumption and production behavior of the SDRAM is equal to a specific Latency-rate (\mathcal{LR}) curve [37] from the client’s point of view. It achieves this by manipulating flow-control signals that govern the acceptance of incoming atoms and their data, and the time at which responses are released by the atom queue. In essence, it delays each response to its *Worst-Case Response Time* (WCRT), as specified by its \mathcal{LR} guarantee. This is a generalization of the *Logical Execution Time* (LET) idea [38, 39], which uses a single number to represent the WCRT. Delay blocks were introduced in [32], and we use the same design here. An introduction on \mathcal{LR} servers is provided later in Sect. 2.4.1.

2.3.1.5 Resource Bus

The *resource bus* grants one client at a time access to the SDRAM back-end. Arbitration decisions are made by a predictable *arbiter* (e.g., any arbiter in the class of latency-rate servers [37]), which schedules one of the eligible atoms from the atom queues to be processed by the back-end. Each scheduling decision corresponds to a single atom, allowing for fine-grained interleaving of atoms from different clients. The resource bus drives the pace at which scheduling decisions are made by requesting scheduling decisions from the arbiter. It can be configured to do that strictly periodically, or on-demand, e.g., when the back-end indicates it is ready to accept new atoms.

Various predictable arbiters are supported within the associated design flow [40]. One option is a reconfigurable TDM arbiter, described in detail in Chap. 7. Other available arbiter types are round-robin [41] and *Credit-Controlled Static-Priority* (CCSP) [42]. The arbiter type is chosen at design time. Other arbiter settings, like TDM slot allocations or the priorities in CCSP for example, are configurable at run-time through the configuration bus. To increase the clock frequency at which the resource bus can be synthesized, the arbitration between clients takes place in a separate pipeline stage.

Although the communication interface between the front-end and back-end uses DTL signals, its flow-control semantics [43] are slightly different compared to the other ports. Once a request for an atom is handed to the back-end, the front-end is required to be able to deliver all the associated data for a write atom *whenever the back-end demands it*. Similarly, the front-end has to accept data from a read atom *whenever the back-end offers it*. Both of these requirements are satisfied by the eligibility test that the atom queues perform before they forward requests (see Sect. 2.3.1.4).

To reduce its complexity, Fig. 2.6 only contains two memory clients. However, up to 16 ports can be instantiated automatically by the associated design flow if required. Section 2.6 evaluates the effect of varying the number of ports on the hardware resource usage.

2.3.2 SDRAM Back-End

The *SDRAM back-end* receives atoms from the resource bus that consist of a type (read/write) and a logical address. Its main function is to select patterns from the *pattern memory*, and to transfer their commands to the PHY, translating atoms into command sequences. It has to ensure that the timing constraints between the commands are satisfied by only issuing valid pattern sequences (Fig. 2.4). It accepts one atom at a time, and based on the type (read or write) and the type of the previously executed pattern, it executes one or two patterns:

1. A write pattern, if the previously executed pattern was a write, refresh or idle pattern, and the current atom is a write.
2. A RTW pattern followed by a write pattern, if the previously executed access pattern was a read, and the current atom is a write.
3. A read pattern, if the previously executed pattern was a read, refresh or idle pattern, and the current atom is a read.
4. A WTR pattern followed by a read pattern, if the previously executed access pattern was a write, and the current atom is a read.

Figure 2.7 shows a pattern execution example. The time between scheduling decisions, or *Scheduling Interval (SI)*, is variable as a result of this behavior, both across

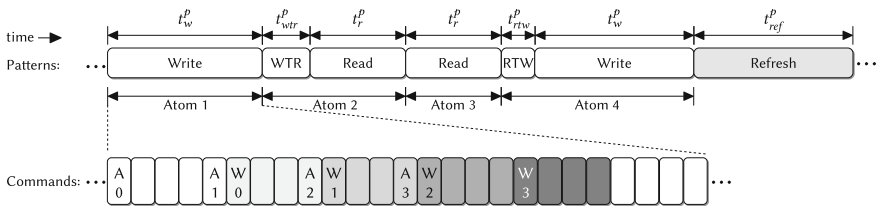


Fig. 2.7 An example of the order in which patterns may be executed. The *shading* on the commands corresponds to bursts of data to different banks

atom types and for atoms of the same type, i.e., a write atom could require a RTW and write pattern, or only a write pattern, as shown in Fig. 2.7. In the continuation of book, we use the following terminology for the pattern lengths: t_r^p , t_w^p , t_{wtr}^p and t_{rtw}^p represent the read, write, write-to-read and read-to-write pattern lengths, respectively. Additionally, the refresh pattern length is denoted by t_{ref}^p .

In contrast to [25], which uses a hard-coded finite-state machine to implement the required functionality, we use a flexible reconfigurable back-end, which is shown in detail in Fig. 2.8. An incoming atom first arrives at the *pattern selector*. It generates an index for the *pattern Look-Up Table (LUT)* based on the atom type (read or write) and the previously executed pattern type. The index represents the type of pattern that should be executed (the basic pattern types are mentioned in Sect. 2.2). There may be more than one pattern set available in the *pattern memory*. An optional *offset* can be added to the pattern index to switch to a different pattern set. Note that this offset is not selectable per atom, but instead is part of the overall back-end configuration. It can be used to switch between configurations in different use-cases, as further explored in Chap. 7.

The pattern LUT contains the starting addresses and the number of commands of all patterns in the pattern memory. Its output is used by the *command player* to read commands from the pattern memory. Both the pattern LUT and the pattern memory are exposed to the resource manager through the configuration bus and are thus reconfigurable.

The pattern memory is conceptually implemented as a simple SRAM memory, containing a representation of an SDRAM command and optional bank at every

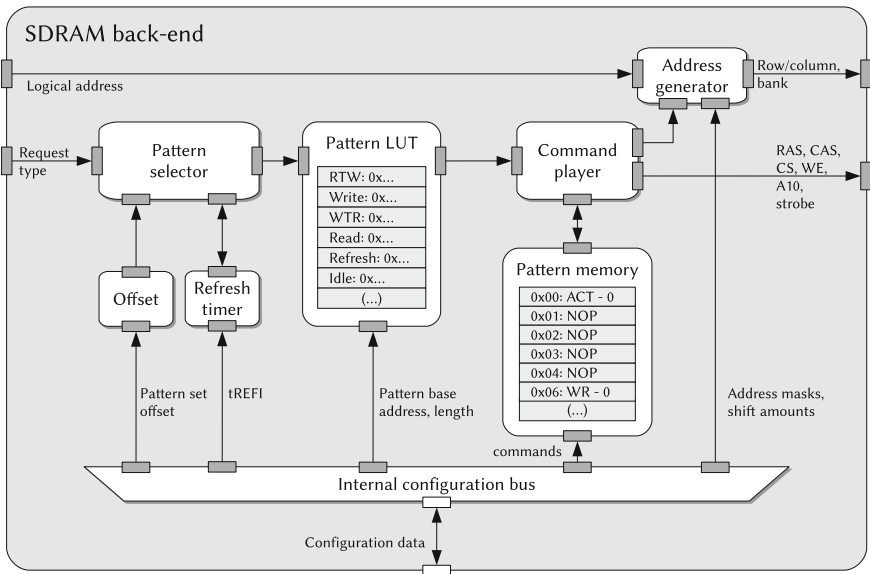


Fig. 2.8 SDRAM controller back-end

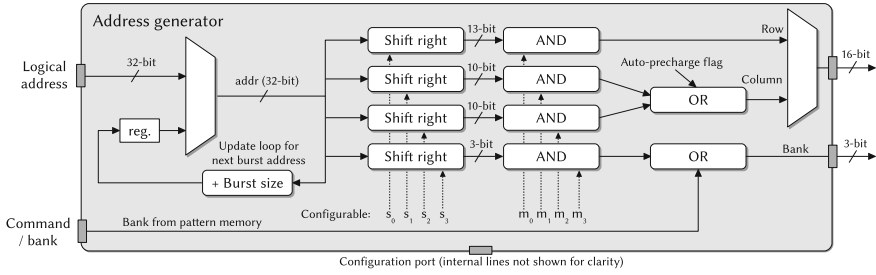


Fig. 2.9 Address generator. Both the shift amounts (s_0 – s_3) and the masks (m_0 – m_3) used by the and-operators are configurable. (The **and**-operators and **or**-operators are bitwise.) The sizes of the row, column, and bank components correspond to the ML605 memory (Appendix B)

entry. The *command player* increments the command address every clock cycle, and triggers a new pattern selection when the current pattern ends, while also converting the commands into control signals for the PHY. Section 2.5 discusses the specific implementation in our FPGA prototype.

The *address generator* translates a logical address to the corresponding bank, row and column (physical) address elements (Fig. 2.9). The command player controls the address generator such that the correct address is given to the PHY at the right time, i.e., the row address when activating and the column address during read or write commands. Auto-precharge flags have to be included in the column address of the associated read or write command. The bit-position (*loc*) of this flag depends on the SDRAM type. Commands in the pattern memory are directed to a specific bank. The address of that bank is referred to as *cmd.bank*, and is included into the address calculation. The address generator has four configurable masks (m_0 – m_3) and shift amounts (s_0 – s_3) through which the logical to physical memory-mapping function can be selected. When combined with the or-operators, the following physical addresses are generated:

$$\text{row} = (\text{addr} \gg s_0) \text{ and } m_0 \quad (2.2)$$

$$\text{column} = ((\text{addr} \gg s_1) \text{ and } m_1) \text{ or } ((\text{addr} \gg s_2) \text{ and } m_2) \text{ or } (\text{autoPreFlag} \ll \text{loc}) \quad (2.3)$$

$$\text{bank} = ((\text{addr} \gg s_3) \text{ and } m_3) \text{ or } \text{cmd.bank} \quad (2.4)$$

Each atom only has one logical address. This address is registered (in the *reg.* block in Fig. 2.9) and incremented after each read or write command to generate the address for the next burst (in case the atom consists of more than one burst). Section 3.2.5 shows how to configure the address decoder, based on the selected memory map.

