Embedded Systems

Sven Goossens
Karthik Chandrasekar
Benny Akesson
Kees Goossens

Memory Controllers for Mixed-Time-Criticality Systems

Architectures, Methodologies and Trade-offs

Springer

Sven Goossens · Karthik Chandrasekar
Benny Akesson · Kees Goossens

Memory Controllers for Mixed-Time-Criticality Systems

Architectures, Methodologies and Trade-offs

Springer

Sven Goossens
Faculty of Electrical Engineering
Technische Universiteit Eindhoven
Eindhoven, Noord-Brabant
The Netherlands

Karthik Chandrasekar
Nvidia Graphics
Bangalore, Karnataka
India

Benny Akesson
CISTER/INESC TEC
Polytechnic Institute of Porto
Porto
Portugal

Kees Goossens
Faculty of Electrical Engineering
Technische Universiteit Eindhoven
Eindhoven, Noord-Brabant
The Netherlands

ISSN 2193-0155 ISSN 2193-0163 (electronic)
Embedded Systems
ISBN 978-3-319-32093-9 ISBN 978-3-319-32094-6 (eBook)
DOI 10.1007/978-3-319-32094-6

Library of Congress Control Number: 2016935587

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

Preface

The authors of this book all worked together at the Eindhoven University of Technology in the Netherlands. They were united in what was unofficially called the “Memory Team,” as either a Ph.D. student, an assistant professor, or a professor. The team worked on various challenging research topics in the context of memory controllers for real-time embedded systems, which matched well with the overall goals of the Electronic Systems group by which they were all hosted. The authors thank the other Ph.D. students in the Memory Team, Manil Gomony and Yonghui Li, for their valuable input during countless discussions over the years, and for all the fun that was had in the process. A large portion of the preliminary exploration work for the topics discussed in this book was done by two excellent master students, Tim Kouters and Jasper Kuijsten. They were great to have around, and delivered good work, for which the authors are grateful.

A memory controller requires a system that tells it what to do. For the controller in this book, the system takes the form of the CompSOC platform. The various hardware and software components it consists of were jointly maintained by Eindhoven University of Technology and Delft University of Technology. Most of the experiments in this book would not have been possible without the infrastructure created by current and past CompSOC team members. The authors would particularly like to thank Anca Molnos, Andrew Nelson, Ashkan Beyranvand Nejad, Davit Mirzoyan, Gabriela Breaban, Juan Valencia, Martijn Koedam, Radu Stefan, Rasool Tavakoli, Reinier van Kampenhout, and Shubhendu Sinha for their work, and the great company they are.

Finally, the authors thank their family and friends, for all the obvious reasons. Without their support, it is very unlikely this book would have existed.

Contents

About the Authors

Sven Goossens received his M.Sc. in Embedded Systems from the Eindhoven University of Technology in 2010. He worked as a researcher in the Electrical Engineering of the same university until 2011, and then started as a Ph.D. student, graduating in 2015. He is currently employed as a Hardware Architect at Intrinsic-ID. His research interests include mixed time-criticality systems, composability, and SDRAM controllers.

Karthik Chandrasekar earned his M.Sc. degree in Computer Engineering from TU Delft in the Netherlands in November 2009. In October 2014, he received his Ph.D. also from the same university. His research interests include SoC Architectures, DRAM memories and memory controllers, on-chip communication networks and performance and power modeling and analysis. He is currently employed as a Senior Architect at Nvidia.

Benny Akesson received his M.Sc. degree at Lund Institute of Technology, Sweden in 2005 and a Ph.D. from Eindhoven University of Technology, the Netherlands in 2010. Since then, he has been employed as a Researcher at Eindhoven University of Technology, Czech Technical University in Prague, and CISTER/INESC TEC Research Unit in Porto. Currently, he is working as a Research Fellow at TNO-ESI. His research interests include memory controller architectures, real-time scheduling, performance modeling, and performance virtualization. He has published more than 50 peer-reviewed conference papers and journal articles, as well as two books about memory controllers for real-time embedded systems.

Kees Goossens received his Ph.D. in Computer Science from the University of Edinburgh in 1993. He worked for Philips/NXP Research from 1995 to 2010 on networks-on-chips for consumer electronics, where real-time performance, predictability, and costs are major constraints. He was part-time Professor at Delft

University from 2007 to 2010, and is now Full Professor at the Eindhoven University of Technology, where his research focuses on composable (virtualized), predictable (real-time), low-power embedded systems, supporting multiple models of computation. He has published 4 books, 100+ papers, and 24 patents.

List of Figures

List of Tables

Acronyms

ACT	Activate
AG	Access granularity
ANP	Activate, no Precharge mode
AP	Activate and Precharge mode
ASIC	Application-Specific Integrated Circuit
AXI4	Advanced eXtensible Interface 4
BC	Burst Count
BGI	Bank-Group Interleaving
BI	Bank Interleaving
BL	Burst Length
BRAM	Block RAM
BS	Bank Scheduling
BS BI	Bank Scheduling, variable BI
BS PBGI	Bank Scheduling with Pairwise Bank-Group Interleaving
CCSP	Credit-Controlled Static-Priority
CDC	Clock Domain Crossing
CompSOC	Composable System-on-Chip
COTS	Commercial-Off-the-Shelf
CSDF	Cyclo-Static Data Flow
DDR	Double Data Rate
DFI	DDR PHY Interface
DIMM	Dual Inline Memory Module
DMA	Direct Memory Access
DRAM	Dynamic Random-Access Memory
DTL	Device Transaction Layer
FAW	Four Activate Window
FIFO	First-in First-out
FPGA	Field-Programmable Gate Array
FR-FCFS	First-Ready First-Come First-Served
FSL	Fast Simplex Link
IDD	Current flow in power supply lines

ILP	Integer Linear Programming
IP	Intellectual Property
IW	Interface Width
JEDEC	Joint Electron Device Engineering Council
$\mathcal{LR}$	Latency-Rate
LSB	Least Significant Bits
LUT	Lookup Table
MMIO	Memory Mapped I/O
MPMC	Multi-Port Memory Controller
MTL	Memory Transaction Layer
NANP	No Activate, No Precharge mode
NAP	No Activate, Precharge mode
NoC	Network-on-Chip
NOP	No Operation
NP	No Precharge mode
ODT	On-Die Termination
PDE	Power-Down Entry
PDX	Power-Down Exit
PHY	Physical interface
PLB	Processor Local Bus
PLL	Phase Locked Loop
PRE	Precharge
PREA	Precharge All
RAM	Random-Access Memory
RD	Read
REF	Refresh
RTW	Read-to-Write (pattern)
SDRAM	Synchronous Dynamic Random-Access Memory
SI	Scheduling Interval
SoC	System-on-Chip
SO-DIMM	Small Outline DIMM
SOS	Special Ordered Sets
SRAM	Static Random-Access Memory
SRE	Self-Refresh Entry
SRL	Shift-Register Lookup
SRX	Self-Refresh Exit
TDM	Time-Division Multiplexing
V_{DD}	Voltage on power supply lines
VHDL	VHSIC Hardware Description Language
WCET	Worst-Case Execution Time
WCIAAT	Worst-Case Inter-Atom Time
WCRT	Worst-Case Response Time
WCSI	Worst-Case Scheduling Interval
WR	Write
WTR	Write-to-Read (pattern)

Symbols

General

b_{wc}	Worst-case bandwidth in MB/s or GB/s
b_{peak}	Peak bandwidth in MB/s or GB/s
e_{ref}	Refresh efficiency $(0 < e_{ref} \leq 1)$
e	Memory efficiency $(0 < e \leq 1)$
f	Clock frequency in MHz
p_{wc}	$\max(p^r_{wc}, p^w_{wc})$ in mW
ρ	$\mathcal{LR}$ (allocated) rate
t	time
t^p_r	Predictable read pattern length in #cycles
t^p_{rtw}	Predictable read-to-write pattern length in #cycles
t^p_{ref}	Predictable refresh pattern length in #cycles
t^p_w	Predictable write pattern length in #cycles
t^p_{wtr}	Predictable write-to-read pattern length in #cycles
Θ	$\mathcal{LR}$ service latency

In Pattern Figures

$\overline{aP}$	Cycle where an auto-precharge is executed
A	Activate (command)
R	Read (command)
W	Write (command)

Chapter 2

c	Client
δ^f_{be}	Pipeline latency of back-end on request path in #cycles

δ^f_{PHY}	Pipeline latency of PHY on request path in #cycles
δ^b_{be}	Pipeline latency of back-end on response path in #cycles
δ^b_{PHY}	Pipeline latency of PHY on response path in #cycles
δ^f	Pipeline latency of back-end and PHY on request path in #cycles
δ^b	Pipeline latency of back-end and PHY on response path in #cycles
δ_{fe}	Pipeline latency of front-end on combined request/response path in #cycles
Δ_r	#cycles from first read pattern command on the SDRAM command bus and first data on the SDRAM data bus
Δ_w	#cycles from first write pattern command on the SDRAM command bus and first data on the SDRAM data bus
Δ'_r	#cycles from first read pattern command on the SDRAM command bus and the first data handshakes on the back-end interface
Δ'_w	#cycles from first write pattern command on the SDRAM command bus and the first data handshakes on the back-end interface
$m_0 - m_3$	Address decoder masks
ρ_{be}	Rate of back-end $\mathcal{LR}$ server in MB/s
ρ^c_{arb}	Rate of arbiter $\mathcal{LR}$ server for client c as a fraction of the total server bandwidth $\left(0 \leq \rho^c_{arb} \leq 1\right)$
ρ^c_{fe}	Rate of front-end $\mathcal{LR}$ server for client c in MB/s
ρ^c_{ctrl}	Rate of combined front-end/back-end $\mathcal{LR}$ server for client c in MB/s
$s_0 - s_3$	Address decoder shift amounts
θ_r	#cycles until service for a read atom that starts a busy period
θ_w	#cycles until service for a write atom that starts a busy period
Θ_{be}	Service latency of back-end $\mathcal{LR}$ server in #cycles
Θ^c_{arb}	Service latency of arbiter $\mathcal{LR}$ server for client c in #scheduling slots
Θ^c_{fe}	Service latency of front-end $\mathcal{LR}$ server for client c in #cycles
Θ^c_{ctrl}	Service latency of combined front-end/back-end $\mathcal{LR}$ server for client c in #cycles

Chapter 3

$d(cmd_a, cmd_b)$	Function that returns the minimum relative delay between cmd_a and cmd_b in #cycles
e_{pc}	Conversion efficiency from predictable to composable patterns $(0 < e_{pc} \leq 1)$
$S^j_k(i)$	Function that returns the ith timestamp $(i \in [1..100])$ in run $j \in [1..122]$ of type $k \in [1, 2, 3, 4]$ of the experiment
$s^j_k(i)$	Function that returns the relative timestamp $S^j_k(i) - S^1_4(i)$
t^c_r	Composable read pattern length in #cycles
t^c_w	Composable write pattern length in #cycles

Chapter 4

E	Energy
E_a	Active energy
E_{ACT}	Energy cost of an ACT command
E_{bg}	Background energy
E_{PRE}	Energy cost of a PRE command
E_{PREA}	Energy cost of a PREA command
E_{RD}	Energy cost of a RD command
E_{REF}	Energy cost of a REF command
E_{WR}	Energy cost of a WR command
I_{bg}	Background
I_{DD0}	One bank active-precharge current
I_{DD1}	One bank active-read-precharge current
I_{DD2P0}	Precharge power-down current—slow-exit
I_{DD2P1}	Precharge power-down current—fast-exit
I_{DD3N}	Active standby current
I_{DD3P}	Active power-down current
I_{DD4R}	Burst read current
I_{DD4W}	Burst write current
I_{DD5B}	Refresh current
I_{DD6}	Self-refresh current
n_{open_banks}	Number of open banks
P_M^{RDQ}	I/O power per data bit during a read
P_M^{WDQ}	I/O power per data bit during a write

Chapter 5

b'	Bandwidth delivered by a worst-case power trace in MB/s or GB/s
$b_r^{measured}$	Measured bandwidth when continuously reading in MB/s
$b_w^{measured}$	Measured bandwidth when continuously writing in MB/s
$b_{rw}^{measured}$	Measured bandwidth when continuously alternating read and write requests in MB/s
p'	Power in mW of a worst-case bandwidth trace
p_{wc}^r	Average power in mW used when *continuously* serving read requests
p_{wc}^w	Average power in mW used when *continuously* serving write requests

Chapter 6

A	#cycles added to time-window by Algorithm 5
PS	Pattern size in #cycles
WS	Time-window size in #cycles after applying Algorithm 5

Chapter 7

c	Client
c_1, c_2	Two independent and distinct allocations for a client
(Θ_1, ρ_1)	$\mathcal{LR}$ parameters corresponding to allocation c_1
(Θ_2, ρ_2)	$\mathcal{LR}$ parameters corresponding to allocation c_2
(Θ_r, ρ_r)	$\mathcal{LR}$ parameters corresponding to the client's requirements
ϕ^c	Number of slots $\left(\in \mathrm{N}_0^+\right)$ allocated to client c
ϕ_1	Number of slots $\left(\in \mathrm{N}_0^+\right)$ allocated to client c in allocation c_1
ϕ_2	Number of slots $\left(\in \mathrm{N}_0^+\right)$ allocated to client c in allocation c_2
ϕ_{ol}	Number of overlapping slots $\left(\in \mathrm{N}_0^+\right)$ across c_1 and c_2
ρ^c_{tdm}	Rate of TDM arbiter $\mathcal{LR}$ server for client c as a fraction of the total server bandwidth $\left(0 \leq \rho^c_{tdm} \leq 1\right)$
ρ_{ol}	Rate corresponding to the overlapping slots across c_1 and c_2 as a fraction of the total server bandwidth $(0 \leq \rho_{ol} \leq 1)$
Θ^c_{tdm}	Service latency of TDM arbiter $\mathcal{LR}$ server for client c in #scheduling slots
Θ'	$\max(\Theta_1, \Theta_2)$
τ	Start of a busy period
τ'	End of a busy period
t_A	The time at which allocation c_2 is fully enabled in the slot table
t_R	The time at which allocation c_1 is fully disabled in the slot table
$\mathcal{T}$	Length of the slot table in a TDM arbiter $\left(\in \mathrm{N}^+_{>0}\right)$
$w_r(t)$	Required $\mathcal{LR}$ service bound of the client
$w_g(t)$	$\mathcal{LR}$ service guarantee given to the client

Appendix A

c	A command 3-tuple (c_t, c_b, c_n)
c_t	Command type, $c_t \in \{ACT, RD, WR, PRE\}$
c_b	Command bank, $c_b \in \{0...\mathrm{BI}-1\}$
c_n	Command incarnation, $c_n \in \{0, 1\}$
C_{ACT}	Set of activate commands
C_{PRE}	Set of (auto) precharge commands
C_{RW}	Set of read/write commands
C	Set of all commands
K	Number of commands of a specific type allowed within a window
L_c	Lower bound on the position of command c
nbg	The number of bank groups in the considered SDRAM device
$N_{heuristic}$	Upper bound on the pattern length based on Algorithm 2
$\hat{\mathrm{PRE}}$	ILP variable representing the position of the last precharge in the pattern

$pos(V_c)$	Returns a sub-expression representing the position of a command c in the pattern
s	Scaling factor to make pattern length the primary optimization goal
TC_{tp}	The value of the timing constraints in #cycles between two commands of type tp, $tp \in \{\mathrm{ACT}, \mathrm{RD}, \mathrm{WR}\}$
U_c	Upper bound on the position of command c
V_c	Set of Boolean variables in the ILP related to command c
X_i^c	Boolean variable. *true* if command c is scheduled in cycle i, *false* otherwise

Chapter 1
Introduction

The average human has a working memory capacity of seven digits, according to one of the most cited publications in psychology [1]. This means if you try to recite the series 3, 8, 5, 3, 2, 1, 1 after simply reading it once you will probably succeed, but repeating the trick with 4, 8, 1, 5, 1, 6, 2, 3, 4, 2 will likely fail. "Working memory" is colloquially used to refer the *Random-Access Memory* (*RAM*) in a computer system. The majority of the RAM in a computer is *Synchronous Dynamic Random-Access Memory* (*SDRAM*), which is the center point of this book. Seven digits is approximately equal to 23 bits of information ($7 \times {}_2\log(10)$), so in some sense, we all are the proud owners of 23 bits of brain-RAM, which is almost enough to store the word "bit" in standard 8-bit ASCII encoding in a computer.

Even though the 640 KB RAM that Bill Gates is rumored[1] to have said "ought to be enough for anybody" is already five orders of magnitude larger than the working memory of the brain, the world's hunger for memory has grown far beyond this number. Memory sizes in the order of gigabytes are now commonplace. The advances in memory capacity are part of a much larger trend, in which the number of transistors that can be manufactured for the same cost grows over time, due to the down-scaling of semiconductor circuits, as stated by *Moore's law* [2]. An equally important aspect of technology scaling is described by Dennard [3], who notes that the power density of a chip remains constant, despite the scaling. The combination of Moore's law and Dennard scaling implied that the amount of potential functionality offered by a chip of constant size and with a constant power envelope grew almost exponentially over the past decades, and even though we may have reached the tail-end of this trend [4, 5], we are still experiencing the benefits today.

In this chapter, we first look at the developments and trends that led to the current way of working with SDRAM in Sects. 1.1–1.4. We then identify the requirements on a modern SDRAM controller in Sect. 1.5, and capture them in the problem statement in Sect. 1.6. Here, we also briefly discuss how the contributions of this book address the raised issues. Finally, we link the contributions to the remaining chapters in Sect. 1.7.

[1]There are no reliable sources that confirm this quote.

S. Goossens et al., *Memory Controllers for Mixed-Time-Criticality Systems*, Embedded Systems, DOI 10.1007/978-3-319-32094-6_1

1.1 The SoC—SDRAM Interface

The ability of chips to harbor more and more transistors led to the integration of relatively powerful computing systems, into what is called a *system-on-chip* (*SoC*). Their large computing capacity makes it possible to merge multiple distinct pieces of functionality onto a single SoC [6], as opposed to using a separate chip for each of them. The main advantage of doing this is cost reduction [7], which can be attributed to several forms of *resource sharing*. Most obviously, consolidating the functionality of multiple chips on a single SoC reduces the number of chips and the associated costs involved in their manufacturing, consisting of raw materials, masks, packaging, etc. Common circuit components related to power distribution and clock generation may be shared. On-chip wires are less expensive in terms of area and power compared to wires that leave the silicon, also reducing the costs.

SDRAM has been largely left out of this integration trend, i.e., most modern SoCs connect to an external SDRAM chip. This can again be attributed to economic pressure that drives the development of *dynamic random-access memory* (*DRAM*) semiconductor technology in the direction of high-density and low-leakage chips. The design goal is to reduce the costs per bit and power consumption, while still satisfying capacity demands, contrary to logic-circuit technology which was mostly guided by speed requirements [8]. Uniting DRAM and logic in the same technology is not fundamentally impossible, but it is generally less cost effective than using separate chips.

The implication of separating the SoCs from the SDRAM is that a chip-to-chip interface has to be used to connect them, as shown in Fig. 1.1. Interfaces (pins) that connect the SoC to external chips are relatively expensive. The *International Technology Roadmap for Semiconductors* (*ITRS*) [9] estimates the costs per pin as 0.21 (dollar) cents for an SDRAM chip, and as 0.20 and 1.21 cents per pin on a general low-end or high-performance SoC package, respectively. Assuming these SoCs have a 84-pin and 240-pin memory interface, their total cost (at the SoC side) is approximately between 0.17 and 2.90 dollars per chip, respectively.

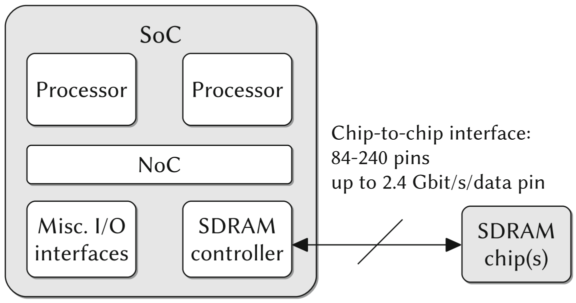


Fig. 1.1 Typical SoC-SDRAM interface

The number of pins that can be spent on the SDRAM interface is limited by the size of the SoCs package, which additionally has to accommodate power-supply pins and all other external connections. This naturally creates a bottleneck at the interface, and the associated requirement to use the available SDRAM pins as efficiently as possible.

A complicating factor at play here is the so-called *memory wall* problem [10, 11], which, in short, consists of the observation that the performance of logic grows faster than the performance of memory, such that memory performance eventually dominates the overall system performance. The memory wall exists for the same reason as the separation between the SoC and the DRAM, i.e., it is an implication of the different optimization goals that are applied to logic-circuit and DRAM technology. Even though 3D stacking promises to increase the number of connections between logic and memory [12, 13], improving the available bandwidth significantly, it does not seem likely that the drive for using the SDRAM interface efficiently will leave the picture.

Looking back at these developments helps to explain the status quo: we only have a relatively narrow interface by which the memory can be reached and shared by the entire SoC. Even though parallelism (in terms of the production and consumption of data) may exist within both the SoC and SDRAM, we require a serializing component that controls what data is transported across the interface at a given time. The *SDRAM controller* fulfills this role, and is discussed in more detail in the next section.