The final block to consider is the *refresh timer*, which is responsible for periodically inserting refresh patterns into the SDRAM. It consists of a cycle counter with a configurable threshold value. When the counter reaches the threshold, it resets to zero and a refresh is scheduled as soon as the currently executing pattern finishes. Automatic refresh can optionally be disabled to allow manual refresh schemes, as described in [24, 44] for example, to be used.

2.3.3 *PHY*

The PHY handles the physical I/O connections to the SDRAM module. It acts as a level of abstraction from the circuit-level details of the SDRAM, and offers a generic interface to the back-end. Several companies create PHY IPs, and specifications like DFI [45], for example, standardize the interface they expose. A PHY is inherently specific to the SDRAM generation it connects to, although there is often a fair amount of logic that can be reused across generations [46]. Since the FPGA prototype is meant for a DDR3 memory, the following description of the PHY functionality is also DDR3 specific.

Each byte on an SDRAM interface is individually clocked with a strobe signal, and both the byte lanes and strobe signals are bidirectional, i.e., the same wires are used for both reading and writing. At initialization, the PHY runs through a calibration procedure (called read-leveling) to determine the time offset between these strobe signals and the presence of valid data on the byte lanes when reading from the memory. Each byte can have a different offset, based on the wire layout of the PHY and its connection to the SDRAM chips. After calibration, the PHY can compensate for these offsets appropriately by inserting delays, such that all the bytes from a single memory word are aggregated and are forwarded to the back-end synchronously. A similar timing-offset issue exists for data flowing into the SDRAM (write-leveling), and it is solved in an analogous manner.

The PHY also configures the SDRAM by programming the mode registers in the device. In this work, we assume that both the calibration and the configuration finish in a bounded amount of time. Since this initialization process happens only once (after the SoC comes out of reset), it can be regarded as part of the boot process and has no further influence on the real-time analysis of the controller, assuming there are no real-time requirements on the boot time.

The additional delay that the PHY introduces after calibration, on the other hand, has to be included in the worst-case response time of the memory controller (in δ_{PHY}^b), as we later discuss in Sect. 2.4.2. Since the hardware in the PHY can only compensate for a limited byte-level offset (in the order of a few cycles), we use this maximum compensation as a worst-case bound for the contribution of the PHY to the WCRT.

2.3.4 *Reconfiguration Infrastructure*

The configuration bus allows various memory-mapped registers to be programmed by a configuration host. The host does this by sending (DTL) configuration requests to the reconfigurable components. Requests are generated by the driver code of the memory controller running on the configuration host.

All components in the front-end can be pre-configured with a design-time selected default configuration after reset, allowing potential early (predictable) access to the

back-end while the rest of the system is still booting. The back-end starts out with an empty pattern memory, and hence needs to be configured before it can be used. A small ROM containing a minimal back-end configuration can be added in case a functioning memory controller is required before the configuration host is active in the system.

2.4 Worst-Case Performance Analysis

This section discusses the worst-case performance analysis of the SDRAM controller architecture that was presented in the previous section. The general structure we apply is similar to that in [1], and relies on a *Latency-rate* (\mathcal{LR}) server abstraction (Sect. 2.4.1) of the controller's behavior. We present a word-level performance model that shows in detail how (hardware) pipelining impacts the analysis. The two performance metrics we derive for the memory controller are

1. *Worst-case bandwidth* (b_{wc}), which specifies how much bandwidth the SDRAM delivers in the worst-case when connected to our controller (assuming there is always at least one request to serve). The worst-case bandwidth is distributable amongst the different ports on the front-end, and
2. *WCRT of a request for a client connected to the front-end*.

The analysis is split in two parts. First, we look at the performance of the back-end in Sect. 2.4.2, which we characterize with a \mathcal{LR} server. Second, we repeat that effort for the front-end in Sect. 2.4.3. Finally, we derive the WCRT of the combination of the back-end and front-end in Sect. 2.4.4 by concatenating their two respective \mathcal{LR} servers.

2.4.1 Latency-Rate Servers

To characterize the (predictable) performance of the memory controller, we rely on a \mathcal{LR} server abstraction [37]. A \mathcal{LR} server guarantees a (client specific) *minimum rate*, ρ , after a *maximum service latency*, Θ , to each of its clients. When the \mathcal{LR} abstraction is applied to a memory controller, the rate (ρ) maps to a certain bandwidth (bytes/second). The service latency is expressed in a unit of time (seconds or cycles), and it intuitively captures the initial latency a client experiences before the server can sustain the guaranteed rate. This linear service guarantee has to (lower) bound the amount of data that can be transferred during any interval. We proceed with a brief intuitive introduction of the properties of \mathcal{LR} servers.

Figure 2.10 plots the *service bound* as a thick black line, given the example *requested service line* (dotted line). A \mathcal{LR} guarantee is conditional and *only applies if the client requests enough service to keep the server busy*. This is captured by the concept of *busy periods*, which are periods where a client requests at least as much

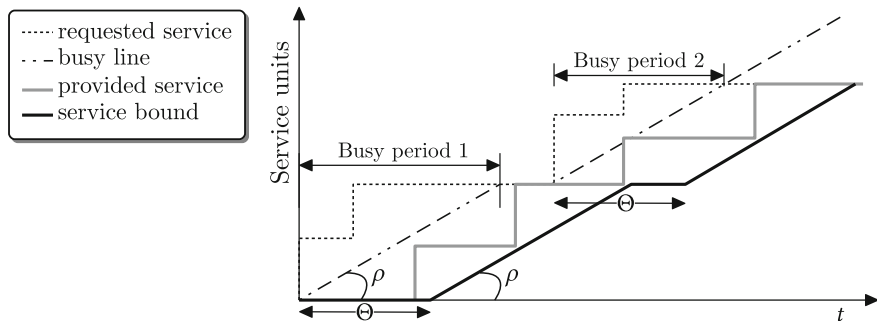


Fig. 2.10 A \mathcal{LR} server and its associated concepts

service as it has been allocated on average (ρ). In Fig. 2.10, the client is busy as long as the requested service line is above the *busy line*, and hence the start of the first busy period is marked by the first intersection of the dash-dotted and dotted line (at $t = 0$). It ends at the second intersection with the dash-dotted line. The second busy period starts when the requested service exceeds the busy line again, which is equivalent to one or more new requests entering the memory controller. After Θ units of time have passed since the start of this second busy period, the server once again guarantees the ρ in the second busy period.

The service bound line is equal to the busy line delayed by Θ , and hence starts Θ after the start of the busy period and increases with rate ρ . The *provided service* is always greater than or equal to the service guarantee, since it follows the actual-case performance, and not the worst-case performance. An example of what the provided service curve could look like is drawn in Fig. 2.10 with the thick gray line. The service bound is maximal if the client continuously remains busy, i.e., if the client requests service at a sufficiently high rate ($\geq \rho$).

Note that requests arrive instantaneously, as shown by the discrete jumps in the requested service line. A read request is considered to instantaneously arrive once the request arrives in the atom queue and there is space for the corresponding response. A write arrives when its last data word arrives in the atom queue. The service bound and busy line are fractional, and therefore shown as continuous curves. The provided service for a memory controller is discrete at the level of words, bursts, or atoms, whichever is preferred (in Sect. 2.4, we use a word-level characterization).

2.4.2 Back-End Performance

The *back-end performance* refers to the performance the SDRAM controller would deliver if it was not shared amongst multiple clients. We characterize back-end performance with a \mathcal{LR} server with parameters (Θ_{be}, ρ_{be}) . The server describes the behavior of the interface at the dotted line in Fig. 2.11 (annotated with “back-end performance”).

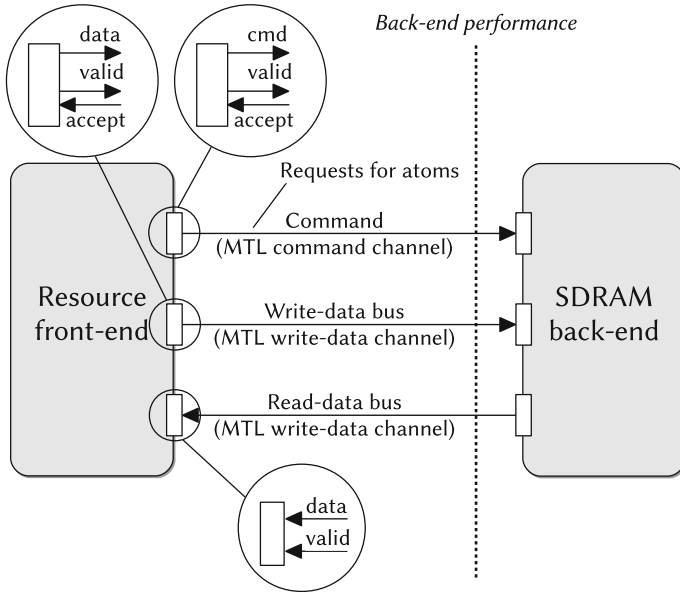


Fig. 2.11 The interface characterized by the back-end performance. The call-outs on the MTL channels show the relevant groups of wires they consist of

The requested service increases by one atom worth of bytes when the request for the atom is offered by the front-end to the back-end. For simplicity, we assume that the back-end runs at the same clock frequency as the SDRAM. It has dedicated read and write data buses (the PHY later serializes writes and reads onto the bidirectional SDRAM data bus). Each of these buses is twice as wide as the IW of the SDRAM, such that the difference in data rate between the controller and SDRAM (SDR vs. DDR) is compensated for. The \mathcal{LR} server gives guarantees on when a specific amount of data is available/consumed by the back-end. This amount corresponds to the sum of the number of handshakes on the read (valid flags) and the write (valid/accept pairs) data buses.

First, we evaluate the overhead of refresh in Sect. 2.4.2.1. Then we focus on worst-case bandwidth, b_{wc} , in Sect. 2.4.2.2. The worst-case analysis of memory patterns in terms of bandwidth has been extensively discussed in related work [1, 47, 48]. We apply the same procedure to derive our results as described in those works, but for convenience and completeness provide a small summary of it in this section. No assumptions are made on the order in which read and write atoms are given to the back-end. This decouples the inter-client scheduling from the analysis of the back-end, simplifying both. Later, in Sect. 2.4.2.3, we determine the value of ρ_{be} and Θ_{be} such that the \mathcal{LR} server with parameters (Θ_{be}, ρ_{be}) conservatively bounds the behavior of the back-end.

2.4.2.1 Refresh

An SDRAM needs to be refreshed once every REFI cycles on average (Appendix B). During a refresh, the SDRAM is unavailable to clients, which impacts the worst-case performance. Most works [1, 29, 30, 49] assume refresh is triggered asynchronously with respect to the inter-client scheduling by an internal timer in the controller, and has precedence over requests from clients. Refresh then impacts both bandwidth and response time.

The *refresh efficiency* describes the refresh-related bandwidth reduction when such a timer-based refresh mechanism is used. It is defined as one minus the fraction of time spent on refreshing, which for a pattern-based controller is equal to

$$e_{ref} = 1 - \frac{t_{ref}^p}{REFI} \quad (2.5)$$

where t_{ref}^p is the length of the refresh pattern as defined earlier. The refresh efficiency ranges from 0.96 to 0.99 for the devices we evaluate in this book, and hence only a small fraction of all requests is actually affected by a refresh. In related works, refresh has been incorporated in the worst-case analysis in several ways.

Busy-Period-Level Refresh

Each request might have to wait for a refresh. A conservative request-level WCRT therefore incorporates at least one refresh pattern. When the worst-case analysis is based on \mathcal{LR} servers, like in [1], then it has to account for at least one refresh at the start of a busy period, which may span many requests.

Application-Level Refresh

Other approaches, like [29, 30, 49, 50], let go of the notion of a conservative request-level WCRT, and instead derive an application-level bound. First, the *Worst-Case Execution Time (WCET)* of an application interacting with the SDRAM is determined, without accounting for refresh. Based on this, the maximum number of interfering refreshes is found by dividing this number by REFI. A cost is assigned to each of these refreshes, and added to the application's WCET. This can lead to smaller application-level WCET bounds compared to [1], as shown in [50], which also does this.

Manual Refresh

Finally, there is an approach that we refer to as *manual refresh* [24, 26, 44]. Activating and precharging a row effectively refreshes it, so data is retained as long as each row is visited at regular intervals. Controllers that use manual refresh do not have an internal timer, but instead have a *refresh client* that cycles over all rows, activating and precharging them.

When manual refresh is used, e_{ref} can be set to 1. The cost of refresh is instead taken into account when bandwidth is set aside for the refresh client in the front-end. Manual refresh is less efficient [44, 51] than the (built-in) REF command, because

it refreshes fewer rows per cycle, and hence the fraction of the available bandwidth that needs to be reserved for the refresh client is larger than $1 - e_{ref}$. However, the number of consecutive cycles for which the SDRAM is unavailable during a manual refresh can be smaller, which generally reduces the WCRT of a single request.

For the remainder of this book, we ignore refresh at the level of busy periods, and assume it is taken into account at a later (application-level) stage, as is done in [29, 30, 49, 50]. Akesson and Goossens [1] shows how to include refresh at the busy-period level for a pattern-based controller for the interested reader.

2.4.2.2 Calculating Worst-Case Bandwidth

The worst-case bandwidth delivered by a pattern set is a function of its pattern lengths, the clock frequency, the amount of data that is transported per read/write pattern (the access granularity, AG), and the refresh period.

The *worst-case bandwidth* (b_{wc}) is a lower bound on the average amount of bytes transported across the data bus per unit of time. This bound is valid during a busy period, for every interval starting Θ_{be} after the start of that busy period. To find b_{wc} , we need to identify the pattern sequence allowed by the pattern state machine that has the lowest average data transfer rate (excluding sequences that include idle patterns). This could imply continuously reading or writing, transporting AG bytes per pattern, or constantly switching between reads and writes, transporting $2 \cdot AG$ bytes per pair of read and write patterns. Note that in the latter case switching patterns are required, reducing the efficiency. All these pattern sequences are periodically interrupted by refreshes, and hence we multiply with the refresh efficiency (e_{ref}) (see Sect. 2.4.2.1 for its definition). Finally, multiplying with the command clock frequency f to obtain a bytes/seconds metric, leads to the following worst-case bandwidth equation:

$$b_{wc} = e_{ref} \cdot AG \cdot \min \left(\frac{1}{t_r^p}, \frac{1}{t_w^p}, \frac{2}{t_w^p + t_r^p + t_{wtr}^p + t_{rtw}^p} \right) \cdot f \quad (2.6)$$

The *peak bandwidth* (b_{peak}) that an SDRAM would theoretically deliver if its data bus was fully utilized is obtained by multiplying the data clock frequency by the interface width in bytes (IW). The data clock frequency is $2 \cdot f$ for double data rate memories

$$b_{peak} = 2 \cdot f \cdot IW \quad (2.7)$$

The ratio of the worst-case bandwidth and the peak bandwidth is referred to as the *memory efficiency* (e) of a pattern set

$$e = \frac{b_{wc}}{b_{peak}} \quad (2.8)$$

The memory efficiency shows how well a certain pattern set performs with respect to the theoretical maximum bandwidth of a memory device.

2.4.2.3 Calculating Back-End Service Latency

The back-end service latency (Θ_{be}) has to be chosen such that b_{wc} bounds the bandwidth after this latency has passed since the start of each busy period. We first consider the scenario in which the largest amount of time passes between the request for an atom (requested service) by the front-end and the associated data handshakes (provided service), since Θ_{be} necessarily has to include this time. Figure 2.12 shows the relation between the variables we introduce, and the events they relate to.

First, we account for the latency related to *pipeline stages* in the hardware, both in the back-end and the PHY. We use the symbol δ to represent these latencies.

1. δ_{be}^f on the request (forward) path: cycles that a request for an atom spends in pipeline stages in the back-end, before the back-end begins to issue commands to the PHY. We assume write data words traversing the back-end experience the same latency.

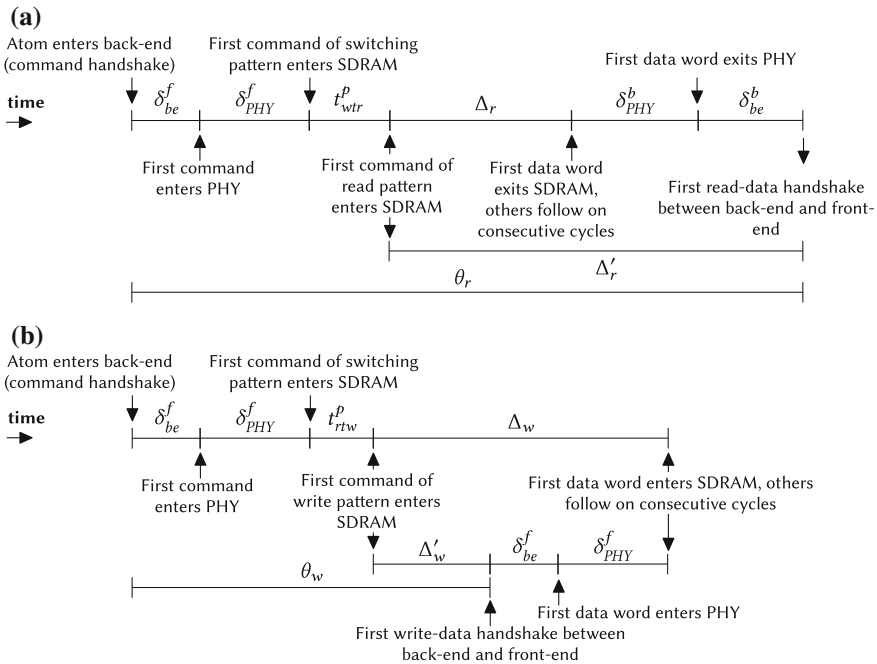


Fig. 2.12 Latency experienced by a read or write atom arriving at an idle back-end at the start of a busy period. **a** Read atom. **b** Write atom

2. δ_{PHY}^f on the request path: cycles that a command or write data word spends in pipeline stages in the PHY before it is issued to the SDRAM.
3. δ_{PHY}^b on the response (backward) path: cycles that a read data word spends in pipeline stages in the PHY before it emerges on its interface to the back-end.
4. δ_{be}^b on the response path: cycles that a read data word spends in pipeline stages in the back-end before it emerges on the back-end interface.