1.2 SDRAM Controllers

An SDRAM controller is the interface for the SoC to the SDRAM devices. SDRAM controllers have one or more ports (on the SoC side), and each port is connected to a *memory client*. We define *clients* as the sources of memory traffic that are directly connected to the SDRAM controller. Clients generate read or write *requests* for the controller, which are queued until it is ready to execute them.

Figure 1.2 shows a simplified memory controller architecture. It is divided into a *front-end* and *back-end*. The front-end deals with the multi-ported nature of the

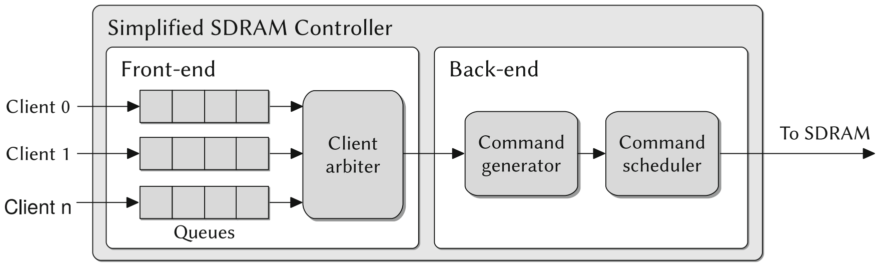


Fig. 1.2 Simplified general memory controller architecture

controller, by deciding on the order in which requests from different clients are executed. It contains an (inter-client) *arbiter* and *queues* for requests that are not executed immediately.

The back-end deals with the SDRAM protocol itself. Scheduled request are translated into *SDRAM commands* by a *command generator*. Once the commands are generated, they can be scheduled for execution on the SDRAM by the *command scheduler*. Although request-level arbitration and command-level scheduling are conceptually separate issues, they may be combined, depending on the controller implementation, although this naturally blurs the line between front-end and back-end.

Command scheduling is complex, since there are multiple timing constraints that have to be satisfied for each individual command to use the SDRAM correctly according to its specification. Each scheduling decision changes the memory state, and thus the constraints need to be taken into account for future decisions. Additionally, a scheduler may have to choose between multiple schedulable commands without a clear indication of the impact on performance and future scheduling options. This leads to a type of emergent behavior that is hard to predict, and thus memory performance is hard to bound in the general case; for many commercial controllers, no analytical bounds can be provided. However, there are real-time memory controllers that do provide hard bounds on the time to serve all requests to assure client-level requirements are always satisfied, as will be discussed in Sect. 1.5.

In the next section, we will have a closer look at what the clients of a memory controller actually represent, and why they are growing in number.

1.3 Cramming More Applications onto (Power-Constrained) SoCs

Cost reduction through resource sharing is the main cause for the rise of the SoC, and their availability paved the way for a growing catalog of *applications* that use them. Applications might be purely software based, like those found in the app-store of the particular phone ecosystem one subscribes to. However, we use "application" here in the broader sense of the word, and also include combinations of hardware and software that offer a certain chunk of functionality to the end user through *sensors, actuators, and communication links* [14].

It is often possible to think of the clients of a memory controller as applications, although this does not always work. For example, an application may be distributed over multiple processing cores, each having its own connection to the controller, and hence *one* application might be represented by *multiple* clients. In our definition, applications cannot communicate or share data (if they do one of these things, they by definition are part of the same application).

The success of SoCs has enabled the use of high-performance multi-core architectures in consumer electronics, like mobile phones [15], tablets [16], wearables (smart

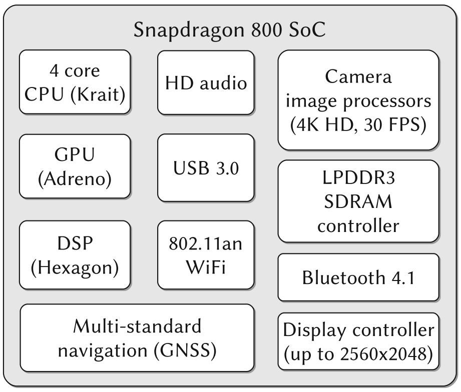


Fig. 1.3 The snapdragon 800 SoC [15]

watches, health trackers), home automation, and smart TVs [17], for example. Similarly, they find their way into cars, which contain a large number of *electronic control units* (*ECUs*) [18], essentially SoCs with control applications. In all these areas, we observe that *the number of applications on a single SoC is increasing, as a logical consequence of growing SoC performance, the availability of the applications, and the drive toward cost reduction*.

For example, consider mobile phones or tablets, which are simultaneously involved in handling a multitude of wireless protocols (like Wifi, Bluetooth, LTE, GPS, and NFC), implemented in dedicated radio solutions or by using *software-defined radio* (*SDR*) [19]. At the same time, they render graphics onto the screen, deal with encryption, while running user apps and the underlying operating system, all on the same SoC [7]. A high-end phone SoC is drawn in Fig. 1.3, illustrating the various applications it supports. Wearables aspire toward the same feature set, although they are significantly more battery constrained. In general, battery capacity does not grow as quickly as the demand for processing power [20]. For mobile devices, the expected battery lifetime constrains the available power budget for the SoC and the SDRAM, and limiting power usage is hence an important design goal in this area.

One of the main challenges in the car industry is to merge the functionality of multiple ECUs, reducing costs in terms of materials, cabling, and weight. Simultaneously, the trend toward (semi-) automated driving increases the required feature-set of cars. Automated driving heavily relies on sensing (vision, radar) and communication applications [21–23], for which custom SoCs are desired to effectively deal with all the required computation.

In conclusion, we see a growth of the number of applications per SoC across application domains. A subset of those applications uses the SDRAM, and hence turn into clients of the memory controller. Unfortunately, the effects of sharing are not all positive, especially when it involves a scarce resource like the SDRAM interface. In the next section, we discuss how applications are judged by their performance and why resource sharing can have a negative impact on it.

1.4 Performance

The evaluation of the success or failure of an application can be qualitative, but for the most part it is quantified in terms of *performance*. Performance is an umbrella term describing the *rate* at which something of interest is produced or consumed, or the *amount of time* it takes to complete a specific operation [24]. Each instance of such rate or quantity of time is called a *performance metric*. For example, a video decoder's performance may be expressed as frames per second, or a control loop can process a specific number of input samples per millisecond. A *better or higher* performance almost universally refers to an *increase* of the rate or *reduction* of time, except when the quantity is consumed energy, in which case a smaller energy/time ratio is considered better.

Some performance metrics straight-forwardly apply to SDRAM controllers [25]. Bandwidth (bytes/second), response time (seconds/access) and power (joules/second) are the ones featured most prominently in this book. At first sight this might seem strange, since there are not many people who actually care how much bandwidth a certain application receives, or how much power the SDRAM consumes on its behalf. Instead, *requirements* are usually expressed at a higher level of abstraction, based on a specification of the functionality for user, e.g., "the video should play smoothly," or "the battery should last for at least 24 h." Once refined in terms of performance metrics, *requirements bound the allowed performance*. Usually, requirements are one-sided (upper/lower) bounds, e.g., "at least 60 frames per second should be generated," or "at most 3 watts may be consumed." *A guarantee bounds the actual performance*. When the guaranteed performance equals or exceeds the required performance, then the requirement is *satisfied*.

1.4.1 Application Requirements

A real-time application typically has a set of timing-related requirements it should satisfy [25, 26]. Such applications often have links to peripherals of the SoC, i.e., the interface to the real world. For example, a SDR application might have to generate a response on the radio interface within a limited amount of time to correctly implement a communication protocol, or an adaptive cruise control system in a car might have to detect slowdowns of the surrounding traffic in time to avoid accidents.

The severity of the consequences of not meeting a deadline is usually expressed as a qualification on the real-time requirement, although the exact definitions vary. Using the definitions from [25, 27], we can distinguish *Hard real-time* (*HRT*) requirements and *Soft real-time requirements* (*SRT*). HRT requirements relate to hard deadlines that cannot be missed without severe loss of functionality for the application user. Sometimes, such requirements are called *critical* or *safety critical*, in case the safety of the user is not guaranteed and if the requirement is not satisfied. SRT requirements, on the other hand, may occasionally be missed, although this is still undesirable. In this book, we use *real-time* to refer to HRT, and we will not discuss SRT requirements. In the absence of real-time requirements, an application works on a *best-effort* basis.

Applications are not alone: instead, the increasing number of applications per SoC leads to a growing amount of (unintentional) interaction between them. This has an impact on the application's performance, and on how we deal with their requirements, as is discussed in the next sections.

1.4.2 Interference

An application that shares SoC resources is susceptible to *interference, i.e., its* (*functional and temporal*) *behavior and that of other applications become interdependent.* Only one application can use a shared resource at a time, leading to resource contention [28]. Other applications inevitably have to wait before they get access to the same resource, and hence experience *timing interference*. In an analogous manner, applications can experience *state interference*, which occurs when multiple applications change the state of a shared resource. As a straight-forward example, consider a memory in which one application overwrites the data of another application, changing its behavior in a potentially destructive manner.

Measures mitigating state interference for memory resources have been researched for quite some time [29], and solutions are available in the form of memory protection [30, 31]/management units [32] or data protection units. These modules effectively cordon off address ranges depending on the source of a memory access. Therefore, we focus on the timing aspect of interference in this book.

In most contexts, the word "interference" represents a negative effect, which is also the case here, since it changes the application's performance with respect to the case where there is no interference in an unpredictable way. In the following two sections, we describe two ways to qualify performance that are useful for real-time applications in the presence of interference.

1.4.3 Predictable Performance

Predictability is a qualification of a performance metric of an application or hardware component [33] that (partially) specifies the assumptions that were made when this performance metric was derived. *The predictable performance is a bound on all actual performances, assuming any possible initial condition, and including worst-case interference on shared resources without assumptions on the behavior of co-running applications* [34]. When we refer to guaranteed or worst-case performance in this book, it is implied that this performance is predictable. When a resource or methodology is predictable, then this means that predictable performance bounds can be derived for it or based on it.

Applications cannot communicate or share data (or else they would be part of the same application). If all resources an application uses provide predictable performance, then its worst-case performance is independent from other applications. The *verification* of the worst-case requirements of the application can then be done independently from other applications, i.e. it only has to consider the (predictable) hardware and the application itself. This reduces the complexity of this analysis compared to the unpredictable case, where all possible combinations of co-running applications also need to be factored in [35]. As such, it *enables incremental verification at a low relative cost, and reduces verification time*. Predictable performance enables model-based verification of requirements, using dataflow [36], network calculus [37], or other traditional real-time approaches, for example [38–40] outlined what this process can entail.

1.4.4 Composable Performance

Predictable performance is sufficient in cases where an application's verification is done based on a (formal) analysis of its worst-case requirements. However, there are cases where such an analysis is not possible, for example, when a model of the application's timing behavior is not available. Such applications might be verified by simulation, instead essentially by executing them with a number of test inputs and assessing the results. Two issues complicate this work flow in multi-application environments: (1) the performance of tested applications can only be definitely assessed after they are integrated with their co-running applications, and (2) the performance in conjunction with *all possible combinations* of co-running applications has to be verified [35, 41]. If requirements are not satisfied, or if *any* application or the system setup is changed for other reasons, then the verification process has to be repeated, making the entire process circular [42].

These issues are avoided by systems offering *composable performance, which means that the actual-case performance an application receives is not influenced by co-running applications* [43, 44]. This definition is strict: intuitively, it means that *a deviation of a single cycle from the actual timing behavior the application expressed when running in isolation qualifies as being non-composable.* Once an application receiving composable performance has been verified in isolation, it is guaranteed to also work correctly after integration, since its actual timing behavior during the verification is (exactly) the same as after integration. Composability is orthogonal to predictability, since it only implies independence of behavior, but by itself says nothing about the existence of worst-case bounds.[2]

[2] In practice, predictability is often used to provide composable performance [34].

1.5 Requirements for SDRAM Controllers in Modern SoCs

In the previous sections, we discussed why the SDRAM is a scarce resource, and how it is shared among more and more applications with diverse requirements. We also discussed the complications interference introduces the evaluation of real-time application performance. Based on these observations, we summarize the 5 main requirements on modern SDRAM controllers in the context of this book as follows:

1. *Some applications have real-time requirements.* To assure these requirements are met, the SDRAM controller must deliver predictable performance when the requirement verification is based on worst-case models, or composable performance, when the verification is based on simulation, or when a certification standard requires temporal isolation.
2. *Some applications benefit from improved average-case* (*typical*) *performance*, i.e., they can make use of all the resources the system can spare for them. Generally, an application's behavior or quality improves with additional resources, benefiting the user. Best-effort applications, i.e., applications without real-time requirements, generally fall within this category. Other examples include video decoding algorithms that support quality scaling [45], and user interfaces, for which higher responsiveness is generally better.
3. *Applications are not active all the time.* Instead, they can be transient, i.e., they only run in specific *use-cases*, in conjunction with a subset of the other applications the SoC supports. A memory controller should hence be able to efficiently deal with the changes in its set of clients.
4. SDRAM technology progresses quickly, and new generations are introduced every 2–3 years. An SDRAM controller architecture should hence be sufficiently flexible to handle the differences between the standards, such that it remains usable for a reasonable amount of time. The same requirement holds for the analysis on which its predictable performance guarantees are based. Ideally, a SoC should use the type of SDRAM that best fits its applications' requirements.
5. *The power budget is limited for* (*battery-powered*) *SoCs*. This requires a careful evaluation of how the SDRAM is used to minimize its power usage, while still satisfying the remaining performance requirements.

Existing memory controllers come in a variety of forms, some of which are a better fit for these requirements than others. We distinguish three categories:

1. Real time, i.e., geared toward maximizing predictable performance. Real-time controllers are designed such that their guaranteed performance is maximal. The underlying assumption is that only the worst-case performance matters, and anything that cannot be guaranteed and analyzed is wasted effort.
2. Best effort, i.e., geared toward maximizing average-case (typical) performance. They are built according to the philosophy that mechanisms that positively impact the average performance are considered worthwhile, even if they negatively affect the worst case. They exploit knowledge that is only available at run-time to make

request-level or command-level scheduling decisions. As a result, they typically provide no useful analytical bounds on performance.

3. Mixed time-criticality,[3] i.e., balancing the needs of real-time and best-effort applications. Ideally, these controllers guarantee sufficient performance to satisfy the worst-case performance requirements of the real-time applications, while maximizing the average-case performance for the best-effort applications.

When these descriptions are matched with the requirements we listed earlier, it is fairly obvious that best-effort controllers do not satisfy them, since they cannot guarantee (sufficient) performance to real-time applications. Complete isolation, as required for composability, is also practically impossible to achieve for these controllers, due to the complex interaction between the various average-case performance-improvement mechanisms, which inevitably leak state-information from one application to the other.

Real-time controllers, on the other hand, miss opportunities to improve the average-case performance that best-effort applications care about. It is hence not surprising that we consider a mixed time-criticality controller the best fit for the requirements. Such a controller is built upon concepts that are known to be real-time analyzable, while selectively using techniques from best-effort controllers to improve the average case.

1.6 Problem Statement and Contributions

The high-level question we answer in this book is:

How should a mixed time-criticality SDRAM controller be constructed that

(1) *provides predictable and composable performance to its real-time applications, both in terms of bandwidth, response time, and potentially within a limited power budget, while exploiting opportunities to improve the average-case performance for best-effort applications,*
(2) *is flexible, both in terms of architecture and worst-case performance analysis, such that it can be used for the various available SDRAM generations, and allows for comparisons between them, and*
(3) *retains these properties in the presence of transient applications, i.e., when it is used in multiple use-cases.*

Existing works on real-time SDRAM controllers focus on providing predictable performance, and some extend this with composable performance. Mixed time-criticality controllers improve average-case performance while retaining predictability. However, their scopes and capabilities are limited in certain areas. We highlight

[3]We focus on the diversity of timing requirements this book. In contrast, the broader term *mixed criticality* is typically used in works that deal with differences in certification requirements, including fault tolerance concerns, and varying degrees of pessimism in WCET estimations based on the required level of certainty [46].

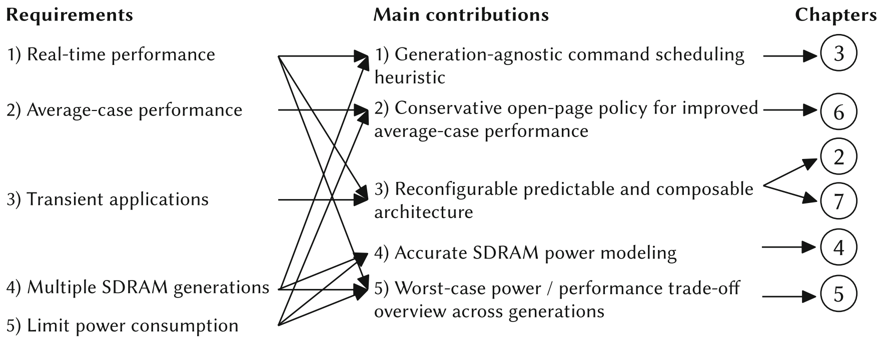

Requirements
1) Real-time performance
2) Average-case performance
3) Transient applications
4) Multiple SDRAM generations
5) Limit power consumption
Main contributions
1) Generation-agnostic command scheduling heuristic
2) Conservative open-page policy for improved average-case performance
3) Reconfigurable predictable and composable architecture
4) Accurate SDRAM power modeling
5) Worst-case power / performance trade-off overview across generations
Chapters
3
6
2
7
4
5

Fig. 1.4 Mapping of requirements to contributions and chapters

these issues in the following sections, and connect them to the contributions in this book that address them. A graphical representation of the mapping of requirements from the previous section to contributions and chapters is shown in Fig. 1.4. A detailed positioning of this book with respect to related work is given in Chap. 8.

1.6.1 Multi-generation Power-Aware Command Scheduling

The rapid development of SDRAM technology means multiple SDRAM generations are at a system designer's disposal at any given time. To select the right memory for a SoC that supports real-time applications, worst-case performance bounds of the memories need to be available and comparable, to satisfy Requirements 1 and 4. Existing real-time controllers and command scheduling algorithms are limited to a single memory device, or one or two memory generations, and ignore the impact of the command scheduling algorithm on the SDRAM power usage, even though this is an important design constraint for (battery-powered) embedded systems [20], as identified by Requirement 5.

To address these issues, *we provide an abstraction that allows us to write down an SDRAM command scheduling algorithm in a general fashion*, i.e., without targeting one specific memory device or generation. Using this abstraction, *we introduce a generation-agnostic command scheduling heuristic* (Chap. 3). The schedules it produces allows us to bound the SDRAM's performance. The quality of the heuristic is evaluated through a comparison with optimal solutions generated by an *integer linear programming* (*ILP*) formulation. We also provide a simple transformation for these schedules to turn the memory controller into a composable resource with negligible impact on the performance bounds.

We introduce a cycle-accurate SDRAM power model, which allows us to determine the power or energy spent in the SDRAM (Chap. 4). This model enables us to evaluate the effects of our scheduling decisions on the energy efficiency by which the memory is used. The command scheduling heuristic is parameterized, such that worst-case performance in terms of bandwidth and response time can be traded against worst-case power, We apply the heuristic to 12 memory devices from 6 different memory generations, and plot this trade-off space, effectively *providing an overview of worst-case power/performance tradeoffs across generations* (Chap. 5).

1.6.2 Improving Average-Case Performance Without Affecting Worst-Case Performance

Opportunities to improve (non-guaranteed) performance in real-time controllers are generally ignored, even though they could have a positive impact on both the application's performance and the power usage. Improving average-case performance and reducing power consumption is desirable, as mentioned in Requirements 2 and 5. Locality of reference influences how long it takes to read or write a unit of data. Memory controllers attempting to exploit locality across requests use an *open-page policy*, while those that do not use a *close-page policy* [47]. Open-page policies have only recently found their way into a few real-time and mixed time-criticality controllers, although they require special measures (bank privatization, explained in Chap. 8) to avoid worst-case performance reduction.

In this book, *we introduce a conservative open-page policy that improves the average-case performance without compromising on worst-case guarantees* (Chap. 6). It exploits locality of reference to reduce the response time of requests, like any open-page policy would. However, it only deviates from how a close-page policy would act when the response time is guaranteed to be smaller while doing so.

1.6.3 Reconfigurable Architecture

Existing controllers typically configure the behavior of the request-level arbiter and the command scheduler only once, when the SoC is booted. Therefore, this configuration has to cover all use-cases. Adapting the controller's behavior per use-case at run-time, to capture the arrival or departure of transient applications, is not considered, contradicting Requirement 3. If existing controllers would be reconfigured at run-time, they provide no bounds on performance to applications that remain active during these use-case transitions, which is unacceptable for real-time applications according to Requirement 1. Given how the number of applications per SoC is growing, it becomes paramount to specialize the controller configuration to the requirements of the active application set, in order to use the SDRAM in the most efficient manner.