We combine the pipeline latencies on the forward and backward paths into a single variable, since we do not require them individually in the continuation of the analysis:

$$\delta_f = \delta_{be}^f + \delta_{\text{PHY}}^f \quad (2.9)$$

$$\delta_b = \delta_{be}^b + \delta_{\text{PHY}}^b \quad (2.10)$$

To account for the time between the start of a pattern and the actual transfer of data on the SDRAM data bus, we use the symbol Δ .

1. Δ_r is the number of cycles between the first command of a read pattern entering the SDRAM, and the emergence of the first word of read data on the SDRAM data bus. It is the sum of the relative cycle of the first RD command in the read pattern with respect to the start of that pattern, and the RD-to-data latency (usually RL) of the SDRAM.
2. Δ_w is the number of cycles between the first command of a write pattern entering the SDRAM, and the transfer of the first word of write data by the SDRAM data bus. It is the sum of the relative cycle of the first WR command in the write pattern with respect to the start of that pattern, and the WR-to-data latency (usually WL) of the SDRAM. Relative to this number, write data handshakes on the back-end interface happen δ^f cycles earlier, under the assumption that commands and data are equally deeply pipelined.

Both for Δ_r and Δ_w we assume that all data associated with a pattern exits/enters the SDRAM on consecutive cycles.²

We define Δ'_r and Δ'_w as the offset from the start of the pattern (first command enters the SDRAM) until data handshakes happen on the back-end interface. For reads, this happens later than Δ_r , since they generate data on the response path, while for writes it happens earlier than Δ_w on the request path

$$\Delta'_r = \Delta_r + \delta^b \quad (2.11)$$

$$\Delta'_w = \Delta_w - \delta^f \quad (2.12)$$

Now, we can describe the number of cycles after which service starts for a read or write atom arriving at the start of a busy period as θ_r and θ_w , respectively

²If this is not the case, i.e., when there are bubbles in the transfer, compensation is required. The number of additional idle cycles should then be added to Δ_r and Δ_w .

$$\theta_r = \delta^f + t_{wtr}^p + \Delta'_r = t_{wtr}^p + \Delta_r + \delta^f + \delta^b \quad (2.13)$$

$$\theta_w = \delta^f + t_{rtw}^p + \Delta'_w = t_{rtw}^p + \Delta_w \quad (2.14)$$

Analogously to Eq. (2.6), we have to conservatively cover three scenarios when we determine (Θ_{be}, ρ_{be}) : continuously reading, writing, or switching between reads and writes. These three scenarios are illustrated in Figs. 2.13, 2.14 and 2.15, respectively. The figures consist of two parts. The bottom half is a gantt chart of the activity in various parts of the controller. When a request for an atom is offered to the back-end by the front-end, a block is drawn on the *atom in line*. The commands that the PHY issues to the SDRAM are drawn as blocks on the *SDRAM command bus line*, and the corresponding pattern is drawn above it on the *pattern line*. Read and write commands result in data transfers on the SDRAM data bus after a certain latency. The blocks on the *SDRAM data bus line*, represent one word of data on this bus. Note that two words can be transferred per clock cycle for a DDR memory, and hence the blocks on the SDRAM data bus line are half as wide as on the command bus. We assume the rate difference is compensated for by the double width of the back-end data buses, as mentioned earlier. Finally, the *back-end (read/write) lines* represent handshakes on the data buses that the back-end exposes to the front-end. Each block on these buses corresponds to a 1 word increase of the *provided service curve* on the top half of the figures. Based on ρ_{be} in each scenario, the *requested service curve* and *busy line* are drawn. Each increase of the requested service corresponds to the arrival of an atom (an atom is worth 4 words in this example). Atoms arrive as late as possible within a busy period, which leads to the minimum provided service.

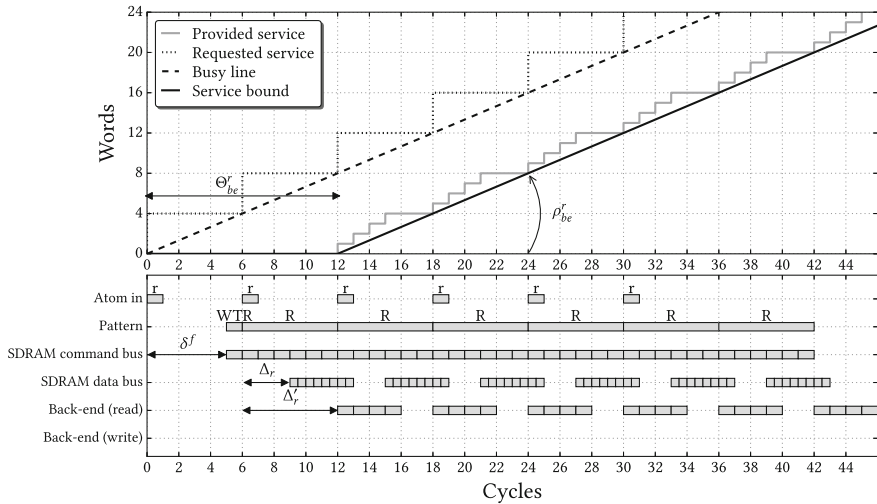


Fig. 2.13 Worst-case back-end behavior for continuous reads. In this (fictional) example, we used: $t_r^p = 6$, $t_w^p = 8$, $t_{rtw}^p = 3$, $t_{wtr}^p = 1$, $\Delta_r = 3$, $\Delta_w = 2$, $\delta^f = 5$, $\delta^b = 3$, and each atom is worth 4 words. To simplify the drawing, we assume $e_{ref} = 1$

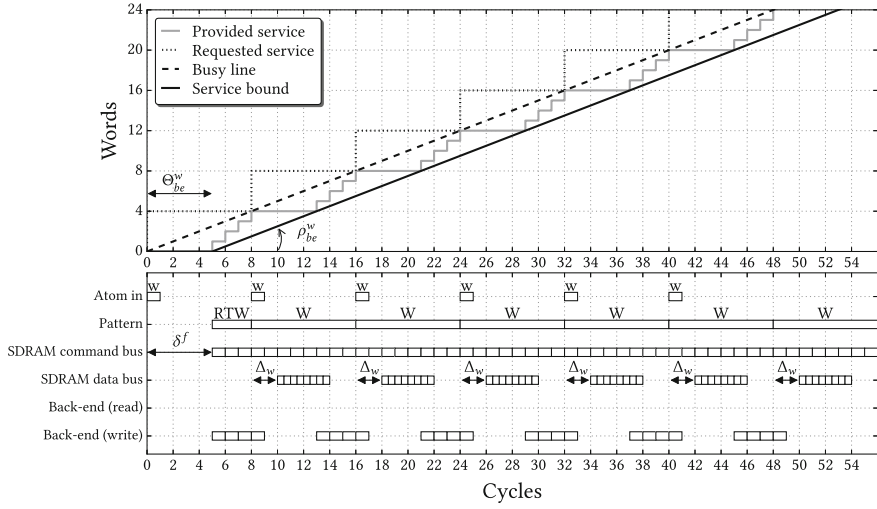


Fig. 2.14 Worst-case back-end behavior for continuous writes, using the same parameters as Fig. 2.13

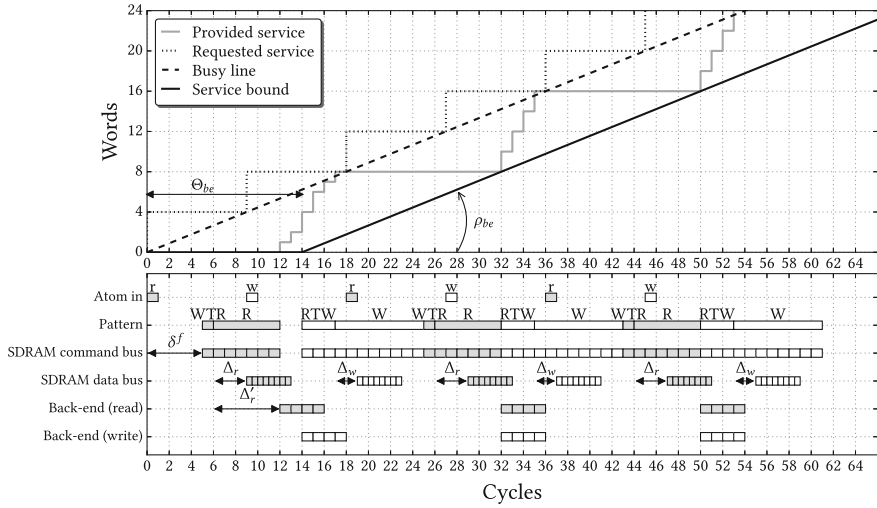


Fig. 2.15 Worst-case back-end behavior for interleaved read/write atoms, using the same parameters as Fig. 2.13

In one of these three scenarios (the worst-case, Fig. 2.15 for the particular set of parameters we used to draw the figures), ρ_{be} is equal to b_{wc} . Which scenario this is, depends on the length of the patterns. We want to make no assumptions on the order of reads and writes, and hence select:

$$\rho_{be} = b_{wc} \quad (2.15)$$

In this scenario, the worst-case distance between the “atom in” blocks in Figs. 2.13, 2.14 and 2.15, the *Worst-Case Inter-Atom Time (WCIAT)*, is given by

$$\text{WCIAT} = \max \left(t_r^p, t_w^p, \frac{1}{2} \cdot (t_w^p + t_r^p + t_{wtr}^p + t_{rtw}^p) \right) \quad (2.16)$$

It is proportional to the slope of the busy line, and shows at what intervals the requested service line has to increase to remain within a busy period.