To address this, *we introduce a reconfigurable SDRAM controller architecture template* (*Chap.* 2), and a proof-of-concept implementation, integrated in a predictable and composable SoC. The performance that is provided to each controller port is characterized by a worst-case analysis, and can be changed at run-time through reconfiguration. This allows for specialization of the controller on a per use-case basis, making it easier to swap transient applications in and out without having to (over) dimension for the most challenging super-set of requirements. The rules that have to be respected to retain predictability or composability for (real-time) applications that are active during reconfiguration are discussed in Chap. 7. We demonstrate how to implement these rules for a *Time-Division Multiplexing* (*TDM*) arbiter, and prove that our reconfiguration protocols are safe.

1.7 Outline

This remainder of this book is structured as follows. Chapter 2 introduces the necessary background information on SDRAM memories and SDRAM controllers. It also shows the architecture template and the worst-case analysis of the memory controller we propose, and looks at one concrete instance of the template in more detail. Chapter 3 discusses how both predictable and composable configurations for the command scheduler in this memory controller can be generated. The used algorithms are transparently applicable to all contemporary SDRAM generations through the introduction of a simple abstraction layer. Chapter 4 discusses the cycle-accurate SDRAM power model we use in the remainder of the book. In Chap. 5, we provide an overview of the worst-case power and performance tradeoffs in 12 memory devices in 6 different memory generations as a function of the different command scheduler parameters. Chapter 6 introduces a mechanism that improves average-case

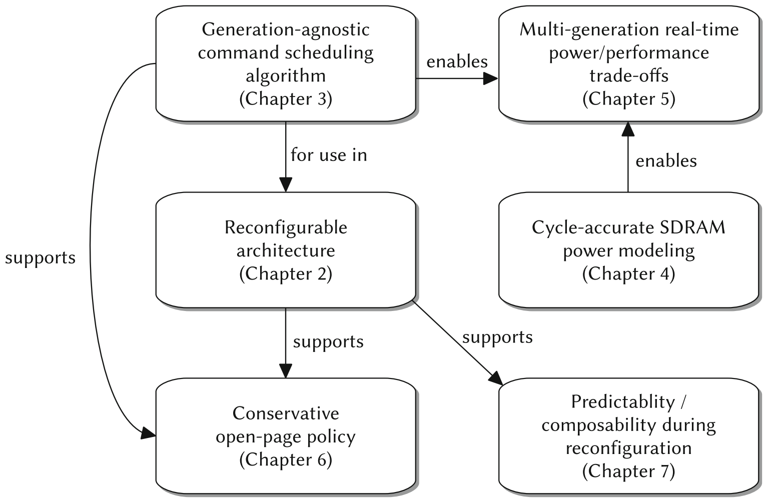

Generation-agnostic command scheduling algorithm (Chapter 3)
enables
Multi-generation real-time power/performance trade-offs (Chapter 5)
for use in
enables
Reconfigurable architecture (Chapter 2)
Cycle-accurate SDRAM power modeling (Chapter 4)
supports
supports
supports
Conservative open-page policy (Chapter 6)
Predictablity / composability during reconfiguration (Chapter 7)

Fig. 1.5 Overview of chapters

performance without sacrificing worst-case guarantees. Chapter 7 then shows when and how the various configurable components in the memory controller can be reconfigured without violating predictable or composable performance bounds. Figure 1.5 shows the relations between Chaps. 2–7. Finally, Chap. 8 relates this book to the state of the art, and we end with conclusions and future work in Chap. 9.

References

1. Miller GA (1956) The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychol Rev 63(2):81
2. Moore G (1965) Cramming more components onto integrated circuits. Electron Mag 38
3. Dennard RH, Gaensslen FH, Rideout VL, Bassous E, LeBlanc AR (1974) Design of ion-implanted MOSFET's with very small physical dimensions. IEEE J Solid-State Circuits 9(5):256–268
4. Esmaeilzadeh H, Blem E, St.Amant R, Sankaralingam K, Burger D (2011) Dark silicon and the end of multicore scaling. In: 2011 38th Annual international symposium on computer architecture (ISCA), pp 365–376
5. ITRS (2013) International technology roadmap for semiconductors (ITRS) - system drivers abstract
6. Henkel J (2003) Closing the SOC design gap. Computer 36(9):119–121
7. Lyne K (2005) Cellular handset integration - sip vs. SOC and best design practices for SIP. In: Custom integrated circuits conference, 2005. Proceedings of the IEEE 2005, pp 765–770
8. Matick R, Schuster S (2005) Logic-based eDRAM: origins and rationale for use. IBM J Res Dev 49(1):145–165
9. ITRS (2012) International technology roadmap for semiconductors (ITRS) - assembly & packaging, 2012 tables
10. Wulf WA, McKee SA (1995) Hitting the memory wall: implications of the obvious. SIGARCH Comput. Architect. News 23(1)
11. McKee S (2004) Reflections on the memory wall. In: Proceedings of the conference on computing frontiers, pp 162–167
12. Jeddeloh J, Keeth B (2012) Hybrid memory cube new DRAM architecture increases density and performance. In: 2012 symposium on VLSI technology (VLSIT), pp 87–88
13. Xie Y (2013) Future memory and interconnect technologies. In: Design, automation and test in Europe conference and exhibition (DATE), pp 964–969
14. Tummala R (2006) Moore's law meets its match (system-on-package). IEEE Spect 43(6):44–49
15. Snapdragon 800 (2015) Snapdragon 800 processor specs. https://www.qualcomm.com/products/snapdragon/processors/800. Online; Accessed 30 Mar 2015
16. anandtech.com (2014) Apple a8x SOC. http://www.anandtech.com/show/8716/apple-a8xs-gpu-gxa6850-even-better-than-i-thought
17. DM-D large-size commercial LED LCD displays (2015) Samsung Electronics America Inc
18. Broy M (2006) Challenges in automotive software engineering. In: International conference on software engineering, pp 3–42
19. Ramacher U (2007) Software-defined radio prospects for multistandard mobile phones. Computer 40(10):62–69
20. van Berkel (C) Multi-core for mobile phones. In: Design, automation and test in Europe conference and exhibition (DATE)
21. Ward L, Simon M (2015) Intelligent transportation systems using IEEE 802.11p (application note 6.2015 - 1MA153_4e)
22. Festag A (2014) Cooperative intelligent transport systems standards in Europe. IEEE Commun Mag 52(12):166–172

23. Ross PE (2014) Europes smart highway will shepherd cars from Rotterdam to Vienna. http://spectrum.ieee.org/transportation/advanced-cars/europes-smart-highway-will-shepherd-cars-from-rotterdam-to-vienna
24. Jacob B, Ng S, Wang D (2007) Memory systems: cache, DRAM, disk. Morgan Kaufmann Pub
25. Steffens L, Agarwal M, der Wolf PV (2008) Real-time analysis for memory access in media processing SOCs: a practical approach. In: Euromicro conference on real-time systems (ECRTS), pp 255–265
26. Bekooij M, Moonen A, van Meerbergen J (2007) Predictable and composable multiprocessor system design: a constructive approach. In: Bits and chips symposium on embedded systems and software
27. Buttazzo G, Lipari G, Abeni L, Caccamo M (2005) Soft real-time systems: predictability vs. efficiency. Series in computer science. Springer
28. Mutlu O, Moscibroda T (2007) Memory performance attacks: denial of memory service in multi-core systems. In: USENIX security
29. Wilkes MV (1982) Hardware support for memory protection: capability implementations. In: Proceedings of the 1st international symposium on architectural support for programming languages and operating systems, pp 107–116
30. Application note 179 cortex - M3 embedded software development (2007) ARM Limited
31. RM57L843 16- and 32-Bit RISC flash microcontroller (2014) Texas Instruments Inc
32. Xilinx (2011) Microblaze processor reference guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/mb_ref_guide.pdf
33. Cullmann C, Ferdinand C, Gebhard G, Grund D, Maiza C, Reineke J, Triquet B, Wegener S, Wilhelm R (2010) Predictability considerations in the design of multi-core embedded systems. Ingénieurs de l'Automobile 807:36–42
34. 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
35. Rumpler B (2006) Complexity management for composable real-time systems. In: Proceedings of ISORC
36. Sriram S, Bhattacharyya S (2000) Embedded multiprocessors: scheduling and synchronization. CRC
37. Cruz RL (1991) A calculus for network delay. I. Network elements in isolation. IEEE Trans Inf Theory 37(1)
38. Richter K, Jersak M, Ernst R (2003) A formal approach to MPSOC performance verification. Computer 36(4):60–67
39. Nelson A, Goossens K, Akesson B (2015) Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. J Syst Architect
40. 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
41. Kopetz H, El Salloum C, Huber B, Obermaisser R, Paukovits C (2008) Composability in the time-triggered system-on-chip architecture. In: 2008 IEEE international SOC conference, pp 87–90
42. Saleh R, Wilton S, Mirabbasi S, Hu A, Greenstreet M, Lemieux G, Pande P, Grecu C, Ivanov A (2006) System-on-chip: reuse and integration. Proc IEEE 94(6):1050–1069
43. 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)
44. Puschner P, Kirner R, Pettit R (2009) Towards composable timing for real-time programs. In: 2009 software technologies for future dependable distributed systems, pp 1–5

45. Wu D, Hou YT, Zhang Y-Q (2001) Scalable video coding and transport over broadband wireless networks. Proc IEEE 89(1):6–20
46. Vestal S (2007) Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: Real-time systems symposium, pp 239–243
47. Rixner S, Dally WJ, Kapasi UJ, Mattson P, Owens JD (2000) Memory access scheduling. In: International symposium on computer architecture (ISCA), pp 128–138

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

S. Goossens et al., *Memory Controllers for Mixed-Time-Criticality Systems*, Embedded Systems, DOI 10.1007/978-3-319-32094-6_2

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.

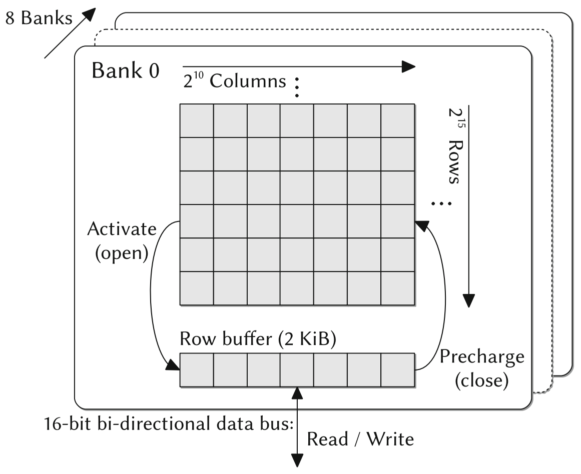


Fig. 2.1 Schematic view on the architecture of an SDRAM device with the dimensions of a 512 MiB DDR3-1600 chip (see Appendix B)

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 800 MHz 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

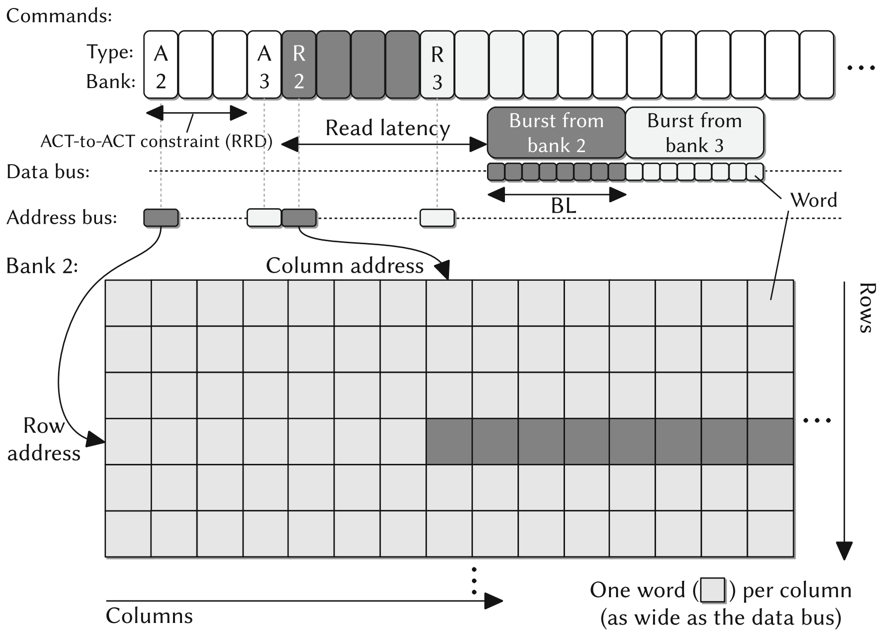


Fig. 2.2 High-level SDRAM operation. The activation of bank 3 happens in parallel with the read command to bank 2. Data bursts of different banks are serialized, since the data bus is shared across banks. The two cycles between A2 and A3 are the result of the ACT-to-ACT timing constraint (RRD)

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.

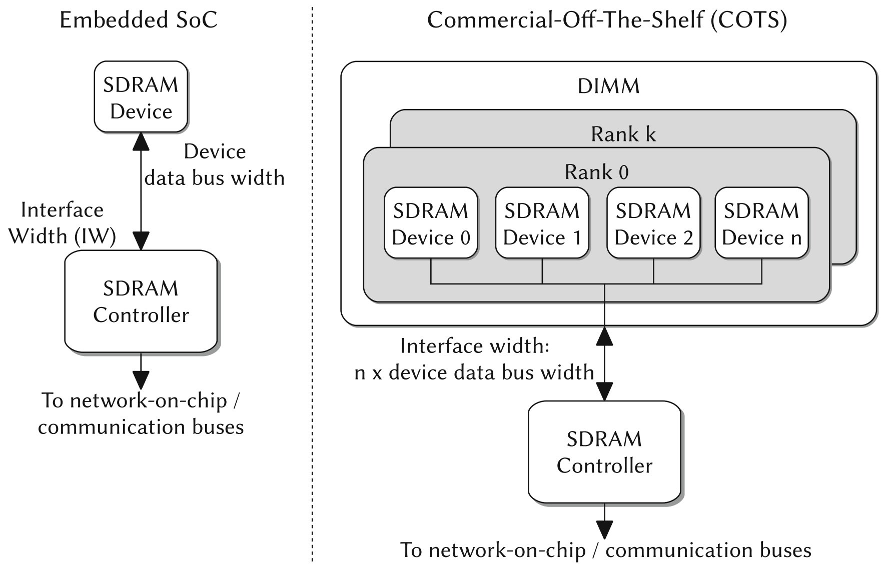


Fig. 2.3 Typical memory hierarchy for embedded SoCs and COTS systems

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.

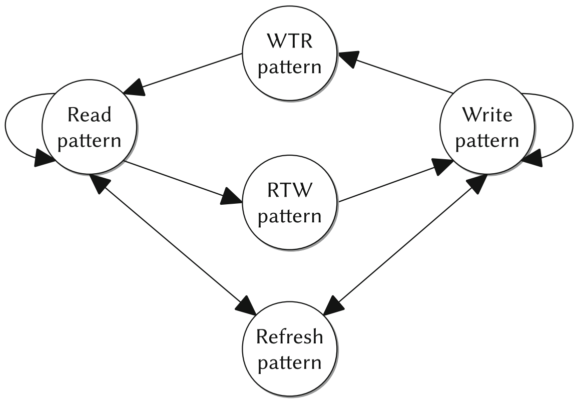


Fig. 2.4 Allowed pattern sequences

- 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

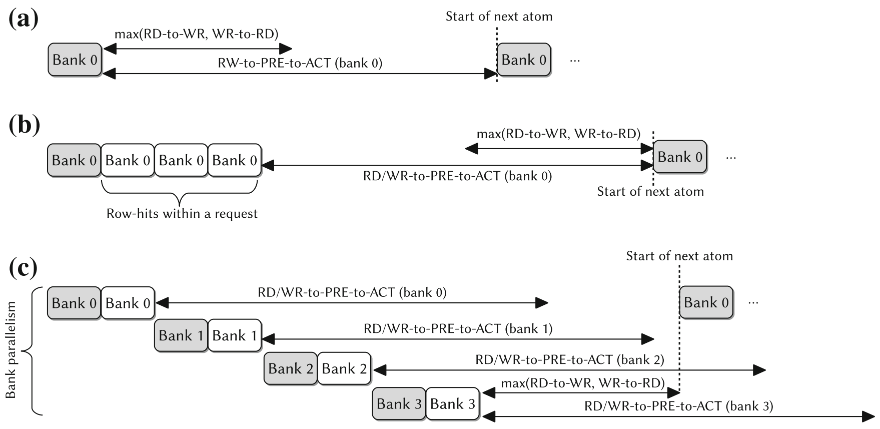


Fig. 2.5 Examples of the effects of grouping bursts. *Shaded* bursts are page misses. It shows how the number of bursts that can be executed within a fixed amount of time varies based on how they are grouped. **a** Using BI 1, BC 1. **b** BI 1, BC 4. **c** BI 4, BC 2

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 $\mathrm{BI} \cdot \mathrm{BC}$, 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):

$$\mathrm{AG} = \mathrm{BI} \cdot \mathrm{BC} \cdot \mathrm{BL} \cdot \mathrm{IW} \tag{2.1}$$

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.

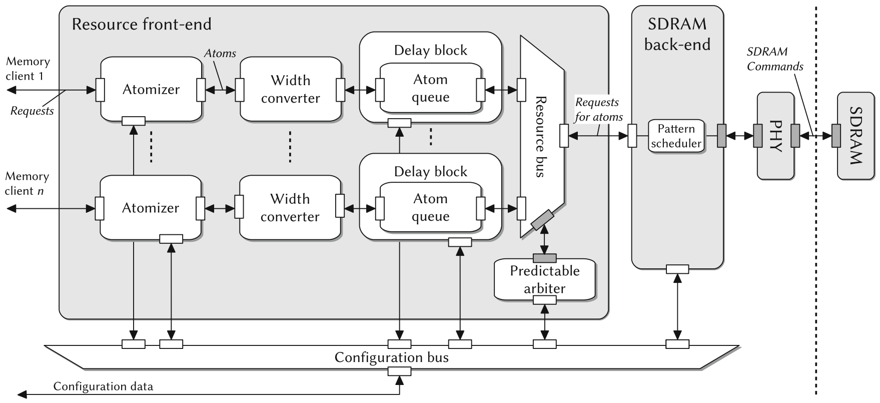


Fig. 2.6 SDRAM controller architecture. *Arrows* indicate the flow direction of data

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 though 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

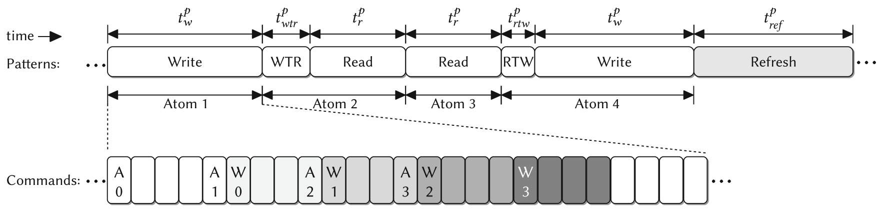


Fig. 2.7 An example of the order in which patterns may be executed. The *shading* on the commands corresponds to bursts of data to different banks

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

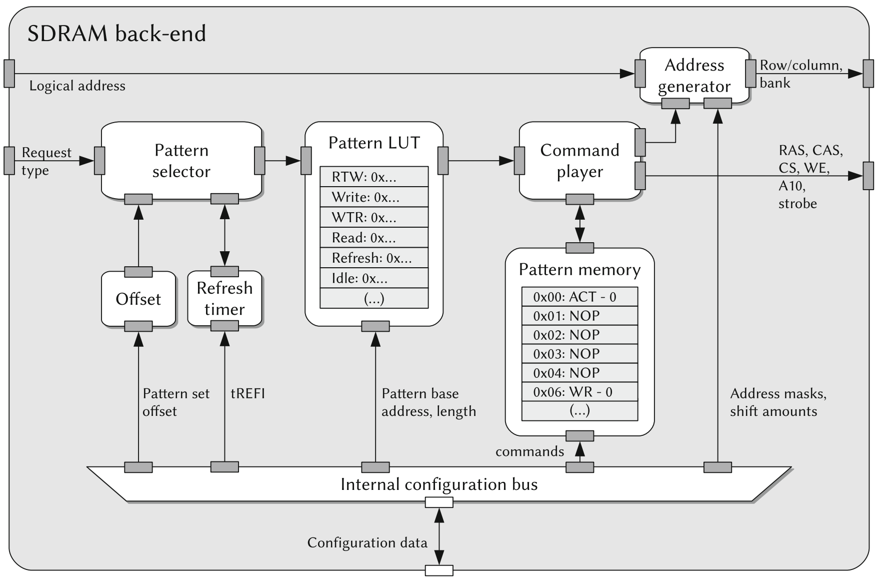


Fig. 2.8 SDRAM controller back-end

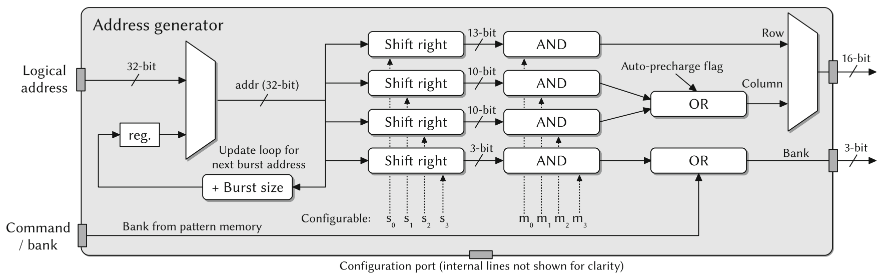


Fig. 2.9 Address generator. Both the shift amounts (s_0–s_3) and the masks (m_0–m_3) used by the and-operators are configurable. (The **and**-operators and **or**-operators are bitwise.) The sizes of the row, column, and bank components correspond to the ML605 memory (Appendix B)

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:

$$row = (addr \gg s_0)\ \textbf{and}\ m_0 \tag{2.2}$$

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

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

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 $\delta^{b}_{\mathrm{PHY}}$), 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

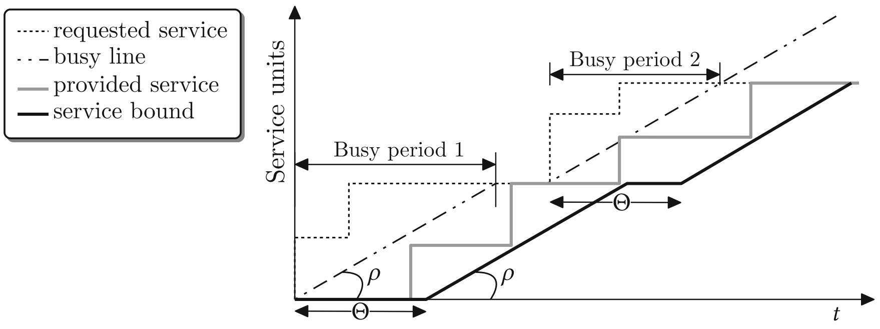


Fig. 2.10 A $\mathcal{LR}$ server and its associated concepts

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").