The number of commands that are executed for *one specific* atom can be larger than WCIAT. For example, if the first argument of the max-term in Eq. (2.16) dominates, then an atom that triggers a switch from writing to reading takes $t_{wtr}^p + t_r^p \geq t_r^p$ cycles. WCIAT is the average time the back-end spends per atom when serving a worst-case sequence of atoms. Equation (2.6), which calculates b_{wc} , uses the same duration. We call the maximum time between two atom scheduling decisions the *Worst-Case Scheduling Interval (WCSI)*:

$$\text{WCSI} = \max (t_{rtw}^p + t_w^p, t_{wtr}^p + t_r^p) \quad (2.17)$$

When $\text{WCSI} > \text{WCIAT}$, the back-end can alternate between generating one atom worth of service quicker than and slower than WCIAT, respectively. This behavior is drawn in Fig. 2.16 as the *atoms completed* line. The graph starts at $\max(\theta_r, \theta_w)$, i.e., at the time where we know the provided service starts to increase when serving only read or write atoms. If read and write atoms are mixed, we must ensure that

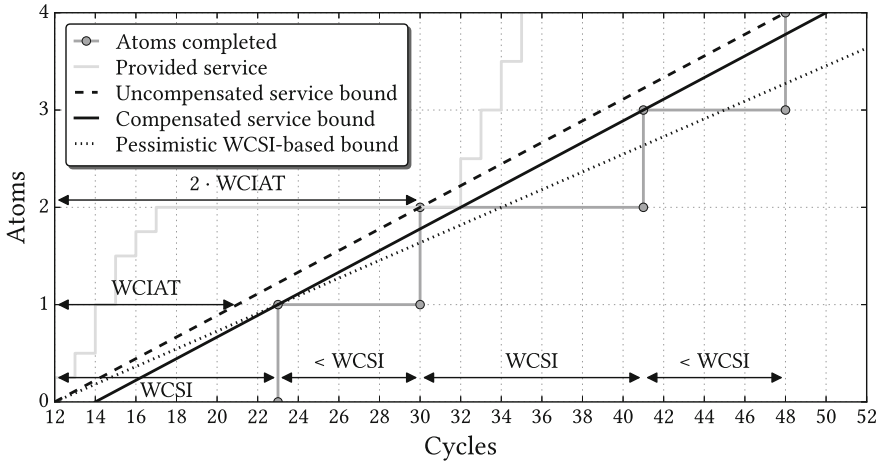


Fig. 2.16 Demonstration of latency compensation for WCSI, using the same parameters as Fig. 2.13. The compensated service bound is conservative in cycles 30 and 31, while the uncompensated service bound is not. Note that the x-axis starts at $\max(\theta_r, \theta_w)$

the time required for each possible pair of atoms is conservatively bounded by the (average) WCIAT. To achieve this, we add $\text{WCSI} - \text{WCIAT}$ to Θ_{be} . This effectively shifts the start of the rate phase of the server forward in time to make the service guarantee conservative. This amount of time can be seen in Fig. 2.16 as the 2-cycle difference between the compensated service bound and the uncompensated service bound. The figure also shows that a bound based on atoms that always take WCSI cycles is overly pessimistic if $\text{WCSI} > \text{WCIAT}$. Finally, the expression for Θ_{be} is equal to:

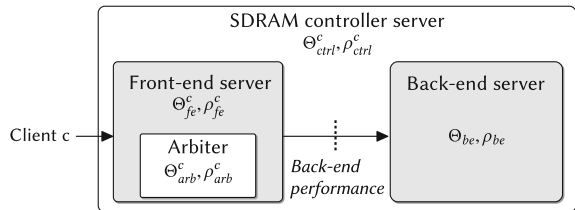
$$\Theta_{be} = \text{WCSI} - \text{WCIAT} + \max(\theta_r, \theta_w) \quad (2.18)$$

2.4.3 Front-End Performance

Clients observe a certain performance from the memory controller through the port by which they are connected to it. The arbiter in the front-end regulates which clients' atom is processed by the back-end. Each client has an abstract *allocation* within the arbiter that for most intents and purposes can be seen as a specific fraction of the total shared resource time. We assume that the allocation of client c in the arbiter can be described with two new \mathcal{LR} parameters, $(\Theta_{arb}^c, \rho_{arb}^c)$. These parameters are normalized, such that ρ_{arb}^c represents the fraction of the total server bandwidth that a client receives after it has waited for Θ_{arb}^c scheduling slots. Because the arbiter schedules atoms, each scheduling slot represents an atom-sized access.

We always assume a predictable arbiter is used within the memory controller, like TDM, round-robin [41] or CCSP [42]. [37] shows how to derive the \mathcal{LR} parameters for various popular arbiter types, [42] focuses on CCSP, and [52, 53] extensively discuss TDM arbiters in the context of \mathcal{LR} servers. For the purpose of this book, we only need to look at the details for TDM arbiters, which is done later in Sect. 7.4.1. All these arbiters guarantee that the allocated fraction of back-end performance is always visible and usable by clients, even during worst-case interference from other clients. The guarantees that our controller gives to a client are (solely) based on this (guaranteed) fraction of the back-end performance (budget), which is hence not dependent on the behavior of other clients. This implies that the memory controller offers predictable performance to a client.

Fig. 2.17 The \mathcal{LR} server describing the memory controller's performance is the concatenation of the front-end server and the back-end server



We characterized the front-end for client c as another \mathcal{LR} server with parameters $(\Theta_{fe}^c, \rho_{fe}^c)$. ρ_{fe}^c represents the bandwidth that is allocated to the client.

$$\rho_{fe}^c = \rho_{arb}^c \cdot \rho_{be} \quad | \quad 0 < \rho_{arb}^c \leq 1 \quad (2.19)$$

Intuitively, we can see that if $\rho_{arb}^c = 1$, the client has the full back-end at its disposal.

Finally, we de-normalize Θ_{fe}^c such that it is expressed in clock cycles instead of scheduling slots. We do this by multiplying with the duration of such a slot in the back-end. We can use WCIAT for this, since the back-end \mathcal{LR} server is guaranteed to process at least one atom per WCIAT once Θ_{be} has passed. Additional pipeline stages in the front-end, on the forward and backward path, are represented by δ_{fe} :

$$\Theta_{fe}^c = \lceil \Theta_{arb}^c \rceil \cdot \text{WCIAT} + \delta_{fe} \quad (2.20)$$

2.4.4 Worst-Case Response Times

A client uses the concatenation of its front-end server and the back-end server. When two \mathcal{LR} servers are concatenated, a single server equivalent has a latency equal to the sum of latencies of the individual servers, and the minimum of their rates [37]. We use $(\Theta_{ctrl}^c, \rho_{ctrl}^c)$ to represent the combined server (Fig. 2.17)

$$\Theta_{ctrl}^c = \Theta_{fe}^c + \Theta_{be} \quad (2.21)$$

$$\rho_{ctrl}^c = \min(\rho_{fe}^c, \rho_{be}) = \rho_{fe}^c \quad (2.22)$$

The WCRT of a request is defined as the maximum time difference between the arrival of the request in the controller and the departure of the response. Intuitively, the WCRT of a request can be read directly from the \mathcal{LR} curve for the client, as the difference between the time at which the request arrives (i.e., where the requested service increases with one request worth of service), and the time at which the service bound reaches the same vertical height (see Fig. 2.15 for example). \mathcal{LR} guarantees are dependent on the client's (prior) behavior (the number of outstanding requests, and when they arrived), and because of that, the WCRT cannot be described as a single simple number, contrary to what we did earlier with the worst-case bandwidth. Instead, each requests may have its own WCRT.

The \mathcal{LR} server that describes a client's memory performance can be included as a component in a larger analysis model to validate the client's requirements. A general outline of this process can for example be found in [2, 54], which use the dataflow [55] model of computation for this purpose. In this context, it is not useful or required to define a single WCRT that is valid for all requests.

Introducing additional assumptions can take the client's behavior out of the equation if this is really desired. Arguably, the most conservative option is to assume that each request starts a new busy period, for example, but this potentially introduces a large amount of undesirable pessimism into the performance analysis. In general, the WCRT of a number of outstanding requests with a total size s for client c is equal to:

$$\text{WCRT}(s) = \Theta_{ctrl}^c + \frac{s}{\rho_{ctrl}^c} \quad (2.23)$$

The remaining contributions of this book directly impact the back-end \mathcal{LR} server, but have little impact on the front-end server, since only δ_{fe} increases slightly due to the addition of a few extra pipeline stages, as explained in Sect. 2.3.1. Hence, we focus on the quantification of the back-end performance in Chap. 5, leaving the front-end (mostly) out the equation.