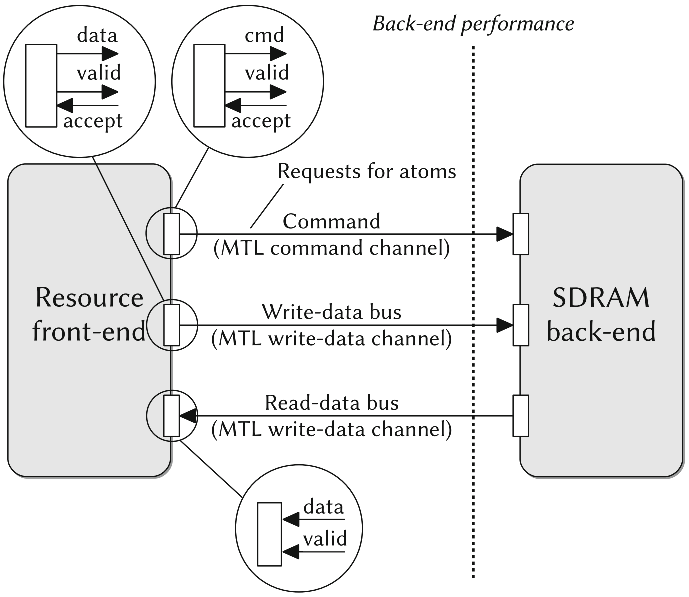


Fig. 2.11 The interface characterized by the back-end performance. The call-outs on the MTL channels show the relevant groups of wires they consist of

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 from/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

$$e_{ref} = 1 - \frac{t_{ref}^{p}}{\text{REFI}} \tag{2.5}$$

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 $2 \cdot \mathrm{AG}$ 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 $\left(e_{ref}\right)$ (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:

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

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

$$b_{peak} = 2 \cdot f \cdot \mathrm{IW} \tag{2.7}$$

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

$$e = \frac{b_{wc}}{b_{peak}} \tag{2.8}$$

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^f_{be} 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.

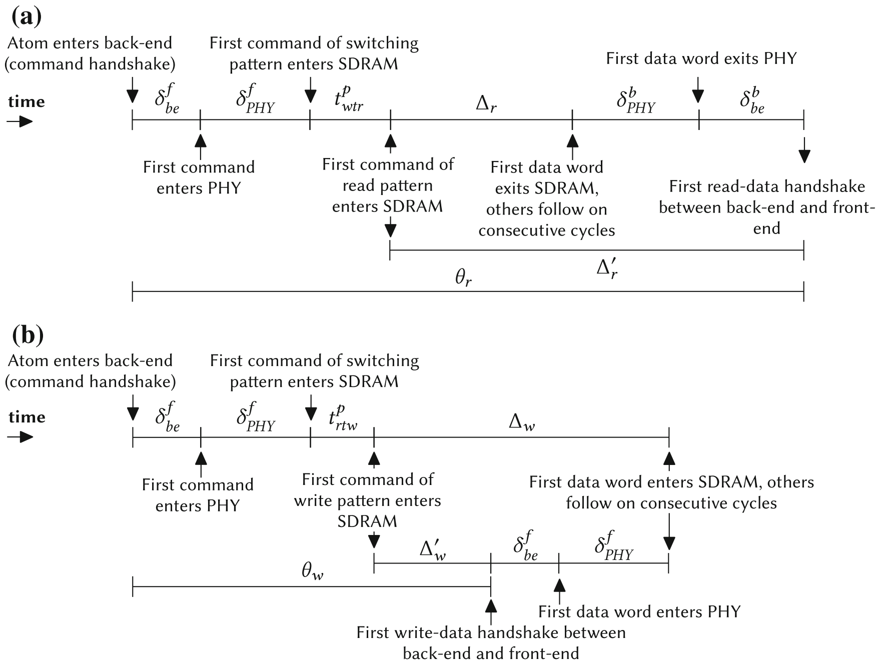

(a)
Atom enters back-end (command handshake)
First command of switching pattern enters SDRAM
First data word exits PHY
time
δ^f_{be}
δ^f_{PHY}
t^p_{wtr}
Δ_r
δ^b_{PHY}
δ^b_{be}
First command enters PHY
First command of read pattern enters SDRAM
First data word exits SDRAM, others follow on consecutive cycles
First read-data handshake between back-end and front-end
Δ'_r
θ_r
(b)
Atom enters back-end (command handshake)
First command of switching pattern enters SDRAM
time
δ^f_{be}
δ^f_{PHY}
t^p_{rtw}
Δ_w
First command enters PHY
First command of write pattern enters SDRAM
First data word enters SDRAM, others follow on consecutive cycles
Δ'_w
δ^f_{be}
δ^f_{PHY}
θ_w
First data word enters PHY
First write-data handshake between back-end and front-end

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^f_{PHY} 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^b_{PHY} 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^b_{be} 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:

$$\delta_f = \delta^f_{be} + \delta^f_{\mathrm{PHY}} \tag{2.9}$$

$$\delta_b = \delta^b_{be} + \delta^b_{\mathrm{PHY}} \tag{2.10}$$

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

$$\Delta'_r = \Delta_r + \delta^b \tag{2.11}$$

$$\Delta'_w = \Delta_w - \delta^f \tag{2.12}$$

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

[2]If 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.

$$\theta_r = \delta^f + t^p_{wtr} + \Delta'_r = t^p_{wtr} + \Delta_r + \delta^f + \delta^b \tag{2.13}$$

$$\theta_w = \delta^f + t^p_{rtw} + \Delta'_w = t^p_{rtw} + \Delta_w \tag{2.14}$$

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.

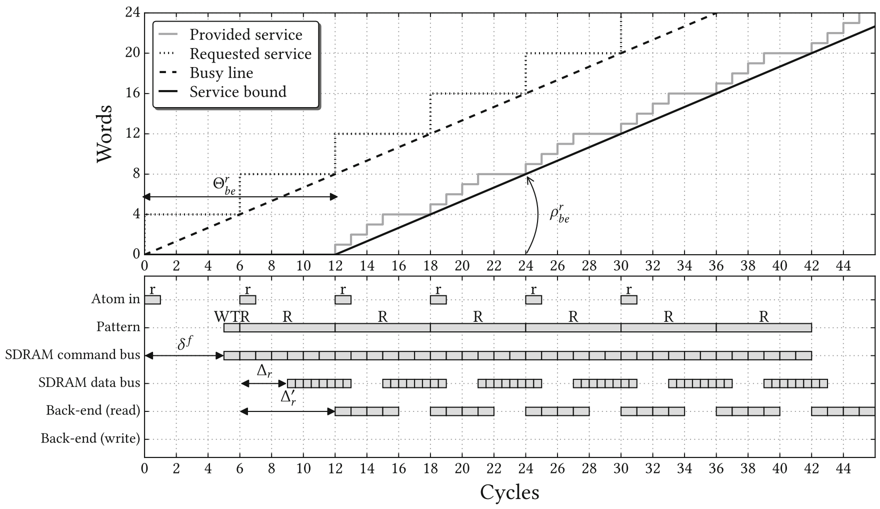


Fig. 2.13 Worst-case back-end behavior for continuous reads. In this (fictional) example, we used: $t^p_r = 6$, $t^p_w = 8$, $t^p_{rtw} = 3$, $t^p_{wtr} = 1$, $\Delta_r = 3$, $\Delta_w = 2$, $\delta^f = 5$, $\delta^b = 3$, and each atom is worth 4 words. To simplify the drawing, we assume $e_{ref} =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

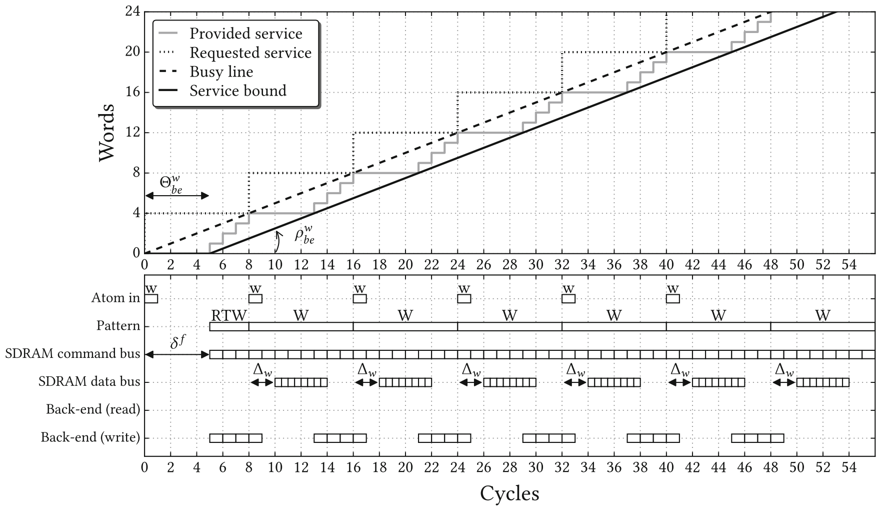


Fig. 2.14 Worst-case back-end behavior for continuous writes, using the same parameters as Fig. 2.13

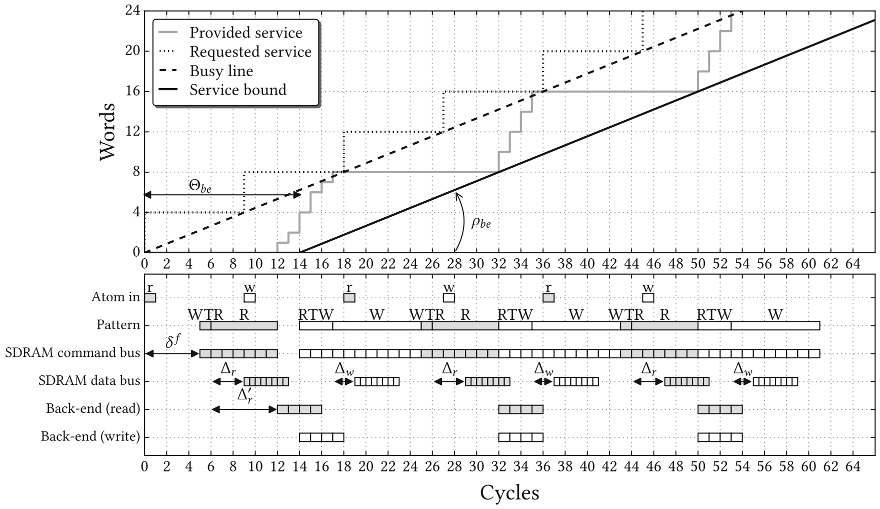


Fig. 2.15 Worst-case back-end behavior for interleaved read/write atoms, using the same parameters as Fig. 2.13

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:

$$\rho_{be} = b_{wc} \tag{2.15}$$

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

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

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*):

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

When WCSI > 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

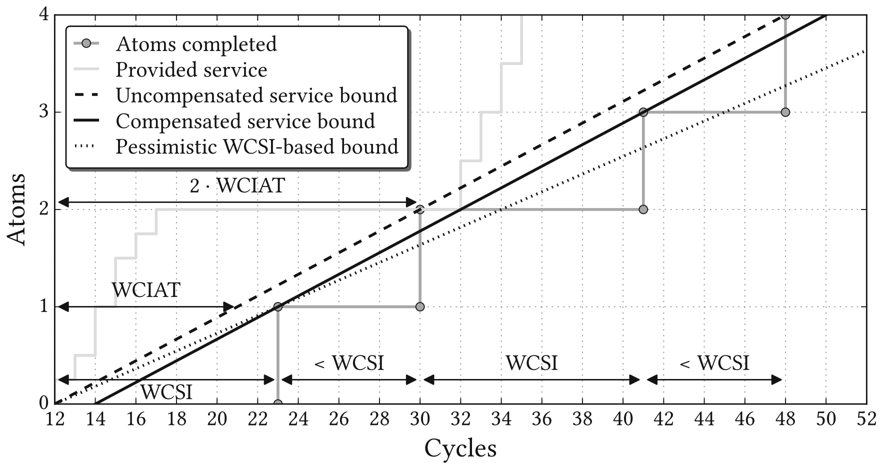


Fig. 2.16 Demonstration of latency compensation for WCSI, using the same parameters as Fig. 2.13. The compensated service bound is conservative in cycles 30 and 31, while the uncompensated service bound is not. Note that the x-axis starts at $\max(\theta_r, \theta_w)$

the time required for each possible pair of atoms is conservatively bounded by the (average) WCIAT. To achieve this, we add WCSI − 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 WCSI > WCIAT. Finally, the expression for Θ_{be} is equal to:

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

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, $\left(\Theta^c_{arb}, \rho^c_{arb}\right)$. These parameters are normalized, such that ρ^c_{arb} represents the fraction of the total server bandwidth that a client receives after it has waited for Θ^c_{arb} 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.

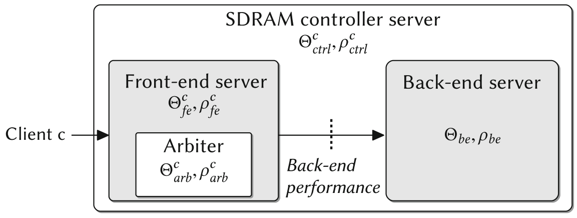


Fig. 2.17 The $\mathcal{LR}$ server describing the memory controller's performance is the concatenation of the front-end server and the back-end server

We characterized the front-end for client c as another $\mathcal{LR}$ server with parameters $\left(\Theta^c_{fe}, \rho^c_{fe}\right)$. ρ^c_{fe} represents the bandwidth that is allocated to the client.

$$\rho^c_{fe} = \rho^c_{arb} \cdot \rho_{be} \quad | \quad 0 < \rho^c_{arb} \leq 1 \tag{2.19}$$

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

Finally, we de-normalize Θ^c_{fe} 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}:

$$\Theta^c_{fe} = \left\lceil \Theta^c_{arb} \right\rceil \cdot \text{WCIAT} + \delta_{fe} \tag{2.20}$$

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 $\left(\Theta^c_{ctrl}, \rho^c_{ctrl}\right)$ to represent the combined server (Fig. 2.17)

$$\Theta^c_{ctrl} = \Theta^c_{fe} + \Theta_{be} \tag{2.21}$$

$$\rho^c_{ctrl} = \min\left(\rho^c_{fe}, \rho_{be}\right) = \rho^c_{fe} \tag{2.22}$$

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:

$$\mathrm{WCRT}(s) = \Theta^c_{ctrl} + \frac{s}{\rho^c_{ctrl}} \tag{2.23}$$

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 **a**nd **p**redic**t**able **o**pen-page controlle**r**, 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.

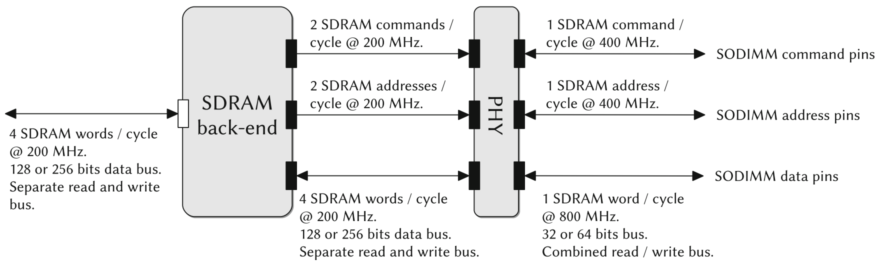

SDRAM back-end
PHY
4 SDRAM words / cycle @ 200 MHz. 128 or 256 bits data bus. Separate read and write bus.
2 SDRAM commands / cycle @ 200 MHz.
2 SDRAM addresses / cycle @ 200 MHz.
4 SDRAM words / cycle @ 200 MHz. 128 or 256 bits data bus. Separate read and write bus.
1 SDRAM command / cycle @ 400 MHz.
1 SDRAM address / cycle @ 400 MHz.
1 SDRAM word / cycle @ 800 MHz. 32 or 64 bits bus. Combined read / write bus.
SODIMM command pins
SODIMM address pins
SODIMM data pins

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 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:

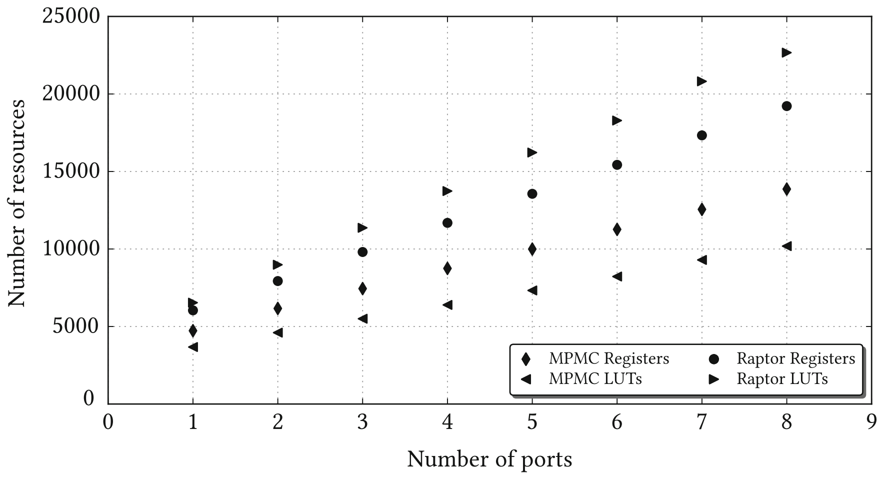


Fig. 2.19 Resource usage of Raptor versus MPMC using 512 byte read/write queues (1024 bytes in total) per port

- 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.

[4]Compared 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].

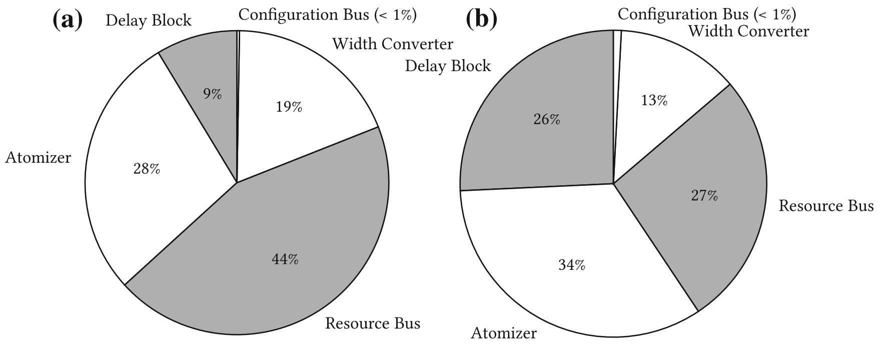


Fig. 2.20 Front-end LUT and register usage break-down per port. 100 % = 1915 registers, 2837 LUTs. **a** Registers. **b** LUTs

- 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.

[5]32 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, Šůcha 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 - MT4JSF6464H - 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

Chapter 3
Memory Patterns

The previous chapter outlined the foundation of the mixed-time-criticality SDRAM controller by showing the hardware it consists of. In the process, it introduced multiple storage elements that reside in the memory controller back-end, like the pattern memory and the registers in the address generator. In this chapter, we focus on the creation of *configurations* for the controller back-end that fill these storage elements.