2.5 CompSOC Controller Instance

The proposed controller has been integrated into the CompSOC flow [40] in two different forms:

1. Transaction-level SystemC. This implementation is flexible in terms of the modeled SDRAM generation. The PHY is not included in this model.
2. Synthesizable *VHSIC Hardware Description Language* (VHDL), targeted at DDR3 devices on the ML605 [56] FPGA development board. A fully functioning PHY is included in the controller design, and hence both simulation with a VHDL simulator such as Modelsim, and actual runs on the FPGA hardware are enabled.

The SystemC model is aimed at prototyping controller features, and verification of its functional correctness. It can produce cycle-level accurate SDRAM command traces, which can for example be used to check for timing constraint violations, and/or power estimation through external tools, like DRAMPower [57] for example. Simulating the model offers superior visibility on the internal state of the controller compared to FPGA-based experiments, but is unfortunately 3–4 orders of magnitude slower.

The VHDL version of the controller for the ML605 board is called *Raptor*.³ This board contains a Virtex 6 FPGA (XC6VLX240T) from Xilinx, which is connected to a DDR3 SO-DIMM slot. The PHY of Raptor is generated by the Xilinx *Memory Interface Generator* (MIG) 3.6 tool [59], and uses an interface that closely resembles the DFI 2.1 standard [45].

³*Raptor* is a forced acronym for **r**econfigurable and **p**redictable **o**pen-page controller, and also short for Velociraptor, a genus of dinosaurs, and a type of Predator [58].

A small LUT in the pattern player converts the commands from the pattern memory into a 6-bit control field and a 3-bit bank field. The control field contains values for the standard RAS, CAS, CS and WE signals, and the value for the 10th address bit in the physical address, which is the auto-precharge flag location for DDR3 (as used earlier in Fig. 2.9). The final bit is reserved for a strobe signal that is specific to the used PHY (and not part of the DFI standard), and selects the desired data bus (read/write) direction. The 3-bit bank field specifies the bank for which the command is meant. The pattern memory is implemented using *Block RAM (BRAM)* resources on the FPGA.

The back-end of the controller runs at half the frequency of the SDRAM command clock, and sends two commands (and four data words) per clock cycle into the PHY to compensate for this difference. This degree of parallelism is needed because the FPGA fabric is relatively slow compared to the SDRAM device, which makes designing a controller that works at the native command rate infeasible [60]. The PHY eventually serializes the commands and data before sending them to the SDRAM. Note that this is common practice, and both the DFI standard and commercially available controllers [61] may provide this operating mode as an option.

The SDRAM slot of an ML605 by default contains a 512 MiB DDR3-1066 device [62] (speed grade 1G1), capable of running at a 533 MHz command clock, although later versions have started shipping with larger and slightly faster devices. Figure 2.18 shows how this memory is typically used. The SDRAM is under-clocked to run at 400 MHz to match it up to the attainable controller frequencies on the FPGA, effectively turning it into a DDR3-800 with the controller back-end running at 200 MHz. The full data bus width of the DIMM is 64 bits, but a user of the CompSOC flow has the option to synthesize a controller with a 32-bit interface (connecting only half of the data pins) to save synthesis time or to emulate memories with a smaller interface, at the cost of making only half the memory accessible.

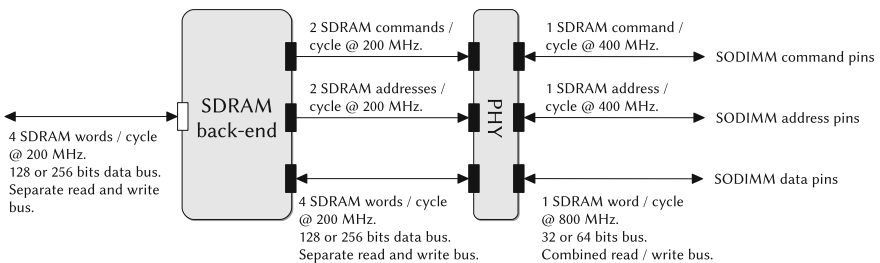


Fig. 2.18 Typical clock frequencies and data bus widths for Raptor

2.6 Evaluation

The goal of this section is to show that the VHDL implementation of our real-time memory controller is not prohibitively expensive in terms of hardware usage, and to give the reader a feeling for its relative size. Section 2.6.1 explains how the experiment was setup and why this specific setup was chosen, while Sect. 2.6.2 discusses the results.

2.6.1 Synthesis Setup

The demonstrated concepts in Raptor are technology agnostic, but its prototype implementation is bound to FPGA: the PHY is FPGA specific, and hence cannot straight-forwardly be synthesized to an *Application-Specific Integrated Circuit* (ASIC). Furthermore, the back-end generates two commands in parallel due to speed restrictions of the FPGA fabric. An ASIC implementation would be significantly different, primarily in terms of the high-speed I/O implementation of the PHY. Comparisons with ASIC implementations would hence have to be based on the front-end and/or back-end only, but that still leaves the 2-to-1 command ratio as a significant difference. Although there are works which describe combinations of back-ends and PHYs on ASIC [46, 63], they provide insufficient information to clearly separate the contribution of the two components, and lack details on the controller implementation. Hence, a comparison with the back-end of these works would be hard to interpret, and at most of limited use.

The authors of [30] provide a verilog implementation of their controller front-end and back-end, and also have an FPGA as synthesis target. However, this controller has only been tested in simulation and lacks an FPGA PHY, the addition of which we expect impacts the back-end design in a similar way as that of Raptor (i.e., requiring a lower clock frequency and a parallel generation of multiple commands per cycle). Since the authors furthermore indicate that improvement of and further elaboration on the implementation is part of future work, we will not attempt to compare to it in its current state. An FPGA implementation of the controller from [64] is available, but it uses a relatively low-frequency SDR SDRAM. The hardware requirements on such a controller are so different from ours that a comparison is not useful.

An appropriate comparison that we can actually make involves the *Multi-Port Memory Controller (MPMC)* controller [65] from Xilinx. The MPMC is widely used, because it is the default SDRAM controller for Virtex 6 FPGAs and relatively easy to instantiate from the Xilinx tools. Its PHY is similar in structure to that of Raptor, uses the same I/O resources, and targets the same memory generation (DDR3). Both controllers generate two commands per (back-end) cycle. This allows us to focus the comparison on the main contributions of Raptor, which are the reconfigurable back-end and front-end. The number of basic FPGA resources (registers and LUTs) consumed by each design is used as the metric for comparison. Version 13.3 of the

Xilinx tools are used, and unless mentioned otherwise, we use the default settings provided by the Base System Builder wizard of the XPS tool to create the MPMC-based controllers. The MPMC version is v6.05.a.

The MPMC by default uses BRAMs to implement its equivalent of the atom queues. This has advantages in terms of timings, since they are essentially dedicated SRAMs on the FPGA fabric, but it also over-allocates the queues in terms of capacity, because the minimal size of a BRAM block is 4 KiB. Alternatively, the MPMC can be configured to use a *Shift-Register Lookup (SRL)* buffer implementation, which also maps efficiently to FPGA resources, but is available at smaller granularities. We select this configuration and set the atom queues in the Raptor front-end to the same size as the default MPMC SRL size, which is 512 B per read or write queue per port. Raptor's atom queues also map to SRL resources on the FPGA, which hence leads to comparable results in terms of size. Note that for Raptor, this queue size is configurable at design time, and does not necessarily have to be 512 B.

The MPMC and Raptor use different protocols for communicating with their clients: MPMC provides several protocol sockets, while Raptor uses DTL. We select *Processor Local Bus (PLB)* as the socket for the MPMC front-end, since it is similar to DTL in terms of wiring signature (AXI4 would be a more obvious choice, but is not available). We use a 32-bit SDRAM bus for both controllers (leaving half of the DIMM unconnected). The Raptor instances use a reconfigurable TDM arbiter, configured to have the same number of table slots as there are ports on the front-end. The MPMC uses a round-robin arbiter. We limit the fan-out of Raptor's configuration bus (Fig. 2.6) to 16 ports, and instantiate multiple buses if more than 16 reconfigurable components (more than 7 clients) are present.

2.6.2 Synthesis Results

Figure 2.19 shows the resource usage of the MPMC and Raptor with a varying number of front-end ports (eight is the maximum number of supported ports on the MPMC). Note that these numbers are indicative only, since place and route has not been done yet at this stage, and hence the wiring cost is not visible yet. The performance (clock frequency) after routing will vary based on the success of the mapping and routing heuristics, which is highly dependent on the other hardware which is placed on the same FPGA.

The figure shows that the LUT and register usage of Raptor and the MPMC are of the same order of magnitude, although Raptor consistently uses more resources: the MPMC uses 1305 registers and 930 LUTs per additional port on average, versus 1882 registers and 2304 LUTs per port for Raptor. The difference in size can mainly be attributed to:

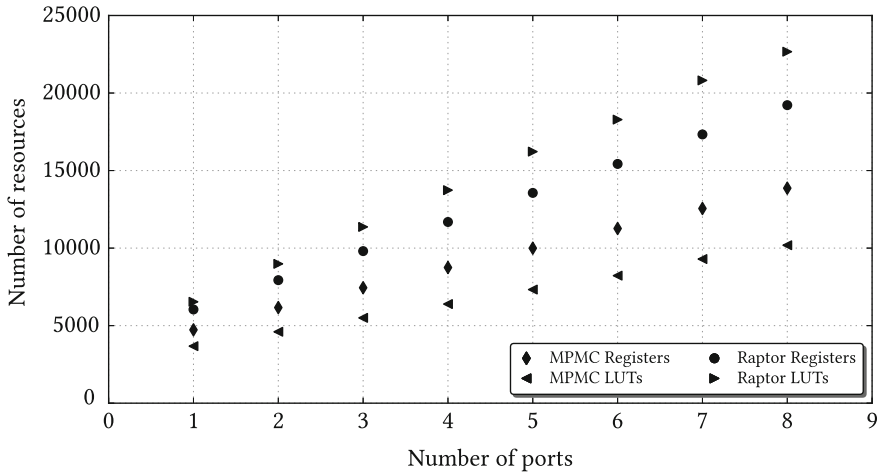


Fig. 2.19 Resource usage of Raptor versus MPMC using 512 byte read/write queues (1024 bytes in total) per port

- The modularity of the design: each DTL port incurs a handshaked-pipeline stage with double buffering for the command and data lines. This modularity allows the blocks in the front-end to be easily reused and individually instantiated as needed, at the cost of more hardware at their interfaces.
- The MPMC is tailored for the Virtex 6, often spelling out the exact mapping to basic FPGA resources, leaving very little to the imagination of the synthesis tool. This improves the maximum clock frequency and lowers the resource usage, but complicates portability to a different FPGA. Raptor is written at a slightly higher level of abstraction, and has not been extensively optimized for size.⁴
- The MPMC is synthesized as a single unit, while Raptor is separated in two, the first one containing the front-end, and the second containing the back-end and PHY. This means that the synthesis tool has more knowledge to exploit when it eliminates constants and unused hardware for the MPMC. Global optimization across blocks happens after the point where the numbers in Fig. 2.19 are extracted, and its results are hence not incorporated in the data set.
- Raptor can generate *any* SDRAM command at *any cycle*, while the MPMC restricts activates and precharges to even-cycles, and read and write commands to odd-cycles. This constraint has a slight performance implication in terms of bandwidth and response time.

⁴Compared to earlier publications on the approximate size of the controller [66], we did however reduce the resource usage of all FIFOs significantly by modifying their implementation such that they map to SRL and LUTRAM resources instead of individual registers. Hence, a 4-port controller with 512 bytes per queue now uses 88 % fewer registers in Fig. 2.19 than a version with 256 bytes per queue in [66].

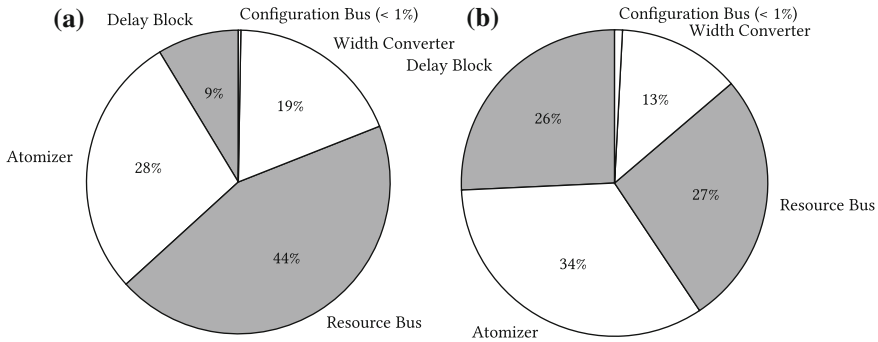


Fig. 2.20 Front-end LUT and register usage break-down per port. 100% = 1915 registers, 2837 LUTs. **a** Registers. **b** LUTs

- The reconfiguration infrastructure and delay block functionality that exists in Raptor is not available in the MPMC.

For a single-port controller, the front-end/back-end ratio is 0.48 for registers and 0.60 for LUTs, i.e., the back-end is bigger, while for an 8-port controller this shifts to 3.7 and 4.5, respectively, with the front-end dominating the resource usage.

Figure 2.20 shows a break-down of the resource usage in the front-end, obtained by individually synthesizing its components. Buses are dimensioned for eight front-end ports, and their size is divided by eight as an approximation of the contribution of each port. Since splitting the front-end into multiple synthesis units reduces the global optimization opportunities as mentioned earlier, the total number of registers and LUTs accounted for by the sum of the components is, respectively, 1.8% and 23% higher than the costs per port estimated based on Fig. 2.19. This underlines the importance of these optimizations, and should serve as a warning that the break-down is approximate only.

Figure 2.20a shows that the resource bus uses the most registers, at least relatively. It contains a set of pipeline registers as wide as its total fan-in (i.e., for each port), implemented as registers. The atom queues store significantly more bits, but use LUTs to do this, which is more efficient.⁵ Hence, the proportional register usage of the resource bus and the delay block might at first glance look unintuitive. The atomizers use a relatively large amount registers because they also contain the input-buffers for the front-end. For similar reasons, they use relatively more LUTs than the other components, as shown in Fig. 2.20b. The delay block spends approximately half of its LUTs on the atom queues, while the resource bus uses practically all of them to implement the required multiplexing logic.

⁵32 bits can be stored in a single LUT (although only one of those 32 bits can be read/written at a time), versus 1 bit per register.

Raptor and MPMC have different design goals: the first one provides real-time guarantees and isolation per client, while the second does not. MPMC is built to sustain a high average-case throughput and was optimized for size, while this is not the main focus of the Raptor prototype. It is hence not possible to connect hard conclusions to a size comparison of the two solutions, since they have different properties and applications areas. We observe that Raptor is consistently larger (2.2 and 1.3 times the size of the MPMC in LUTs and registers, respectively, according to Fig. 2.19). However, keeping in mind that Raptor is still the prototype stage, *the results indicate that the cost of the extra functionality that Raptor offers appear to be manageable.*

2.7 Conclusion

This chapter introduced the architecture template of a real-time memory controller. The main novel feature is its reconfigurability, which is expressed in two ways. Firstly, the components in the front-end are reconfigurable, allowing the performance that is provided to each port to be changed at run-time by modifying its front-end settings, i.e., budgets in the arbiter and delay block settings. Secondly, the back-end contains a pattern memory that holds the SDRAM commands the controller issues to the memory. The contents of the pattern memory can be changed at run-time to modify the properties of the scheduling algorithm implemented by the patterns. The application, properties and limitations of the available reconfiguration mechanisms will be discussed further in Chap. 7, while Chap. 3 elaborates on the possible configurations of the scheduling algorithm used to create the memory patterns that are stored in the back-end. Furthermore, we have shown how the worst-case performance of our SDRAM controller can be characterized in terms of worst-case bandwidth and WCRT. We apply this analysis later in Chap. 5 to compare the worst-case performance of different contemporary memory devices.

The *Raptor* instance of this controller template has been implemented and customized for use on an FPGA, and is a part of the CompSOC platform. The complete integration all the way down to the PHY level shows the controller successfully communicates with real SDRAM devices, and allowed for a resource usage comparison with the MPMC controller from Xilinx. This proved that our controller template can provide real-time capabilities at competitive costs, which has significant added value for mixed time-criticality systems. Additionally, *Raptor* has been used on a daily basis both in lab-based courses [67] and as a research vehicle [2, 68] for several years now, and has shown to be a stable and versatile component for these purposes.

References

1. Akesson B, Goossens K (2011) Memory controllers for real-time embedded systems. Embedded systems series. Springer, New York
2. Goossens K, Azevedo A, Chandrasekar K, Gomony MD, Goossens S, Koedam M, Li Y, Mirzoyan D, Molnos A, Nejad AB, Nelson A, Sinha S (2013) Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. SIGBED Rev 10(3):23–34
3. DRAmExchange (2015) Monthly worldwide DRAM output in 2015. http://www.dramexchange.com/Market/Market_Activity. Online; Accessed 15 Oct 2015
4. Amsterdam internet exchange (2015) Historical monthly traffic volume. <https://ams-ix.net/technical/statistics/historical-traffic-data?year=2015>. Online; Accessed 15 Oct 2015
5. Dennard RH (1968) Field-effect transistor memory. US Patent 3,387,286
6. Jacob B, Ng S, Wang D (2007) Memory systems: cache, DRAM, disk. Morgan Kaufmann Pub
7. JEDEC (2009) Low power double data rate specification JESD209B
8. JEDEC (2010) DDR3 SDRAM specification JESD79-3E
9. JEDEC (2010) Low power double data rate 2 specification JESD209-2D
10. JEDEC (2012) DDR4 SDRAM specification JESD79-4
11. JEDEC (2013) Low power double data rate 3 specification JESD209-3B
12. JEDEC (2009) DDR2 SDRAM specification JESD79-2F
13. Rixner S, Dally WJ, Kapasi UJ, Mattson P, Owens JD (2000) Memory access scheduling. In: International symposium on computer architecture (ISCA), pp 128–138
14. Mobile LPDDR2 SDRAM (2010) *2gb_mobile_lpddr2_s4_g69a.pdf - Rev. N 3/12 EN*. Micron
15. DDR3L SDRAM (2011) *4Gb_DDR3L.pdf - Rev. I 9/13 EN*. Micron
16. Kollig P, Osborne C, Henriksson T (2009) Heterogeneous multi-core platform for consumer multimedia applications. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1254–1259
17. RM57L843 16- and 32-Bit RISC Flash Microcontroller (2014). Texas Instruments Inc
18. van der Wolf P, Geuzebroek J (2011) SOC infrastructures for predictable system integration. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1–6
19. Snapdragon 800 (2015) Snapdragon 800 processor specs. <https://www.qualcomm.com/products/snapdragon/processors/800>. Online; Accessed 30 Mar 2015
20. JEDEC (2014) 240 pin DDR3 DIMM, 1.00mm pitch MO-269J
21. JEDEC (2014) DDR3 unbuffered SODIMM reference design specification 4.20.18, revision 2.8, release 24
22. Gomony MD, Akesson B, Goossens K (2015) A real-time multichannel memory controller and optimal mapping of memory clients to memory channels. ACM Trans Embed Comput Syst 14(2):25:1–25:27
23. Chandrasekar K, Akesson B, Goossens K (2012) Run-time power-down strategies for real-time SDRAM memory controllers. In: Design automation conference (DAC), pp 988–993
24. Reineke J, Liu I, Patel HD, Kim S, Lee EA (2011) PRET DRAM controller: bank privatization for predictability and temporal isolation. In: Proceedings of CODES+ISSS, pp 99–108
25. Akesson B, Goossens K (2011) Architectures and modeling of predictable memory controllers for improved system integration. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1–6
26. Ecco L, Tobuschat S, Saidi S, Ernst R (2014) A mixed critical memory controller using bank privatization and fixed priority scheduling. In: Embedded and real-time computing system and application (RTCSA)
27. Hassan M, Patel H, Pellizzoni R (2015) A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In: Real-time and embedded technology and application symposium (RTAS), pp 307–316
28. Shah H, Raabe A, Knoll A (2012) Bounding WCET of applications using SDRAM with priority based budget scheduling in MPSoCs. In: Design, automation and test in Europe conference and exhibition (DATE), pp 665–670