A pattern memory, as the name suggests, contains memory patterns (Sect. 2.2), which are groups of prescheduled SDRAM commands that the controller selects and executes at run-time. Our goal is to provide a memory controller template that is not bound to a specific memory device or SDRAM type, and hence the memory pattern-generation algorithm should be written in such a way that it is easily transferable across SDRAM types (DDR2/3/4 and LPDDR1/2/3). Section 3.1 describes an abstraction step that allows us to do this. Section 3.2 applies this abstraction in the form of a parameterized pattern-generation heuristic, which is later refined to improve its effectiveness for DDR4 memories. As an alternative, we provide the option to generate patterns that are guaranteed to be optimal, although this is more time consuming. The connection between the pattern configuration and memory map is also explained in this section. Figure 3.1 shows a flowchart of the pattern generation and post-processing steps.

A predictable pattern set can be converted into a composable pattern set, as we show in Sect. 3.3, hence providing an alternative to the *delay blocks* (Sect. 2.3.1) as a method of creating a composable memory resource. We show that the impact on the memory efficiency can be expressed as a function of the original pattern lengths.

At the end of this chapter, in Sect. 3.4, we first introduce the set of memory devices that are used to evaluate the effectiveness of the pattern-generation heuristics. We compare the heuristics with the optimal solution to evaluate the quality of their solutions to the pattern-generation problem. The produced schedules are also the basis for the worst-case performance evaluation in Chap. 5. The same range of memory devices is used to quantify the typical conversion efficiency from predictable to composable patterns. Finally, we use the FPGA instance of our memory controller to demonstrate how using composable patterns isolates the timing behavior of two co-running applications.

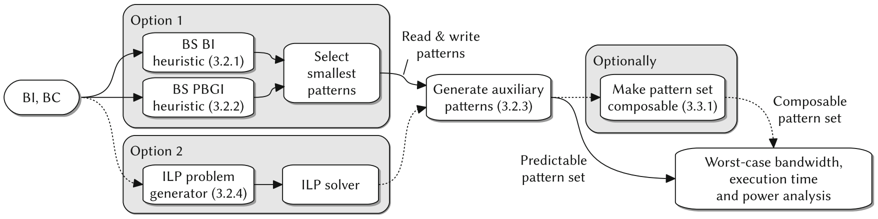

Option 1
BS BI heuristic (3.2.1)
BS PBGI heuristic (3.2.2)
Select smallest patterns
BI, BC
Option 2
ILP problem generator (3.2.4)
ILP solver
Read & write patterns
Generate auxiliary patterns (3.2.3)
Optionally
Make pattern set composable (3.3.1)
Composable pattern set
Predictable pattern set
Worst-case bandwidth, execution time and power analysis

Fig. 3.1 The pattern flow in this chapter. The related section numbers are written in *round brackets*

3.1 Generalized Command Scheduling Rules

SDRAM command scheduling is an old problem, and solutions to this problem are (almost) equally old: an SDRAM cannot be used without a controller that schedules commands to it. Even though the number of popular SDRAM standards introduced in the past 10 years is large [1–6], their structural differences from the scheduler's perspective are fortunately quite small, although this is not immediately apparent when the standards are compared. The numerical values of the timings that govern the memory behavior vary, timings are renamed, and certain constraints are expressed differently across generations, obfuscating the similarities. These factors complicate the reuse of command scheduling algorithms and make it hard to compare them structurally, since the details of a particular SDRAM standard are often intermixed with the actual algorithm. Even though individual timings may be required for the (circuit-level) SDRAM design or detailed modeling of power usage, for example, they are of little use for command schedulers (and arguably for memory controllers as a whole), and can hence be hidden from them. Instead, all they need to know is *the minimum delay between pairs of commands*.

For this purpose, we introduce a function d, which serves as the interface for these algorithms to obtain the minimum relative delay between two commands, cmd_a and cmd_b. Based on five properties of these commands and the SDRAM type, it is conceptually a lookup table that determines the delay. The first two properties describe the types of the commands, $type_a, type_b \in \{\text{ACT, RD, WR, PRE, REF}\}$, while the remaining boolean properties specify the relative physical location of the bank at which the commands are targeted, i.e., if they go to the same or a different rank, bank group (DDR4) or bank, respectively:

$$\begin{aligned} d_{sdramType}(cmd_a, cmd_b) = LUT_{sdramType}(&type_a, type_b,\\ &sameRank(cmd_a, cmd_b),\\ &sameBankGroup(cmd_a, cmd_b),\\ &sameBank(cmd_a, cmd_b)) \end{aligned} \tag{3.1}$$

Based on the JEDEC specifications [1–6], we collected the delays that this function should produce for six SDRAM generations in Tables 3.1 and 3.2, effectively

Table 3.1 Common constraints across SDRAM types (definition of $d()$)

$type_a$	$type_b$	sameBank	Constraint
ACT	ACT	True	RC
ACT	ACT	False	RRD_X
ACT	PRE	True	RAS
ACT	RD/WR	True	RCD - AL
PRE	ACT	True	RP
PRE	REF	True or false	RP
REF	ACT	True or false	RFC

condensing hundreds of pages of documentation into the bare minimum required to create memory command scheduling algorithms. Since expressing a five-dimensional lookup table compactly on a two-dimensional page is quite challenging, a few notational shortcuts are applied:

- If a combination of inputs is not mentioned in the tables, then it is either unconstrained, or not allowed by the state machine of an SDRAM bank. For example, a read followed by an activate to the same bank is not mentioned, since they should at least be separated by a precharge command.
- The timings that are postfixed with _x depend on the $sameBankGroup$ argument. For DDR4, an implementation of $d()$ selects the _L or _S versions of the timing if $sameBankGroup$ is true or false, respectively, as required by the specification [4]. For other SDRAM types, the non-postfixed version of the timing is used.
- The *Four Activate Window* (*FAW*) timing is not mentioned in the table, because it is a window-based constraint, and not a simple delay. It has to be taken into account separately on a per-rank basis for all SDRAM types except LPDDR, which has no FAW constraint.

For brevity, and because multi-rank operation is not standardized, the tables only show constraints for the cases where the command pair is sent to the same rank. Commands across ranks are generally not constrained, unless they use the (shared) data bus, i.e., RD and WR commands. An additional delay, typically one or two cycles, has to be taken into account in those cases to make sure only one rank at a time drives the bus.

By writing scheduling algorithms in terms of calls to the $d()$ function instead of referring directly to timing constraints, they can be written in a compact, SDRAM-type-agnostic manner, as later demonstrated in Algorithm 2. To create an SDRAM-type specific instance of a schedule, one only has to substitute the relevant device-specific timings in the constraints in the tables, and resolve the $d()$-calls the scheduler makes. Figure 3.2 illustrates this process.

Table 3.2 SDRAM-type specific constraints (definition of $d()$)

Memory type	$type_a$	$type_b = PRE$ sameBank = true	$type_b = RD$ sameBank = true or false	$type_b = WR$ sameBank = true or false
LPDDR	RD	B	B	B + CL
LPDDR	WR	B + DQSS + WR	B + DQSS + WTR	B
DDR2	RD	B + AL − 2 + max(RTP, 2)	B	B + RTW
DDR2	WR	B + WL + WR	B + CL − 1 + WTR	B
DDR3	RD	AL + max(RTP, 4)	B	B + CL − CWL + 2
DDR3	WR	B + CWL + AL + WR	B + CWL + WTR	B
DDR4	RD	AL + RTP	CCD_X	B + CL − CWL + PA
DDR4	WR	B + CWL + AL + WR	B + CWL + WTR_X	CCD_X
LPDDR2/3	RD	B + max(0, RTP − D)	B	B + RL − WL + DQSCK $_{MAX}$ + 1
LPDDR2/3	WR	B + WL + WR + 1	B + WL + WTR + 1	B

B represents the burst transfer time, equal to BL/2. For DDR2, if BL = 4, RTW = 2, and if BL = 8, RTW = 6. For DDR4, PA depends on the selected read/write preamble. If a read or write preamble of 1 cycle is used, PA = 2. With a preamble of two cycles, PA = 3. For LPDDR2/3, D = 1, 2, or 4 for LPDDR2-S2, LPDDR2-S4, or LPDDR3 devices, respectively

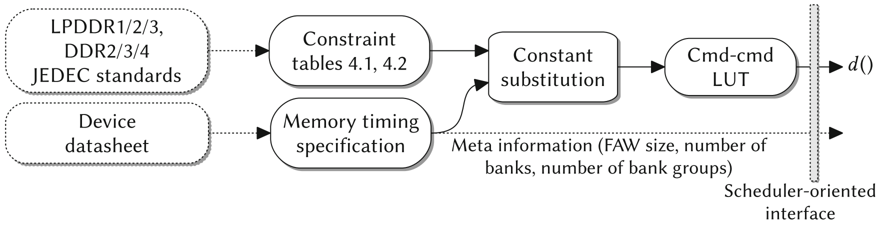
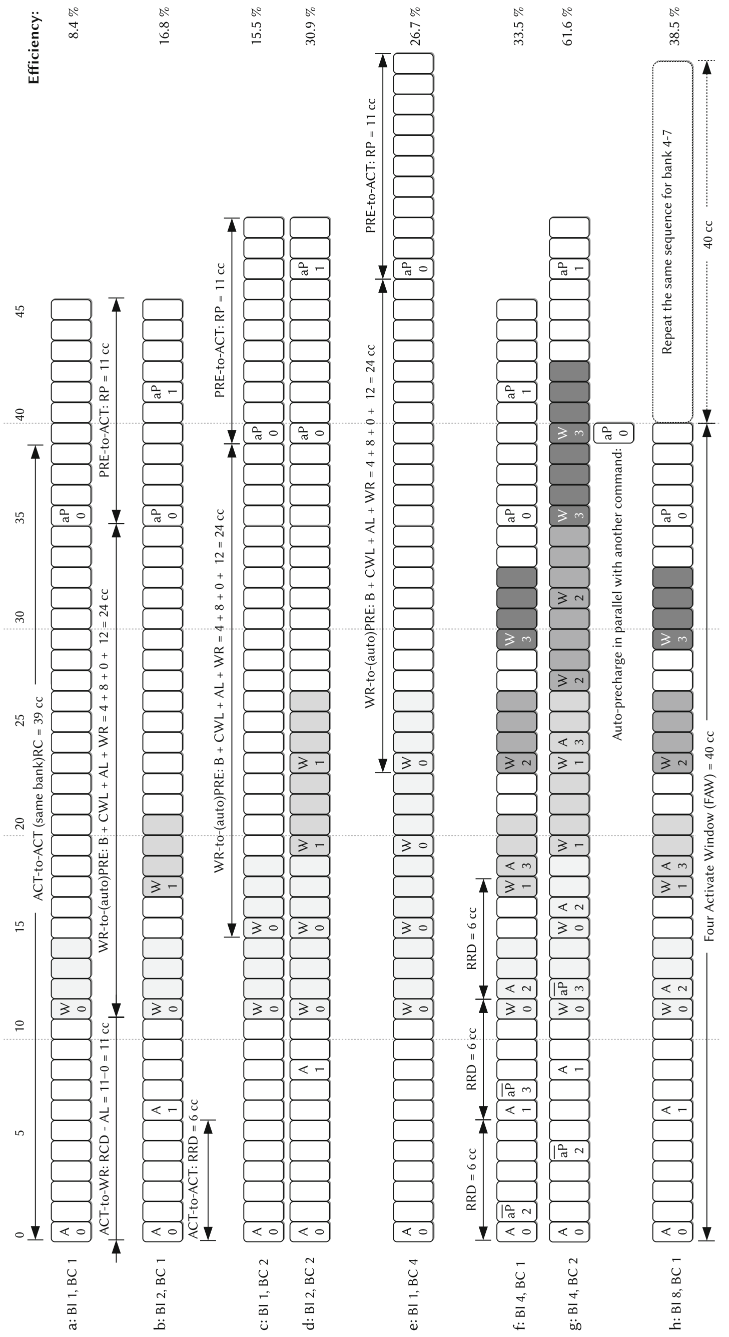

LPDDR1/2/3, DDR2/3/4 JEDEC standards
Device datasheet
Constraint tables 4.1, 4.2
Memory timing specification
Constant substitution
Cmd-cmd LUT
d()
Meta information (FAW size, number of banks, number of bank groups)
Scheduler-oriented interface

Fig. 3.2 Constraint abstraction

3.2 Predictable Patterns

Commands for all devices in the previously mentioned SDRAM generations can be scheduled by respecting the constraints in Sect. 3.1. By means of the examples in Fig. 3.3, we discuss some of the options the back-end has when it comes to deciding *which* commands to generate and schedule. The semantics of the figure are explained first, and then we look at its content.

Each block in this figure represents a command, and each line of blocks represents a possible schedule for a DDR3-1600 device. A block may contain a letter representing the command type, and a number, representing the bank to which the command is directed. Empty blocks represent NOP commands. Shaded blocks rep-



Efficiency:
0
5
10
15
20
25
30
35
40
45
a: BI 1, BC 1
8.4 %
ACT-to-ACT (same bank)RC = 39 cc
ACT-to-WR: RCD - AL = 11−0 = 11 cc
WR-to-(auto)PRE: B + CWL + AL + WR = 4 + 8 + 0 + 12 = 24 cc
PRE-to-ACT: RP = 11 cc
b: BI 2, BC 1
16.8 %
ACT-to-ACT: RRD = 6 cc
c: BI 1, BC 2
15.5 %
d: BI 2, BC 2
30.9 %
WR-to-(auto)PRE: B + CWL + AL + WR = 4 + 8 + 0 + 12 = 24 cc
PRE-to-ACT: RP = 11 cc
e: BI 1, BC 4
26.7 %
WR-to-(auto)PRE: B + CWL + AL + WR = 4 + 8 + 0 + 12 = 24 cc
PRE-to-ACT: RP = 11 cc
f: BI 4, BC 1
33.5 %
RRD = 6 cc
RRD = 6 cc
RRD = 6 cc
g: BI 4, BC 2
61.6 %
Auto-precharge in parallel with another command:
h: BI 8, BC 1
38.5 %
Four Activate Window (FAW) = 40 cc
Repeat the same sequence for bank 4-7
40 cc

Fig. 3.3 DDR3-1600 example schedules

resent activity on the data bus, caused by a read (R) or write (W) command. Each of those commands generates a burst transfer, lasting multiple cycles. The size of the burst is determined by the burst length setting (BL), which is typically 8 for DDR3, resulting in 4 clock cycles of data-bus activity. In reality, there is a certain delay between the read or write command and the corresponding data burst, but in Fig. 3.3 the shading starts immediately at the command for simplicity.

Activate commands are annotated with the letter A, while cycles where an auto-precharge is executed have the letters aP written in them. We assume precharge commands are always implemented using auto-precharge flags that are attached to the final read or write command to a bank, which has as advantages that they do not need to be scheduled explicitly, and do not take up space on the command bus (we draw them as regular commands when they happen in parallel with NOPs). Figure 3.3g illustrates this, with the auto-precharge to bank 0, which happens in parallel with the write to bank 3. If an explicit precharge was used, then either the write or the precharge would have to happen one cycle later, increasing the schedule length and reducing performance.

Double-headed arrows are annotated with the timing constraints that determined the shape of the schedules. Each schedule is extended with NOPs until it is repeatable after itself. This is the behavior that a memory controller would exhibit in the worst case if it used the shown schedule to service requests, since in the worst case different rows in the same banks that were used earlier are accessed by each request. Auto-precharges that are scheduled relatively late can be pipelined with commands designated for following requests, i.e., they might be executed during the next repeat of the same schedule. In the figure, this is indicated by the overline on their $\overline{aP}$ annotation.

The fact that the figure has to be tilted awkwardly to fit on the page while most of the blocks are empty illustrates an important point: schedules are relatively long compared to the amount of useful commands that are executed in them. In a naive implementation, a memory controller could chop all incoming requests into atoms the size of *one burst*, and issue one ACT, one RD/WR, and one PRE command to service each of them. Figure 3.3a shows what such a command schedule would look like. Due to overhead of activating and precharging, the schedule is significantly longer (46 cycles) than the actual data transfer time (4 cycles assuming a burst length of 8), a difference of more than an order of magnitude. The worst-case bandwidth is thus a lot smaller than the peak bandwidth obtained by only considering the data rate, and this efficiency gap grows as the memory clock frequency increases, as later shown in Chap. 5.

3.2.1 Pattern Generation with Variable Bank Interleaving

The distribution of the bursts in the pattern across banks and the associated worst-case performance is determined by how the memory controller groups bursts (Sect. 2.2.1). This section introduces a command scheduling algorithm that changes its behavior based on a selected burst-grouping configuration. Instead of interleaving each atom over all banks in the memory [7–10], or assuming that each atom maps only to a single burst [11, 12], *we treat the number of banks involved in executing an atom as a free parameter*. With this extra degree of freedom, we use the parameters defined earlier in Sect. 2.2.1 to denote

1. *BI* as the number of banks that are accessed by an atom, and
2. *BC* as the number of bursts per bank.

The number of bursts per atom is then equal to BI · BC. These parameters can be used to generate a range of possible *pattern configurations* characterized by a (BI, BC) combination. BI can be equal to or smaller than the number of banks in the memory. If BI is smaller than the number of available banks, then there are multiple ranges of BI banks that could be accessed by a pattern, offset by BI banks from each other. For example, if BI = 2, then the accessed banks could be { 0, 1 } or { 2, 3 }, but not { 1, 2 } or { 0, 5 }. We require the ranges to be mutually exclusive, such that the worst-case sequence of banks is generated by successive atoms accessing a different row in the same range of BI banks. Figure 3.4 illustrates this for a DDR3-1066. If we were to abandon that requirement, then an atom could potentially start its pattern with the same bank the previous atom ended its access with, effectively eliminating most of the (guaranteed) bank parallelism, and requiring longer patterns to satisfy all constraints, reducing performance. Therefore, *the addresses of atoms that enter the back-end are required to be aligned at atom-sized boundaries.* This is enforced by the atomizer in the controller's front-end. BI and BC effectively define the low-level memory map for bursts (see Sect. 3.2.5).

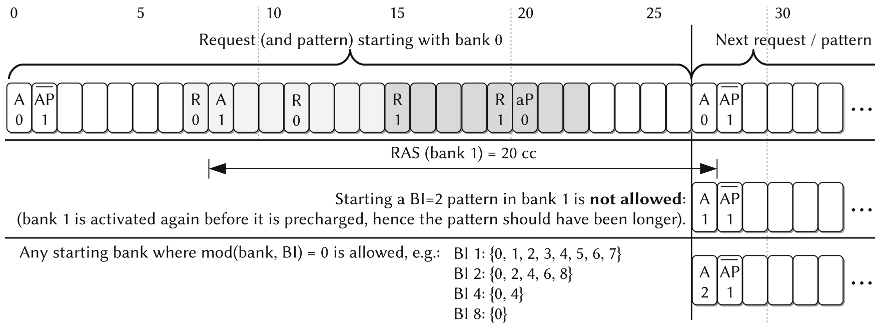

0
5
10
15
20
25
30
Request (and pattern) starting with bank 0
Next request / pattern
A 0
AP 1
R 0
A 1
R 0
R 1
R 1
aP 0
A 0
AP 1
RAS (bank 1) = 20 cc
Starting a BI=2 pattern in bank 1 is **not allowed**:
(bank 1 is activated again before it is precharged, hence the pattern should have been longer).
A 1
AP 1
Any starting bank where mod(bank, BI) = 0 is allowed, e.g.:
BI 1: {0, 1, 2, 3, 4, 5, 6, 7}
BI 2: {0, 2, 4, 6, 8}
BI 4: {0, 4}
BI 8: {0}
A 2
AP 1
...

Fig. 3.4 A (BI2, BC2) read pattern for a DDR3-1066

Figure 3.3 illustrates how the schedules change for different (BI, BC) combinations for the DDR3-1600 example (Note that each line is annotated with a (BI, BC) pair). Figure 3.3b demonstrates the benefit of bank interleaving. Two bursts are interleaved, i.e., BI = 2. The ACT-to-RD/WR delay (RCD) of bank 1 is (partially) hidden by the data access to bank 0.

Increasing BC enables hiding the ACT-to-ACT constraint between banks (RRD_X). This is relevant for all memory devices for which the maximum activate command rate is lower than the read/write command rate (RRD_X $>$ B), a relatively common property (Appendix B). The DDR3-1600 memory from Fig. 3.3b has RRD $=$ 6, and this constraint causes the two-cycle pause in the data transfer between the burst to bank 0 and bank 1. Figure 3.3c, d shows how this issue is resolved when BC is increased.

For memories that have more than 4 banks, configurations with BI $\leq$ 4 are of particular interest, since they deal better with the *Four Activate Window* (*FAW*) constraint than those that interleave over more than 4 banks. With at most 4 activate commands within a pattern, the FAW can only play a role if multiple consecutive patterns are considered. This allows NOPs inserted at pattern edges to satisfy the FAW constraint to overlap with NOPs that resolve other constraints, like RC (ACT-to-ACT) and RP (PRE-to-ACT) for example, and hence these configurations are more efficient. Figure 3.3g, h illustrates this: the schedules contain the same number of bursts, but Fig. 3.3g avoids the FAW penalty, increasing the efficiency from 38.5 to 61.6 %. Efficiency generally increases with the number of bursts per atom, because the constant activate/precharge overhead is amortized over more and more data. In practice, the atom size is limited by the size of the requests the memory clients generate, since there is no point in fetching data at high efficiency when it has to be discarded later because the client is not interested in it. Chapter 5 quantifies the effects of BI and BC on the worst-case performance for a set of different memories in more detail. Here, we now focus on how to automatically generate patterns, such as those shown in Fig. 3.3, based on a particular BI and BC.

Pattern-Generation Heuristic

The functions in Algorithms 1 and 2 build up the pattern in the set **P**, in which each element is a 3-tuple representing the type, bank, and clock cycle (cc) of a command. Record-/struct-like semantics are used to access the elements in the tuple, i.e., $x.cc$ accesses the clock cycle element of the tuple. The max() function executed on a set returns the largest element within that set. Indentation delimits code blocks. We use the *bank scheduling heuristic* (BS) described in [13] as a starting point for the order and placement of the commands, because it has been shown to perform well for DDR2/3 memories. In this heuristic, read and write patterns are created independently. Within these patterns, read or write commands are scheduled as soon as possible, accessing banks in ascending order. Activate commands are scheduled as late as possible, but just in time not to delay the read or write commands. Typically, this is RCD cycles before the first read or write to the associated bank, or earlier if this cycle is already taken by another command. Patterns start with bank activation, and the final access to a bank has an auto-precharge flag. This heuristic is extended to include the new BI parameter, and we refer to bank scheduling with variable bank interleaving as BS BI. Algorithm 2 shows the most relevant read and write pattern-generation functions; a complete executable version can be found in [14]. For BS BI, the input argument USEBSPBGI should be set to *false*. (The same algorithm is reused later in Sect. 3.2.2 with this parameter set to *true*.

Algorithm 2 uses two small helper functions, shown in Algorithm 1. The EARLIEST function returns the earliest cycle at which a command cmd_b may be scheduled, given the location of the commands in the (partial) pattern **P**. It uses the $d()$ function (Eq. 3.1), which symbolizes the lookup in Tables 3.1 and 3.2. If the **P** parameter is an empty set, then 0 is returned (line 6). Figure 3.5 shows the function output with an example. The MINPATTERNDISTANCE function finds the smallest number of cycles (NOPs) that must be inserted between two patterns to satisfy all constraints spanning across them. An example for this function is shown in Fig. 3.6.

Now we move on to Algorithm 2. The nested loops (lines 4–8) generate 1 activate and BC read or write commands per bank. The ADDACTANDRW function schedules an activate command using ADDACT before the first burst to a bank (lines 17–18). Additionally, it schedules the read/write commands as soon as possible (lines 16, 19).

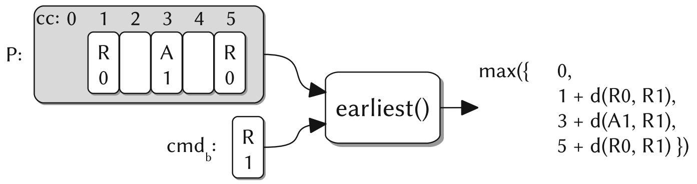


Fig. 3.5 Example execution of the EARLIEST function

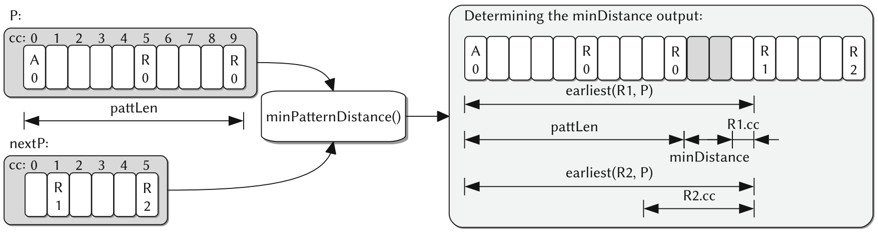


Fig. 3.6 Example execution of the MINPATTERNDISTANCE function (The commands in the example are merely there to show the functionality, but do not resemble real patterns)

ADDACT first finds a range of possible locations for an ACT command. A lower bound (lb) for the location is based on the ACT-to-ACT and FAW constraints (lines 22–23). The definition of REMAININGFAWCYCLESAT can be found in Algorithm 6 in Appendix C. An upper bound (ub) is set by the minimum distance between the planned location of the RD/WR command (rwCc) and the ACT command (line 25). Set **S** contains the cycles within this range that are not occupied by other commands (line 26). If this set is not empty, then the largest option is chosen, scheduling the ACT as late as possible (lines 27–28). Otherwise, the RD/WR command is postponed by a cycle, and by extension the upper bound for the ACT shifts forward, until a suitable location is found (lines 24, 30). This guarantees the algorithm always finds a schedule.

What remains is to determine the location of the precharge commands (lines 9–12), which are stored in a separate copy of the pattern ($\mathbf{P}'$) since they are implemented using auto-precharge flags and hence are not explicitly scheduled. Their location is still relevant, because the precharges can constrain commands that follow them, within the next incarnation of the pattern. Additionally, they are required to generate the auxiliary patterns.

Given the commands in $\mathbf{P}'$, MAKEREPEATABLE finds the minimum length the pattern should have to be repeatable after itself without violating regular constraints (line 33) and the FAW constraint (lines 34–35) spanning across pattern incarnations. The definition of FAWSATISFIEDACROSS can be found in Algorithm 6 in Appendix C. Each time the length is increased, one NOP is implicitly added to the end of the pattern. Finally, the scheduled commands and the length of the pattern are returned and the algorithm ends.

Algorithm 1 Helper functions

```
 1: function EARLIEST(cmd_b, P)
 2:     // d() is a lookup in Table 3.1-3.2. It returns -inf if the command combination is
 3:     // not mentioned.
 4:     pos := 0
 5:     for all cmd ∈ P do
 6:         pos := max(pos, cmd.cc + d(cmd, cmd_b))
 7:     return pos
 8: function MINPATTERNDISTANCE(pattLen, nextP, P)
 9:     // Determine the minimum distance between P and nextP,
10:     // given the length of P is pattLen.
11:     minDistance := 0
12:     for all cmd ∈ nextP do
13:         minDistance := max(minDistance, EARLIEST(cmd, P) - cmd.cc - pattLen)
14:     return minDistance
```

Algorithm 2 Bank scheduling heuristic for BS BI and BS PBGI

```
function PATTERNGEN(BI, BC, rdOrWr, useBsPbgi)
    BGi := if BI > 1 and useBsPbgi == true then 2 else 1
    P := { } // The pattern
    for all bankPair ∈ { 0...BI/BGi − 1 } do
        for all burst ∈ { 0...BC − 1 } do
            for all offset ∈ { 0...BGi − 1 } do
                bnk := bankPair × BGi + offset
                P := ADDACTANDRW(bnk, rdOrWr, burst, P)
    P' := P // A copy with explicit precharges.
    for all bnk ∈ {0...BI − 1} do
        preCc := EARLIEST((PRE, bnk, 0), P)
        P' := P' ∪ { (type: PRE, bank: bnk, cc: preCc) }
    return MAKEREPEATABLE(P, P')
function ADDACTANDRW(bnk, rdOrWr, burst, P)
    rw := (type: rdOrWr, bank: bnk, cc: 0)
    rwCc := EARLIEST(rw, P)
    if burst == 0 then
        P, rwCc := ADDACT(rw, rwCc, P)
    return P ∪ { (type: rdOrWr, bank: bnk, cc: rwCc) }
function ADDACT(rw, rwCc, P)
    act := (type: ACT, bank: rw.bank, cc: 0)
    lb := EARLIEST(act, P)
    lb := lb + REMAININGFAWCYCLESAT(lb, P)
    while true do
        ub := rwCc − d(act, rw)
        S := {$i$ ∈ {lb...ub − 1} | cmd.cc $\neq i$ ∀ cmd ∈ P}
        if S $\neq \varnothing$ then
            P := P ∪ { (type: ACT, bank: rw.bank, cc: max(S)) }
            return P, rwCc
        rwCc := rwCc + 1
function MAKEREPEATABLE(P, P')
    len := max({cmd.cc ∀ cmd ∈ P}) + 1
    len := len + MINPATTERNDISTANCE(len, P', P)
    while not FAWSATISFIEDACROSS(len, P) do
        len := len + 1
    return P, len
```

3.2.2 BS PBGI Heuristic for DDR4 Pattern Generation

The heuristic that was presented in the previous section works well for most contemporary SDRAM types, as will be shown in Sect. 3.4. However, due to the introduction of bank groups in DDR4, there is room for improvement in the heuristic for this SDRAM type, such that it works around the penalties related to accessing the same bank group twice in a row (see Sect. 2.1.3.1 and [4]). To generate more efficient DDR4 patterns and avoid hitting the CCD_L constraints between bursts, read or write commands should be interleaved across bank groups. To this end, we propose a *pair-*

wise bank-group interleaving heuristic, as demonstrated in Fig. 3.7. Two banks from different bank groups are paired together. In contrast to regular bank scheduling, which finishes all BC bursts to a bank before switching to the next bank, the read or write commands of such a pair are interleaved per burst. The remaining rules of the heuristic are the same as described in Sect. 3.2.1. We refer to bank scheduling with pairwise bank-group interleaving as BS PBGI. Setting USEBSPBGI to *true* in Algorithm 2 generates the proposed interleaving. The algorithm assumes that consecutive bank ids map to different bank groups (and wrap around once the bank groups run out). This can be implemented by wiring the least significant bits of the bank address (described in Sect. 3.2.5) to the bank-group bits on the DDR4 interface.

Algorithm 3 Creating auxiliary patterns

```
1: function RTWPATTERN(rdP, wrP, rdPLen)
2:     return MINPATTERNDISTANCE(rdPLen, rdP, wrP)
3: function WTRPATTERN(rdP, wrP, wrPLen)
4:     return MINPATTERNDISTANCE(wrPLen, wrP, rdP)
5: function REFPATTERN(rdP, wrP, rdPLen, wrPLen)
6:     // Create the refresh pattern.
7:     refP := { (type: REF, bank: 0, cc: 0) }
8:     prefix := max(MINPATTERNDISTANCE(rdPLen, rdP, refP),
9:                   MINPATTERNDISTANCE(wrPLen, wrP, refP))
10:    refP := { (type: REF, bank: 0, cc: prefix) }
11:    postfix := max(MINPATTERNDISTANCE(prefix + 1, refP, rdP),
12:                   MINPATTERNDISTANCE(prefix + 1, refP, wrP))
13:    refPLen := prefix + 1 + postfix
14:    return refP, refPLen
```

In cases where $\text{BI} \geq 2, \text{BC} \geq 2$, this heuristic behaves differently from BS BI. Pairwise interleaving reduces the time between successive read and write commands from CCD_L to CCD_S, which typically saves one or two cycles per burst pair for the currently released DDR4 devices. In total, this could save $(\text{CCD\_L} - \text{CCD\_S}) \cdot \text{BI} \cdot (\text{BC} - 1)$ cycles per pattern, assuming there are no other constraints that force a separation larger than CCD_S between (some of) the bursts.

The advantage of interleaving only two instead of, for example, four bank groups is that the last access to the first bank pair happens relatively early in the pattern, as shown in Fig. 3.7. As a result, these banks can be precharged (partially) while accesses to other banks pairs are executed, eventually allowing them to be activated earlier. Interleaving more than two banks reduces the overlap, and could hence lead to patterns that require more NOPs at the end of the pattern to satisfy the constraints required to repeat the pattern, without any benefits.

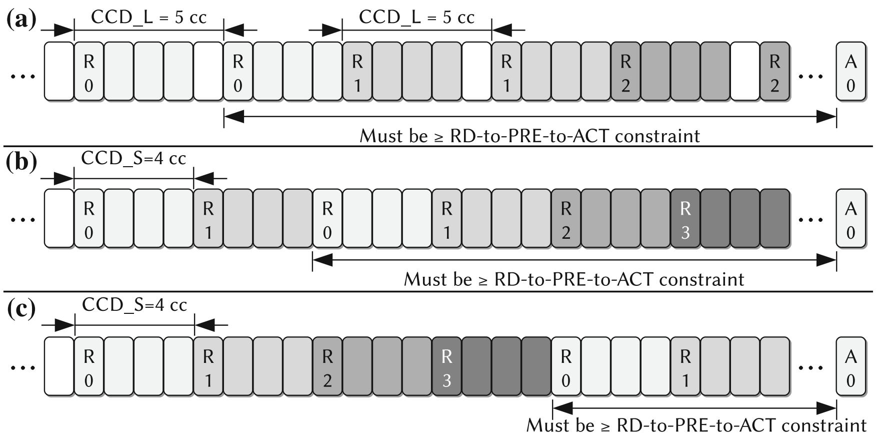


Fig. 3.7 (Partial) DDR4-1866 read pattern. Odd and even banks are in a different bank group. Schedule *a* does not use (BS PBGI), while *b* does. *c* shows how the distance to the next activate in a following pattern reduces as more bank groups are interleaved, resulting in longer (less efficient) patterns

3.2.3 Auxiliary Patterns

To finish a pattern set, it needs auxiliary patterns, i.e., read-to-write and write-to-read switching patterns, and a refresh pattern. These patterns are created based on the read and write patterns (**rdP**, **wrP**), and their respective lengths (rdPLen, wrPlen), as shown in Algorithm 3. The first two functions in this algorithm determine the length of the switching patterns, which are inserted between access patterns of the opposite type to resolve constraints between them. Since switching patterns consists solely of NOPs, it is sufficient to return only the length of these patterns.

The refresh pattern, generated by the third function, consists of a single refresh command, optionally surrounded by NOPs. First, we determine the number of NOPs that has to precede it, and store that in the PREFIX variable. The REF command is scheduled after this prefix. Finally, the minimum number of NOPs that has to separate the refresh command from the start of the next read or write pattern, whichever is larger, is appended at the end of the refresh pattern.

3.2.4 ILP-Based Pattern Generation

Sections 3.2.1 and 3.2.2 describe two heuristics, BS BI and BS PBGI, which generate close-page read and write patterns. The lengths of these patterns can be used as a measure for their quality, since they determine the worst-case memory performance, as later shown in Sect. 3.5. The commands in a pattern are chosen and fixed once a (BI, BC) combination is selected, so *we can define a pattern as optimal in terms of length*

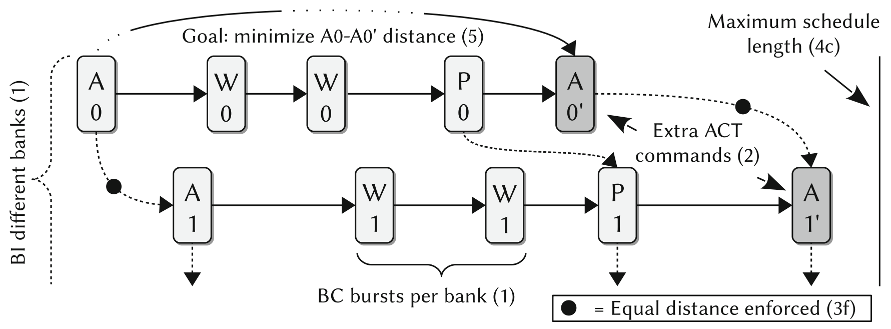


Fig. 3.8 Example of the ILP precedence constraints. An edge between a set of commands means that the source command has to be scheduled before the destination command. *Numbers* in *round brackets* refer to the associated rule in the ILP description

if there is no other permutation of this set of commands satisfying pattern scheduling rules and timing constraints resulting in a shorter pattern. The limitation of this definition is that it considers the read and write patterns separately. When combined, it is possible (although not likely) that they might not make an optimal pattern set (in terms of worst-case bandwidth for example), since the auxiliary patterns are not taken into account during their creation. We can, however, always use them as a lower bound on the length of the individual read and write patterns, and contrast that with the output of our heuristics.

This section explains how (length) optimal patterns can be generated using a parameterized ILP formulation of the command scheduling problem. Based on a (BI, BC) combination and an implementation of Eq. (3.1), we create an ILP problem that, when solved, finds the optimal pattern size and the location of the commands within the pattern. Any memory controller that uses a close-page policy and relies on memory patterns in analysis or implementation, like [7–10, 15], can use this formulation to improve the schedules it uses, or to extend its scope to different memory devices or generations.

We later use the ILP formulation as a means to evaluate the BS BI and BS PBGI heuristics. The translation to a formal problem definition is available in Appendix A, but here we only describe the formulation in natural language: Fig. 3.8 illustrates a subset of the properties of the formulation.

1 Create a set of variables representing the locations of the commands in the pattern that should be generated as a function of the selected (BI, BC) combination: there are BI different ACT commands, one for each bank, and BI times BC different RD/WR commands, and BI PRE commands (as auto-precharge flags).
2 Add variables representing the location of an *extra activate command* for each bank in this set. These activate commands represent the start of a second instance of the pattern, which should be schedulable immediately after the first instance, because read/write patterns should be repeatable after themselves (Sect. 2.2).

3 Given this set of memory commands, assign a *single* location in the schedule to each command such that

   (a) An ACT to bank 0 is scheduled in cycle 0.
   (b) No two commands are scheduled in the same cycle. Precharges are exempted from this rule, since they are executed using auto-precharge flags and do not require a slot in the schedule.
   (c) The relative delays between any pair of commands is at least as large as prescribed by Eq. (3.1), and there are at most four ACTs in each FAW window.
   (d) The commands for each bank are executed in the proper order, i.e., start with an activate, followed by BC read or write commands, followed by a precharge. This formulation allows different banks to be used in parallel, i.e., one can be activating while another is used for reading or writing. The extra activate commands added in Step 2 should happen after the precharge to the associated bank.
   (e) Commands for the second instance of the pattern must be scheduled after the extra activate to bank 0, and commands for the first instance must be scheduled before the extra activate to bank 0. This activate command itself and all precharge commands are exempt from this rule. Precharges may be (automatically) pipelined across patterns, because auto-precharge flags are used and they are hence scheduled automatically.
   (f) The first and second instances of the pattern are the same. A set of constraints enforces that the distance between the extra activate command to a bank and the extra activate to bank 0 is equal to the distance between the first activate command to that bank and cycle 0. As a result, the activates appear in the same relative position in the second pattern instance. The positions of the read or write commands in the second instance follow from the positions of the activates, and do not need to be scheduled explicitly.

4. To limit the search space and eliminate equivalent symmetric solutions, we add the following constraints (see Fig. 3.8):

   (a) The order of the read or write commands to the same bank is fixed, because we cannot (and do not want to) distinguish between different bursts to the same bank within a pattern.
   (b) Banks are activated and precharged in ascending order. For DDR4, we again use the assumption that consecutive bank ids map to different bank groups (and wrap around once the bank groups run out).
   (c) An upper bound on the optimal length of the pattern in cycles can be found based on the BS BI or BS PBGI (DDR4) heuristics. Both provide a valid bound, so we use whichever is the smallest to limit the solution space as much as possible, and hence reduce the computation time of the solver.
   (d) A lower bound for the pattern length is the size of a schedule where the commands for bank 0 are scheduled as soon as possible. A lower bound for the location of the extra activate commands of other banks is derived

from this bound, since they must at least be scheduled one ACT-to-ACT constraint away from bank 0's activate.

5. The optimization goal is to minimize the pattern length. Therefore, we minimize the location of the second activate to bank 0, which signifies the start of the next incarnation of the pattern.

The ILP formulation might create shorter patterns than BS BI and BS PBGI, because it does not restrict the relative ordering of bursts across banks nor the placement of bursts within the pattern, and it has no preferred location for activate commands (i.e., it could postpone them compared to the heuristics). Section 3.4 evaluates how close to optimal the heuristic results are.

3.2.5 Memory Map Implications

The order of the data bursts within a memory pattern is fixed, meaning they are mapped to consecutive logical memory addresses by definition. The pattern configuration (BI, BC) thus has a direct influence on the decoding of the (least significant) portion of the logical address into a physical address, as it partially determines which bits should be selected for the bank, row and column address. *A (BI, BC) combination is hence as much a memory map configuration as it is a pattern or scheduler configuration.* This is illustrated in Fig. 3.9, which shows how the lower $log_2(\text{BI} \cdot \text{BC} \cdot \text{BL} \cdot \text{IW}/8)$ bits of the logical address (top row) are mapped to the column and bank address *Least significant Bits LSB*, respectively. The connection to the memory map practically limits the possible values for BI and BC to powers of two.

The address generator that was shown in the previous chapter (Fig. 2.9) can extract 4 chunks of consecutive bits from an incoming logical address. Two of these chunks form the column address, concatenating the *remaining column bits* and *BC* chunks, while forcing the lower *BL* bits to zero, since sub-burst addressing is not supported. The address generator can extract a portion of its bank address bits from the command bank pairs that are stored in the pattern memory, as discussed in Sect. 2.3.2. They are

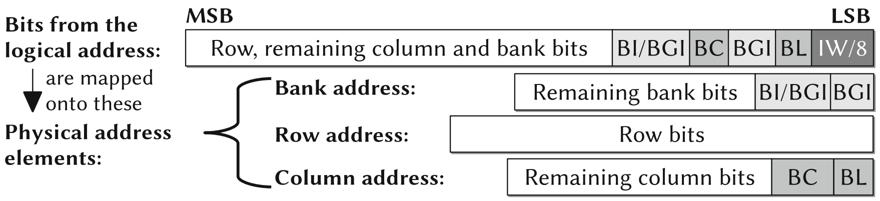


Fig. 3.9 Memory map from logical to physical address. BGI refers to the degree of bank-group interleaving, which we limited to 2 in Sect. 3.2.2. Bits from the logical address map to the similarly marked locations in the physical address. For example, $log_2(\text{BI/BGI})$ bits from the corresponding position in the logical address are used in the similarly marked position in the bank address

combined with the *remaining bank bits* chunk to complete the bank address. This feature allows us to transparently use the more complex bank interleaving orders that the ILP formulation might produce. The final chunk of bits simply represents the row address. Note that if a parameter like for example BGI has the value 1, then its corresponding chunk disappears from the address ($log_2(1) = 0$).