29. Paolieri M, Quiñones E, Cazorla FJ (2013) Timing effects of DDR memory systems in hard real-time multicore architectures: issues and solutions. *ACM Trans Embed Comput Syst* 12(1s):64
30. Krishnapillai Y, Pei Wu Z, Pellizzoni R (2014) ROC: a rank-switching, open-row DRAM controller for time-predictable systems. In: Euromicro conference on real-time systems (ECRTS), pp 27–38
31. Kim H, de Niz D, Andersson B, Klein M, Mutlu O, Rajkumar R (2014) Bounding memory interference delay in COTS-based multi-core systems. In: Real-time and embedded technology and application symposium (RTAS), pp 145–154
32. Akesson B, Hansson A, Goossens K (2009) Composable resource sharing based on latency-rate servers. In: Digital system design (DSD)
33. Device transaction level (DTL) protocol specification (2002) Version 3.2. Philips semiconductors
34. AMBA AXI and ACE protocol specification (2011). ARM Limited
35. Akesson B, Molnos A, Hansson A, Ambrose Angelo J, Goossens K (2010) Composability and predictability for independent application development, verification, and execution. In: Hübner M, Becker J (eds) Multiprocessor system-on-chip — hardware design and tool integration, Circuits and systems, chapter 2. Springer. ISBN 978-1-4419-6459-5
36. Hansson A, Goossens K, Bekooij M, Huisken J (2009) CompSOC: a template for composable and predictable multi-processor system on chips. *ACM TODAES* 14(1)
37. Stiliadis D, Varma A (1998) Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Trans Netw* 6(5)
38. Kopetz H (1997) Real-time systems: design principles for distributed embedded applications. Springer
39. Ghosal A, Henzinger TA, Kirsch CM, Sanvido MA (2004) Event-driven programming with logical execution times. In: Hybrid systems: computation and control, pp 357–371. Springer
40. Goossens S, Akesson B, Koedam M, Nejad AB, Nelson A, Goossens K (2013) The CompSOC design flow for virtual execution platforms. In: Proceedings of the 10th FPGAWorld conference, pp 7:1–7:6
41. Nagle JB (1987) On packet switches with infinite storage. *IEEE Trans Commun* COM-35(4)
42. Akesson B, Steffens L, Strooisma E, Goossens K (2008) Real-time scheduling using credit-controlled static-priority arbitration. In: Embedded and real-time computing system and application (RTCSA), pp 3–14
43. Memory transaction level (MTL) protocol specification (2002) CoReUse 3.2.1. Philips semiconductors
44. Bhat B, Mueller F (2011) Making DRAM refresh predictable. *Real-Time Syst* 47(5):430–453
45. Denali (2010) DDR PHY interface (DFI) specification version 2.1.1
46. Kaviani K, Wu T, Wei J, Amirkhany A, Shen J, Chin T, Thakkar C, Beyene W, Chan N, Chen C, Chuang BR, Dressler D, Gadde V, Hekmat M, Ho E, Huang C, Le P, Mahabaleshwara CM, Mishra N, Raghavan L, Saito K, Schmitt R, Secker D, Shi X, Fazeel S, Srinivas G, Zhang S, Tran C, Vaidyanath A, Vyas K, Jain M, Chang K-Y K, Yuan X (2012) A tri-modal 20-Gbps/Link differential/DDR3/GDDR5 memory interface. *IEEE J Solid-State Circuits* 47(4):926–937
47. Akesson B (2010) Predictable and composable system-on-chip memory controllers. PhD thesis, Eindhoven University of Technology
48. Akesson B, Hayes Jr W, Goossens K (2010) Classification and analysis of predictable memory patterns. In: Embedded and real-time computing systems and applications (RTCSA), pp 367–376
49. Wu ZP, Krish Y, Pellizzoni R (2013) Worst case analysis of DRAM latency in multi-requestor systems. In: Real-time systems symposium, pp 372–383
50. Shah H, Knoll A, Akesson B (2013) Bounding SDRAM interference: detailed analysis vs. latency-rate analysis. In: Design, automation and test in Europe conference and exhibition (DATE), pp 308–313
51. Bhati I, Chishti Z, Lu SL, Jacob B (2015) Flexible auto-refresh: enabling scalable and energy-efficient DRAM refresh reductions. In: International Symposium on Computer Architecture (ISCA)

52. Akesson B, Minaeva A, Sucha P, Nelson A, Hanzalek Z (2015) An efficient configuration methodology for time-division multiplexed single resources. In: Real-time and embedded technology and application symposium (RTAS)
53. Minaeva A, Šúcha P, Akesson B, Hanzálek Z (2016) Scalable and efficient configuration of time-division multiplexed resources. *J Syst Softw* 113:44–58
54. Nelson A, Goossens K, Akesson B (2015) Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. *J Syst Archit*
55. Sriram S, Bhattacharyya S (2000) Embedded multiprocessors: scheduling and synchronization. CRC
56. Xilinx (2011) ML605 documentation UG533. http://www.xilinx.com/support/documentation/boards_and_kits/ug533.pdf
57. Chandrasekar K, Weis C, Li Y, Akesson B, Wehn N, Goossens K (2014) Drampower: open-source DRAM power and energy estimation tool. <http://www.drampower.info>
58. Akesson B, Goossens K, Ringhofer M (2007) Predator: a predictable SDRAM memory controller. In: Proceedings of CODES+ISSS
59. Xilinx (2011) Virtex-6 FPGA memory interface solutions - user guide UG406
60. Cosoroaba A (2013) Achieving high performance DDR3 data rates, Xilinx, WP383 (v1.2). White paper
61. Cadence Design Systems Inc (2014) Multi-protocol LPDDR4/3/DDR4/3 controller and PHY subsystem IP. http://ip.cadence.com/uploads/file/1021/638/Cadence_Multi-Protocol_LPDDR4_3_DDR4_3_Subsystem_ds.pdf
62. *DDR3 SDRAM SODIMM - MT4J5F6464H - 512MB JSF4C64_64x64HY.fm - Rev. B 3/08 EN* (2007). Micron
63. Chang K, Lee H, Chun J-H, Wu T, Chin T, Kaviani K, Shen J, Shi X, Beyene W, Frans Y, Leibowitz B, Nguyen N, Quan F, Zerbe J, Perego R, Assaderaghi F (2008) A 16Gb/s/link, 64GB/s bidirectional asymmetric memory interface cell. In: 2008 IEEE symposium on VLSI circuits, pp 126–127
64. Lakis E, Schoeberl M (2013) An SDRAM controller for real-time systems. In: 2013 IEEE 16th international symposium on object/component/service-oriented real-time distributed computing (ISORC), pp 1–8
65. Xilinx (2011) LogiCORE IP - multi-port memory controller DS643
66. Goossens S, Kuijsten J, Akesson B, Goossens K (2013) A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In: 2013 international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 1–10
67. Nelson A, Molnos A, Nejad AB, Mirzoyan D, Cotofana S, Goossens K (2013) Embedded computer architecture laboratory: A hands-on experience programming embedded systems with resource and energy constraints. In: Proceedings of the workshop on embedded and cyber-physical system education, pp 7:1–7:8
68. Schoeberl M, Abbaspour S, Akesson B, Audsley N, Capasso R, Garside J, Goossens K, Goossens S, Hansen S, Heckmann R, Hepp S, Huber B, Jordan A, Kasapaki E, Knoop J, Li Y, Prokesch D, Puffitsch W, Puschner P, Rocha A, Silva C, Sparsø J, Tocchi A (2015) T-CREST: time-predictable multi-core architecture for embedded systems. *J Syst Archit*

Memory Controllers for Mixed-Time-Criticality Systems

Architectures, Methodologies and Trade-offs

Goossens, S.; Chandrasekar, K.; Akesson, B.;

Goossens, K.

2016, XXVII, 202 p. 78 illus. in color., Hardcover

ISBN: 978-3-319-32093-9