The remaining (white) portion of the logical address can be mapped freely to different physical address elements, such that for example (1) separate memory regions are generated for different clients (spatial partitioning), and (2) locality is promoted as much as possible in case a conservative open-page policy (Chap. 6) or cache [16] is used. This could imply different things for different types of clients. If incremental request address sequences are to be expected then it makes sense to map consecutive addresses to the same bank and row as much as possible to maximize locality. Similarly, for caches, it can be beneficial to map addresses that map to the same cache line to the same row and bank as much as possible.

Figure 3.10 shows three data layout examples in an extremely small 4-bank memory, resulting from different (BI, BC) and 'useBsPbgi' settings (written as BGI in the figure). We map the bits that are not bound to the pattern configuration (the white part in Fig. 3.9) such that for consecutive logical addresses the bank changes first, followed by the column, and finally the row.

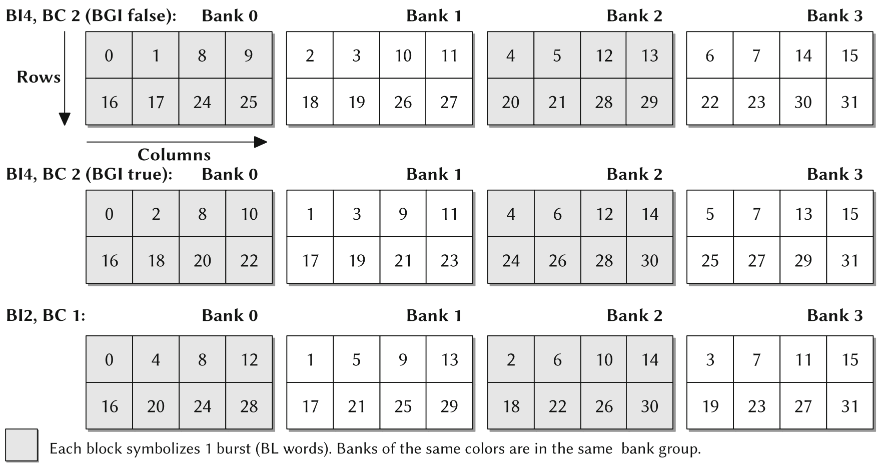


Fig. 3.10 Three memory map examples, showing where the bursts of requests to consecutive logical addresses (separated by the access granularity) are written. The third configuration, using (2, 1), behaves the same regardless of the BGI setting

3.3 Composable Pattern Conversion

The previous section discussed different means to create predictable memory patterns. These can be used within the memory controller template to offer worst-case bandwidth and response time guarantees to the memory clients. However, there is still interference between memory clients, since variation exists in the amount of time it takes to process atoms based on their types. For example, read and write patterns can be of different lengths, and switching patterns are inserted between access patterns of different types, both causing differences in the execution time of the sequence of commands belonging to an atom.

Delay blocks can be added to create the cycle-level isolation required for composable behavior, although they have some disadvantages, besides the obvious additional hardware cost (Sect. 2.6). Delay blocks turn the actual-case performance into the analytically determined worst-case performance, which additionally may be pessimistic if computed bounds are not tight. Additionally, a delay block cannot just isolate and eliminate variation caused by other clients, but instead removes *all* variation. It needs to assume that each busy period starts one cycle after an arbitration decision has been made, such that the client misses its first chance on a scheduling slot. As a result, the benefit generated by better-than-worst-case request arrival times of the client's own requests with respect to the arbitration cycle of the resource is absorbed by the delay block, harming the client. This section introduces a new mechanism to extend the predictable pattern-based memory controller to create a composable SDRAM resource in which requests from separate memory clients are temporally isolated, while avoiding the previously mentioned disadvantages.

The key idea is to *share the SDRAM through non-work-conserving TDM arbitration, and to make the start of a client's time slots independent from other clients*. In this context, 'non-work-conserving' means that slots that are not claimed by their owner cannot be used by other clients. To ensure that a slot always starts at the same time, all slot lengths have to be equal regardless of the atom type or the presence or absence of an eligible atom. Also, the state of the memory must return to neutral after each atom, such that following atoms are not constrained by previous atoms from other clients. This implies that a close-page policy must be used. The influence of the atom type must also be eliminated, meaning that the timing constraints that allow *both* read and write patterns must be satisfied at the end of the slot. To meet these requirements, the predictable memory patterns are converted to *composable patterns*. Section 3.3.1 discusses that process, after which performance bounds for these patterns are derived in Sect. 3.3.2.

3.3.1 Composable Memory Pattern Generation

Composable memory patterns are constructed at design time in a similar manner as predictable patterns. The goal is to create composable read and write patterns that

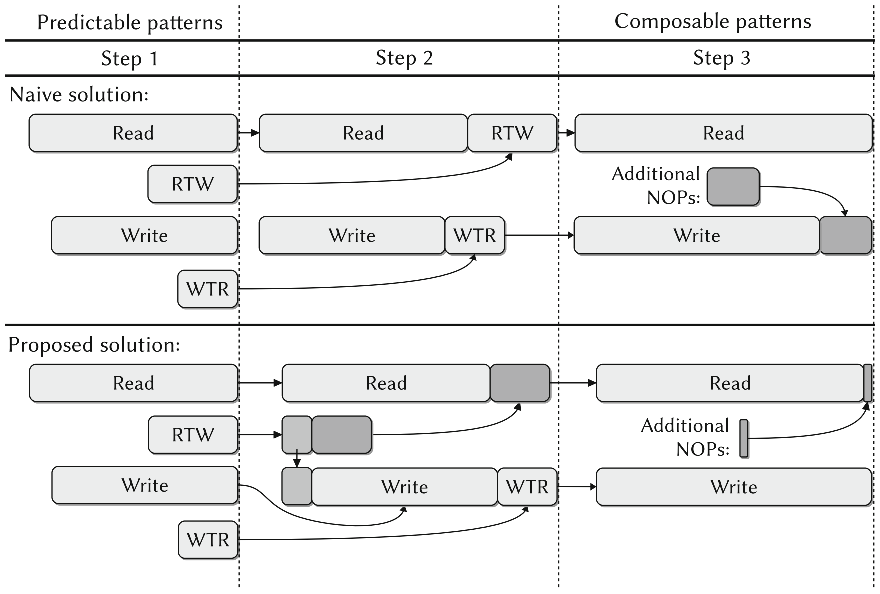


Fig. 3.11 Composable pattern-generation example. The naive solution simply concatenates the switching patterns to the access patterns and then adds NOPs to equalize the length, while the proposed solution uses the switching patterns to balance the lengths as much as possible before adding more NOPs, leading to shorter patterns

can be scheduled arbitrarily without violating timing constraints, and are equal in length. Their length determines the length of one TDM slot.

The composable patterns are generated in three steps. The first step generates a predictable pattern. The last two steps make the pattern set composable. Figure 3.11 shows the relation between a predictable pattern set and its composable counterpart. We proceed by discussing each of these steps in more detail:

1. Read and write patterns are generated based on one of the methods in Sect. 3.2.1. NOPs have already been added at the end of these patterns, such that they can be repeated after themselves without violating SDRAM timing constraints. This determines the minimum length of the predictable access patterns. Based on this, the lengths of the switching patterns are determined.
2. Composable pattern sets cannot contain switching patterns, since they introduce timing dependencies on the previous atom type. Instead of having separate switching patterns, the required NOPs are distributed amongst the read and write patterns. NOPs resolving read-to-write (RTW) constraints can be added either at the end of the read pattern or the beginning of the write pattern, while NOPs resolving write-to-read (WTR) constraints can be added either at the end of the write pattern or the beginning of the read pattern. A naive approach, which simply concatenates the full switching patterns to the corresponding read or write pattern, incurs unnecessary overhead in Step 3. *Therefore, we aim to equalize the lengths*

of the access patterns with all available NOPs from both switching patterns. They are distributed to balance the pattern lengths as much as possible, reducing the conversion overhead in terms of efficiency, as shown in Fig. 3.11.
3. Finally, any length difference that still remains between the read and write pattern has to be compensated for by adding NOPs at the end of the shortest pattern.

TDM slots that are not occupied by an atom are filled with idle patterns consisting of only NOPs. The idle pattern length is made equal to the composable read or write pattern length, which guarantees that all slots always take the same number of cycles. The refresh timer triggers the execution of refresh patterns *at the end* of a regular or idle slot after every REFI cycles. The actual insertion time is not influenced by the running clients since all slots are equally long, meaning refresh is also composable.

"The start of a time slot" is an abstract concept, which in the hardware implementation of the memory controller is translated into valid-accept handshakes on the DTL interface between the atom queue, which is private to a client, and the resource bus. The resource bus experiences back-pressure from the back-end, i.e., the back-end does not accept a new atom while it is still working on a previous one. This back-pressure is forwarded to the port-specific hardware in a pipelined fashion. The hardware implementation is only composable if the timings of the valid-accept handshakes on the *command, read data, and write data channels* of the bus-to-back-end DTL interface (see Fig. 2.11) do not leak information from one client to the other. This is assured in different ways for each channel:

- The command channel has a guaranteed acceptance rate, because patterns have a known fixed length and hence a new command can be inserted at regular intervals. At the start of a slot, that is, after a scheduling decision by the arbiter, the command channel is guaranteed to only exert back-pressure if a refresh was inserted. This delays new scheduling decisions independently of the clients, creating room for the refresh in the schedule, and all future slots shift forward in time appropriately.
- The time between the acceptance of an atom on the command channel and the arrival of data on the read channel is fixed, since the SDRAM commands in a pattern have a fixed delay with respect to the valid-accept handshake on the command channel, and read data has a fixed delay with respect to the RD commands in a pattern. The atom buffer is guaranteed to have space for the data (since it otherwise would not have presented the atom to the resource bus), and hence the valid-accept handshake on this channel is not client-dependent.
- The rate at which data enters the back-end is the same as the rate at which it exits the back-end, although the time at which data leaves is shifted with respect to the start of the pattern. This happens because the first few commands of a pattern are typically not WR commands, and there is an additional required delay (write latency) between the WR command and the associated data on the SDRAM data bus. The write latency could theoretically span across pattern/slot boundaries for certain memories and (BI, BC) configurations, and in the mean time the data is temporarily buffered in the back-end. The write-data buffer accepts all data as soon as possible at the start of a pattern if it is empty. Since the presence or absence of buffered write data connected to a slot is dependent on the type of atom that

was scheduled in it, we must make sure that this can never cause back-pressure for another slot, since that implies there is cross-client interference. To guarantee this, the size of the write-data buffer in the back-end is increased such that it never causes back-pressure at slot edges. In practice, this means it has a capacity of 2 atoms worth of data, assuming the maximum offset between the command and write channel is less than 2 slots, which holds for all SDRAMs we are aware of, including those considered in our experiments (Appendix B).

The impact of the conversion from predictable to composable patterns on the memory efficiency is shown in the next section.

3.3.2 Impact on Memory Efficiency

The worst-case analysis for predictable patterns is based on the notion of memory efficiency, as shown in Sect. 2.4.2.2. To evaluate the performance of composable patterns, the efficiency loss with respect to the corresponding predictable pattern set has to be determined. This section derives an expression for the efficiency loss. In Sect. 3.4.3.1, we apply it to quantify the efficiency loss for a set of relevant memories.

The lengths of the predictable read, write, write-to-read, and read-to-write patterns are denoted by t_r^p, t_w^p, t_{wtr}^p, and t_{rtw}^p, while the composable access pattern lengths are denoted by t_r^c and t_w^c, respectively. We need to distinguish three different cases, depending on the length of the predictable patterns. For this purpose, we use the *dominance classes* from [13]:

1. If the read pattern is longer than the write pattern plus both switching patterns, then the worst-case request sequence consists of only read requests, and the pattern set is *read dominant*. The composable access patterns are as long as the predictable read pattern, $t_r^c = t_w^c = t_r^p$.
2. If the write pattern is longer than the read pattern plus both switching patterns, then the worst-case request sequence consists of only write requests, and the pattern set is *write dominant*. The composable access patterns are as long as the predictable write pattern, $t_r^c = t_w^c = t_w^p$.
3. Pattern sets that do not fit in class 1 or 2 show worst-case behavior if read and write requests are alternated. These pattern sets are *mix dominant*. The pattern set in Fig. 3.11 is an example of this class. In this case,

$$t_r^c = t_w^c = \left\lceil \frac{t_r^p + t_w^p + t_{wtr}^p + t_{rtw}^p}{2} \right\rceil \tag{3.2}$$

In the worst case, only the dominant pattern of a read or write dominant pattern set is used. Executing this pattern is the most time-consuming way to transfer one atom, so it determines the worst-case efficiency. Composable patterns based on read or write dominant predictable patterns have composable read and write patterns lengths that are equal to the dominant pattern length. This means their worst-case efficiency is unaffected by the conversion.

If the composable pattern set is based on a mix dominant pattern set, then the worst-case efficiency is only affected if the two switching patterns are smaller than the length difference between the read and write pattern, and NOPs had to be added in Step 3 of the conversion to balance the patterns. At most one NOP is required for this by definition, or else the pattern set would not be mix dominant. If $t_r^p + t_w^p + t_{wtr}^p + t_{rtw}^p$ is odd, then the sum of the composable pattern lengths is equal to the sum of the predictable pattern lengths plus 1, as a result of rounding up in Eq. (3.2). In all other cases, the composable pattern set efficiency is equal to the predictable pattern set efficiency. The conversion efficiency (e_{pc}) is thus as follows:

$$e_{pc} = \begin{cases} \frac{t_r^p + t_w^p + t_{wtr}^p + t_{rtw}^p}{1 + t_r^p + t_w^p + t_{wtr}^p + t_{rtw}^p} & \text{if mix dominant and } t_r^p + t_w^p + t_{wtr}^p + t_{rtw}^p \text{ is odd,} \\ 1 & \text{otherwise.} \end{cases} \tag{3.3}$$

3.4 Evaluation

This chapter presented two novel pattern-generation heuristics, an ILP formulation of the pattern-generation problem, and a conversion method to create composable patterns out of predictable patterns. This section evaluates the efficiency of these ideas. First, we introduce the set of test memories that are used in the remainder of this book in Sect. 3.4.1. For these memories, Sect. 3.4.2 compares the schedule lengths generated by the two heuristics to those generated by the ILP instance, while Sect. 3.4.3 evaluates the technique for converting predictable patterns into composable patterns.

3.4.1 Test Memories

We consider two devices per SDRAM type. Each device is part of a speed bin defined by the associated JEDEC standard for that SDRAM type. We select speed bins based on the commercial availability of the device and data sheets at the time of writing, the range of clock frequencies they cover (we select a device from the slowest and fastest bin if available), and comparability with speed bins of other SDRAM types (select common speeds and data bus widths as much as possible). Table 3.3 shows the specifications of the selected devices. All devices are made by the same vendor, since this makes it more likely that consistent safety margins (σ) have been applied to the specifications in the data sheet to compensate for variation [17]. This is especially important for the I_{DD} current measures we supply to the power model in Chaps. 4 and 5, as it makes the evaluation across devices fairer. Furthermore, it is important to note which data sheet revision and *die revision* are used in the comparison, since SDRAM manufacturers frequently update both documentation and the design of their chips. Appendix B contains timings and currents per memory that were used for the experiments.

Table 3.3 Memory specifications

Name	Clock frequency (MHz)	Data bus width	Capacity (Gib)	Part number	Die revision
LPDDR-266	133	×16	1	MT46H64M16LF-75	B
LPDDR-400	200	×16	1	MT46H64M16LF-5	B
DDR2-800	400	×16	1	MT47H64M16-25E	H
DDR2-1066	533	×16	1	MT47H64M16-178E	H
DDR3-1066	533	×16	1	MT41J64M16-178E	G
DDR3L-1600	800	×16	4	MT41K256M16-125	E
LPDDR2-667	333	×32	2	MT42L64M32D1-3	A
LPDDR2-1066	533	×32	2	MT42L64M32D1-18	A
LPDDR3-1333	667	×32	4	EDF8132A1MC-15	1
LPDDR3-1600	800	×32	4	EDF8132A1MC-125	1
DDR4-1866	933	×8	4	MT40A512M8-1G9	A
DDR4-2400	1200	×8	4	MT40A512M8-2G4	A

The access granularities we consider vary from a single SDRAM burst to multiple grouped bursts up to a size of 256 bytes. This range thus includes typical cache miss sizes (8–64 bytes), as well as larger DMA or accelerator-based transactions.

3.4.2 *Evaluation of Pattern-Generation Heuristics*

The lengths of the patterns determine the efficiency of a pattern set; a shorter pattern is preferred over a longer pattern if it transfers the same amount of data. To evaluate the quality of the BS BI and BS PBGI heuristics, we use the ILP formulation from Sect. 3.2.4.

3.4.2.1 Non-DDR4 Memories

The ILP formulation and the BS BI heuristics are used to generate read and write patterns for the selected memories (Table 3.3), for all (BI, BC) combinations with access granularities up to 256 bytes, and we compared the resulting pattern lengths. For the non-DDR4 memories, we conclude that *the bank scheduling heuristic generates patterns of the same length as the ILP formulation for all considered devices except LPDDR3-1333*, and is hence optimal for most devices.

For two LPDDR3-1333 configurations (4, 2) and (2, 4) BS BI is nonoptimal; the write patterns are 1 cycle too long in these cases. Figure 3.12 shows the patterns generated by BS BI and the ILP formulation for the first of these two configurations.

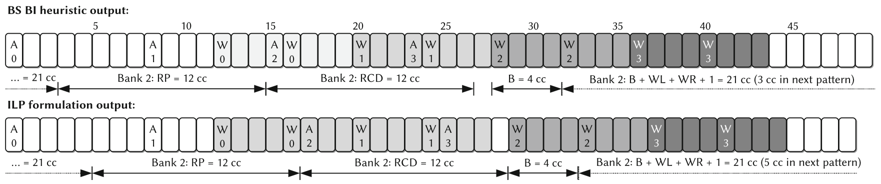


Fig. 3.12 Exceptional nonoptimal result for LPDDR3 in the (4, 2) configuration

The nonoptimality is caused by an activate command that is scheduled earlier than RCD (ACT-to-RD/WR) cycles before the write command, due to a conflict with an earlier write command that occupies the desired spot in the schedule. As a result, the distance between the precharge from a previous pattern incarnation and this activate shrinks a cycle. Once all commands are scheduled, the heuristic detects that the PRE-to-ACT constraint for this pair is violated, and it compensates by making the pattern one cycle longer. An alternative conflict resolution strategy, which postpones both the write command and the activate, leads to the optimal solution, but this cannot be determined at the time the heuristic makes the decision without introducing cycles in the algorithm. Given how exceptional this effect is (2 out of 120 non-DDR4 configurations are affected), and its relatively low cost ($<2\,\%$ length increase), we do not explore this further.

3.4.2.2 DDR4 Memories

For the DDR4 memories, we generate patterns using both the BS BI and BS PBGI heuristics and the ILP formulation. Figure 3.13 displays the resulting write pattern lengths for a DDR4-1866 memory (the trends for the read patterns and the DDR4-2400 look the same). For access granularities where BI and BC are both larger than 1 (and could hence use bank-group interleaving), BS BI generates patterns that are on average 8 % longer than the optimal length. If we consider both BS BI and BS PBGI then there are only two configurations left where neither BS BI or BS PBGI are optimal (the trends look similar for DDR4-2400).

(4, 2) suffers from the same effect as the LPDDR3-1333 pattern that was discussed earlier. The remaining configuration, (2, 16), uses a complex bank interleaving order in the ILP's solution. Consider, for example, the generated interleavings (target bank id of a burst is given by the number in the list) and pattern lengths for the (2, 16) write patterns:

$$\begin{aligned}
\text{BSBI}_{(2,16)} &\rightarrow \{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1\},\ 168\ cc\\
\text{BSPBGI}_{(2,16)} &\rightarrow \{0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1\},\ 174\ cc\\
\text{ILP}_{(2,16)} &\rightarrow \{0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1,1,1,1,1,1,1\},\ 153\ cc
\end{aligned}$$

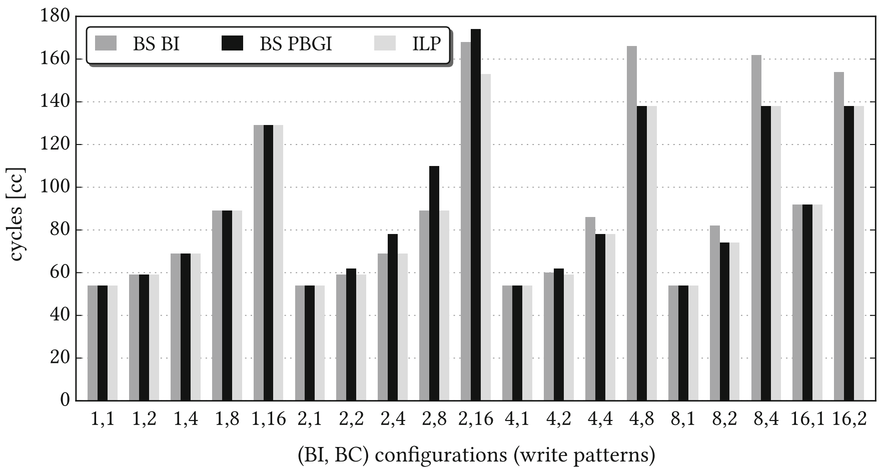


Fig. 3.13 Comparison of write pattern lengths for DDR4-1866 using bank scheduling (BS BI), bank scheduling with pairwise bank-group interleaving (BS PBGI), and the ILP formulation (ILP). Lower is better

The ILP solution behaves almost like a hybrid of the two heuristics, although its exact properties are dependent on the particular numerical values of the timing constraints for the memory device under consideration.

Since the run-time of both heuristics is negligible (it takes less than a second to generate all heuristic-based patterns in Fig. 3.13), it is feasible to execute them both for configurations where BI and BC are larger than one, and then select the best result. Cases may exist where the read pattern is smaller in BS PBGI, while the write pattern is smaller in BS BI or vice versa. *We propose to select the pattern set that delivers most worst-case bandwidth in those cases*. Note that the read and the write patterns *have* to use the same bank interleaving order to avoid permuting the data when sequentially reading and writing the same address.

For access granularities where BI and BC are both larger than 1, the average pattern generated by this procedure is 1.1 % larger than optimal. Since the potential gains of creating a more refined heuristic that mimics the ILP solutions more closely are quite small, and because it is not straightforward to define it generically, *we propose to use a combination of BS and BS PBGI as a fast way to generate patterns*, i.e., run both algorithms, and select the shortest patterns. This also keeps the heuristic simple enough to allow online implementations, although pre-computing and storing the relevant pattern sets instead of the heuristic in the memory controller's driver would generally be more space-efficient.

As one might expect, generating a solution through the ILP formulation is significantly more time consuming than using the heuristics. The number of variables and constraints in each problem is a function of the number of commands to schedule ($3 \cdot \mathrm{BI} + \mathrm{BI} \cdot \mathrm{BC}$), and of the upper bound on the optimal length. The largest problem we generated is the $(16, 2)$ configuration for DDR4-1866, which contains 14780 variables and 2848 constraints, while the average size for this memory is 3431 variables and 1083 constraints. It takes about a second to generate one small pattern with just a single burst, and about an hour for the biggest patterns with 32 bursts. Using the ILP solution offline may therefore be feasible if the access granularity and thus the number of bursts are small enough, and the required number of iterations over different configurations and SDRAM types is limited. The selected read and write patterns should use the same bank interleaving order, for the same reason mentioned earlier. Extra constraints may be added to the formulation to force a matching order once one of the two patterns is generated, although we did not explore this option. Overall, this experiment shows that *efficient patterns can be generated in reasonable time, either optimally within hours, or near-optimally in a second.*

3.4.3 Composable Patterns

This section evaluates the composable pattern conversion that was introduced in Sect. 3.3.1. Section 3.4.3.1 evaluates the efficiency loss with respect to predictable pattern sets. Section 3.4.3.2 demonstrates that the VHDL instance of the proposed controller delivers composable performance to its clients through the application of composable patterns.

3.4.3.1 Conversion Efficiency

In this experiment, we first generate predictable patterns for the same set of SDRAM devices we used earlier, using our heuristic approach. We then apply the composable pattern conversion, and show the conversion efficiencies (e_{pc}, Eq. 3.3) in Table 3.4.

Table 3.4 e_{pc} (Eq. 3.3) for a range of SDRAM ×16 devices

BI	1	1	1	1	2	4	1	2	4	8
BC	1	2	2	4	2	1	8	4	2	1
LPDDR-266	1.00	1.00	1.00	1.00	0.97	0.98	1.00	0.99	0.99	–
LPDDR-400	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	–
DDR2-800	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.99	0.99
DDR2-1066	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
DDR3-1066	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
DDR3L-1600	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
LPDDR2-667	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
LPDDR2-1066	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
LPDDR3-1333	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
LPDDR3-1600	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
DDR4-1866	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
DDR4-2400	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Configurations containing up to 8 bursts per pattern are considered, i.e., the maximum product of BI and BC is 8. There are no BI 8 results for LPDDR memories since they only have 4 banks. The maximum efficiency loss is observed for LPDDR-266 (2.6 %). Only pattern sets that require switching patterns are susceptible to efficiency loss. Switching patterns are usually required when the pattern set implements a large access granularity and, as a result, has a higher inherent efficiency (a more in-depth discussion on pattern efficiency can be found in Chap. 5). The slower the memory, the smaller the access granularity has to be to reach high efficiency, which explains why the slower memories are relatively more likely to suffer. However, the timing constraints on which the patterns are based determine the actual losses. On average, the efficiency decreases by 0.12 % due to the conversion, so the loss is typically negligible. Including configurations with up to 16 bursts changes this number to 0.14 %. *We conclude composable patterns are a low-overhead alternative to delay blocks for real-time clients.* The strength of delay blocks is that they can be enabled or disabled per client (providing predictable performance when switched off), while composable patterns are by definition used by all clients at once. Composable patterns are hence best used in use-cases where all clients require composable performance.

3.4.3.2 Composable Memory Operation

Experiments that demonstrate composability usually *rely on the repeatability of an execution trace*. First, we discuss how such an experiment is generally structured. After that, we explain why we need to deviate slightly from this approach in our experiment.

1. An application *A* is first executed in isolation, where a complete characterization of its behavior is recorded.
2. The execution platform is reset for a second run, where application *A* is combined with another interfering application *B*, with which it shares common resources.
3. The platform and application *A* are assumed to be deterministic, i.e., when the same code (with the same inputs) executes in the second run, the same outputs are produced as in the first run, at the same (relative) time.
4. The reset operation of the execution platform is assumed to be perfect, such that it is in exactly the same state in both the first and the second run when application *A* starts.
5. Given these assumptions, a change in the recorded behavior of application *A* in the second run relative to first run is likely to be caused by application *B*, which would indicate that at least one component in the execution platform is not composable. When no changes are observed, the platform might be composable.

Note that there are many caveats related to the experiment, and no hard conclusions can be drawn from its outcome alone. For example, the absence of change does not prove that a repeat of the experiment with a different interfering application will have the same result. Similarly, delaying the start of application *B* by as little as one

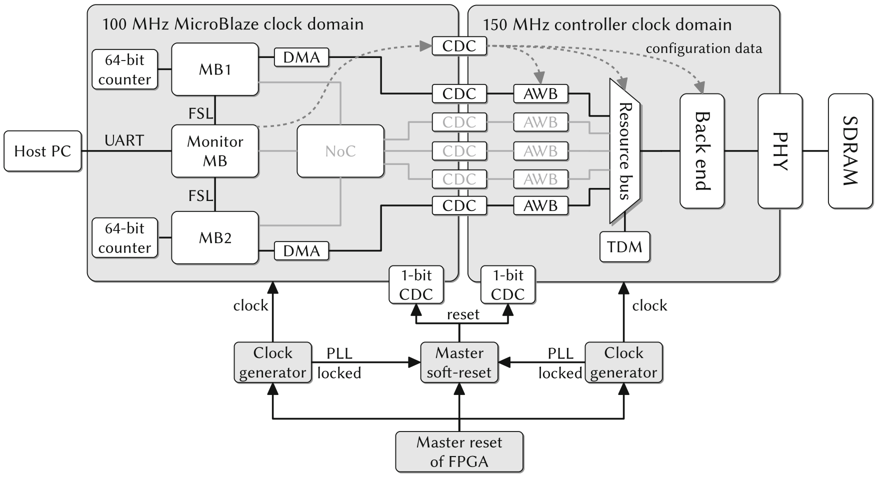


Fig. 3.14 Setup of the composability experiment. To simplify the drawing, we combine the atomizer, width converter, and atom buffer into a single block called AWB. Three ports on the controller are not used and grayed out. The (logical) configuration connections are drawn in *gray*, but their exact path is not shown for simplicity

cycle could reveal or hide changes in the execution trace of A. A single experiment is simply not enough to prove that in all reachable system states and for all possible inputs composability is maintained. Therefore, it can only be supplementary to a larger argument based on the design of the execution platform, which by construction should be convincingly composable. Chapter 2 and Sect. 3.3.1 contain that argument in the form of the description of the memory controller combined with composable patterns, and we demonstrate its workings here with an experiment that follows the same principles as described above.

Experimental Setup

The experiment uses a five-port VHDL instance of the Raptor memory controller, as shown in Fig. 3.14. Two MicroBlaze [18] processors (MB1 and MB2) are connected to the controller through their private DMAs modules, using the first 2 controller ports. Both MicroBlazes also have an (unused) *Memory Mapped I/O (MMIO)* connection to the controller that runs through the NoC, occupying 2 more ports. The fifth port is meant for debugging, and is also not used in the experiment. The MicroBlazes run in a single 100 MHz clock domain, while the controller front-end uses a 150 MHz clock. An additional monitor MicroBlaze can communicate with the host PC, and also configures the memory controller through the configuration port. A timeline of events that happen during each experiment is shown in Fig. 3.15. We now discuss the experimental procedure in detail.

Each run begins with programming the FPGA (loading bitstream). This acts as a nearly perfect reset operation, since most hardware the FPGA contains is fully

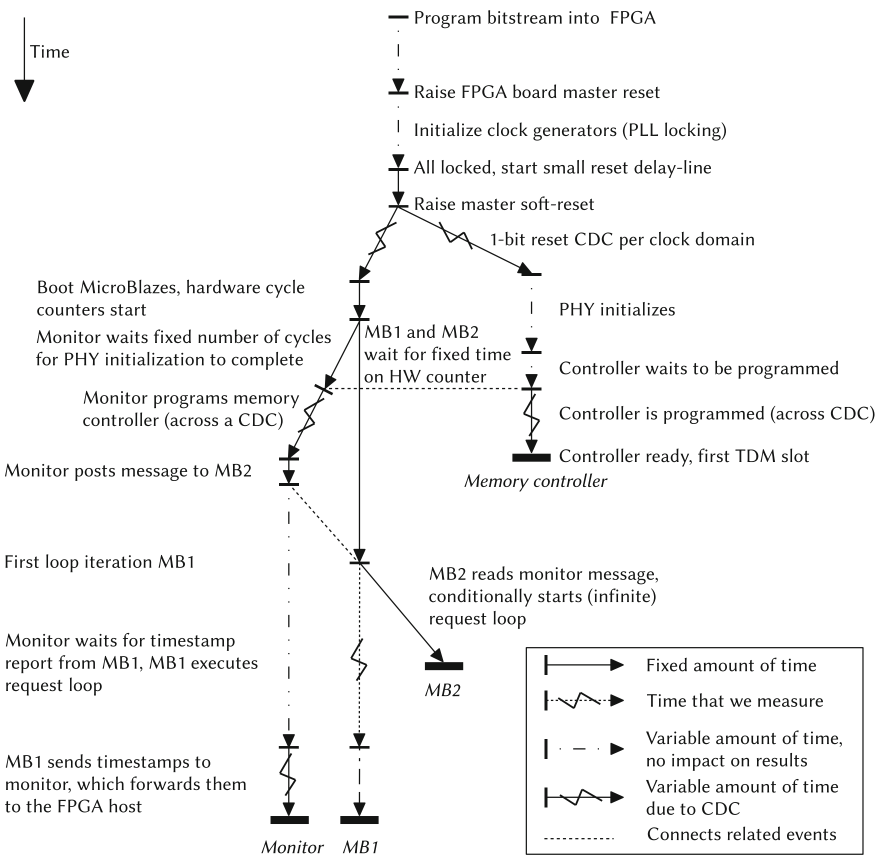

Time
Program bitstream into FPGA
Raise FPGA board master reset
Initialize clock generators (PLL locking)
All locked, start small reset delay-line
Raise master soft-reset
1-bit reset CDC per clock domain
Boot MicroBlazes, hardware cycle counters start
Monitor waits fixed number of cycles for PHY initialization to complete
MB1 and MB2 wait for fixed time on HW counter
PHY initializes
Controller waits to be programmed
Monitor programs memory controller (across a CDC)
Controller is programmed (across CDC)
Controller ready, first TDM slot
Memory controller
Monitor posts message to MB2
First loop iteration MB1
MB2 reads monitor message, conditionally starts (infinite) request loop
Monitor waits for timestamp report from MB1, MB1 executes request loop
MB2
MB1 sends timestamps to monitor, which forwards them to the FPGA host
Monitor
MB1
Fixed amount of time
Time that we measure
Variable amount of time, no impact on results
Variable amount of time due to CDC
Connects related events

Fig. 3.15 Timeline of events during the experiment. The timeline splits when a new parallel group of hardware components is activated. Timelines end when there are no more changes in the behavior of the associated process

re-initialized to a fixed state each time it is reprogrammed. Once programming is done, the *master reset of the FPGA board* is raised. At this time, the clock generators for the controller and MicroBlaze clock domains start their initialization procedure. This involves locking a PLL to a reference clock, which take a variable amount of time per generator, and therefore inserts non-determinism into the system. The effect it has on execution traces is minimized, because a *master soft-reset*, which is part of the synthesized architecture, is not released until they are all locked. This reset signal is synchronized to each individual clock domain through a 1-bit *Clock Domain Crossing* (*CDC*), and within each domain only synchronously reset hardware (based on the domain-specific reset) is used. All clock domains hence start functioning at approximately the same time, but there is no fixed or enforced phase relation between their clocks.

The connections between the clock domains use special-purpose CDC components (containing an asynchronous FIFO) to ensure coherency of multi-bit values as they are passed from one domain into the other. Because the phase relation between the communicating domains can be different for every run, crossing clock domains is inherently nondeterministic [19], the CDCs act as nondeterministic components. This violates one of the assumptions (3) required for the experiment to work flawlessly. *Even an execution trace of an application running in isolation should hence not be expected to be repeatable in our experiment*. However, as the results will show later, we can still with reasonable certainty distinguish changes in the execution trace related to non-composability from this inherent non-determinism, since the former's relative amplitude in this specific setup is much larger.

When the per-domain reset is raised, the PHY begins to initialize, and in parallel the MicroBlazes start executing their code. The monitor MicroBlaze waits for a fixed number of cycles (much greater than the PHY initialization time), and then configures the memory controller front-end and back-end. It then posts a message to MB2 through a debug (*Fast Simplex Link* (*FSL*)) link: the contents of the message determine if MB2 will be actively using the SDRAM in the remainder of the experiment. Both MB1 and MB2 are equipped with a 64-bit hardware counter. They both wait until a fixed time after their local reset based on their counter. This time is chosen to be (much) greater than the PHY initialization time plus the time the monitor needs to configure the controller. As a result, the connection to the SDRAM is initialized before these MicroBlazes become active, thus eliminating the influence of the variable initialization time of the PHY on the experiment.

After its fixed waiting period, MB2 reads the message from the monitor, and conditionally activates a loop that generates sDMA requests for the SDRAM. Each iteration generates two 128-byte write requests, followed by two read requests of the same size. MB1 executes a similar loop, but does this unconditionally. It keeps track of the completion time of each loop iteration in a local memory, and reports them to the host of the FPGA (through the monitor) at the end of each experiment. This trace of timestamps is used as a substitute for the complete behavioral characterization an ideal experiment would record.

A pattern set with BI2 and BC2 is used, corresponding to an atom size of 256 bytes. The DTL requests that are sent by the DMAs are only half of that size, being 128 bytes each, which means that atomizer needs to pad them to fill an entire atom. This in effect exploits the atomizer as a local amplifier of the amount of traffic that each DMA generates, compensating for the bandwidth gap that exists between the 32-bit DMA running at 100 MHz and the controller back-end that can process more than 1.2 GB/s in this configuration. This makes it easier to generate contention with MB2 that has a visible effect on MB1's timestamp trace.

The predictable pattern set with this configuration is write dominant and has read and write patterns of 22 and 32 cycles, respectively, and 0 cycle switching patterns. The composable pattern set has read and write patterns of 32 cycles. A non-work-conserving TDM arbiter determines which port gets access to the memory. Its slot table size is set to 20. MB1 has 10 slots in the table, while MB2 has only 1 slot, and 9 slots are empty (and cannot be used MB1 or MB2). This creates a bottleneck for

MB2, making it more likely for it to be using its slot while MB1 is also present, and hence to generate noticeable interference. The time between scheduling decisions by the arbiter (scheduling interval) is set to 22 cycles if the predictable patterns are used, and to 32 cycles when the composable patterns are used. Four different types (k) of runs are performed:

- $k = 1$: Predictable patterns, only MB1 accesses the SDRAM
- $k = 2$: Predictable patterns, both MB1 and MB2 access the SDRAM
- $k = 3$: Composable patterns, only MB1 accesses the SDRAM
- $k = 4$: Composable patterns, both MB1 and MB2 access the SDRAM

Each type is executed 122 times, for a total of 488 experiments. Each experiment generates a trace of 100 timestamps expressed in tile clock cycles (at 100 MHz), one for each loop iteration of MB1. The first timestamp is collected right after the first read DMA request completes.

Results

To refer to the timestamps that were recorded in the experiments we use the function $S_k^j(i)$. It returns the ith timestamp ($i \in [1..100]$) in run $j \in [1..122]$ of type $k \in [1, 2, 3, 4]$. The absolute values of the timestamps relate to the initial time offset by which we delay the start of the experiment, and are thus relatively uninteresting. Therefore, we choose the first experiment from run type Sect. 3.4.3.2 as the baseline trace, and pointwise subtract its results from all traces. To refer to these relative timestamps we use the function $s_k^j(i)$, which is defined as follows:

$$s_k^j(i) = S_k^j(i) - S_4^1(i) \tag{3.4}$$

All the composable runs ($s_3^j(i)$ and $s_4^j(i)$) would contain only zeros if the experiments were run on a deterministic composable platform. However, different runs can generate different traces, even if they are of the same type, since repeatability is not guaranteed on our nondeterministic platform. Within our window of 100 timestamps, we observed two distinct traces for each run type (marked as “gray (x)” and “black”), present in almost equal numbers, as shown in Table 3.5. To visualize this, Fig. 3.16 plots these two unique traces per type in a graph. Note that the vertical axis ranges of the top and bottom graphs are different.

Table 3.5 The number of runs of a specific type that follow the gray (x) or black plotted trace

Run type k	1	2	3	4
Gray (x) runs	58	67	57	60
Black runs	64	55	65	62

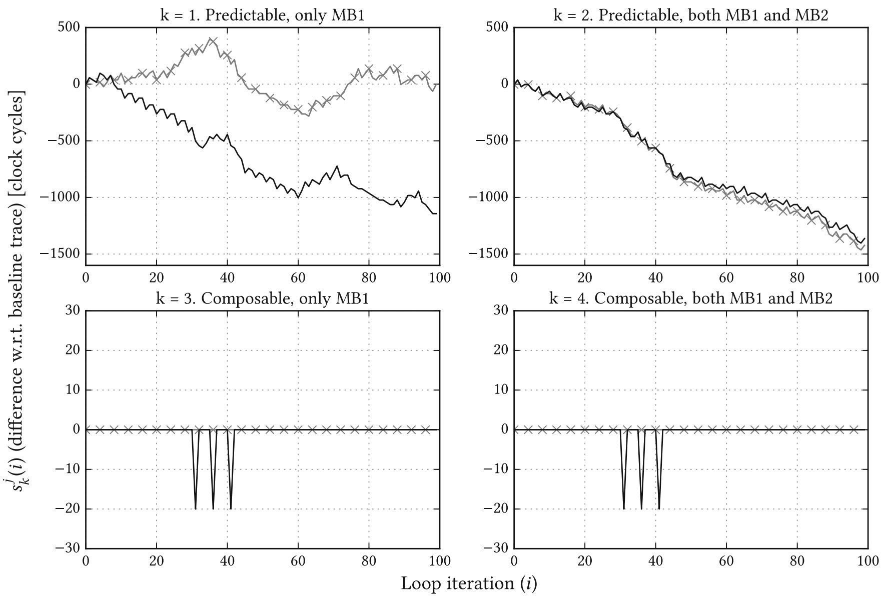


Fig. 3.16 Difference in the execution trace of MB1 in different run types k compared to the baseline trace ($S_4^1(i)$, which corresponds to the gray flatline in graph (4)). In 122 runs, two unique traces are observed for each scenario, drawn as *one gray line* with ‘x’-markers, and *one black line* per graph

Starting with the graph in the upper-left corner (1), we see two different and slowly diverging execution traces for MB1 in different runs. When MB2 is enabled in graph (2), the behavior of MB1 is influenced significantly, indicated by the change in timestamp distribution compared to (1), which again shows two distinct but different traces. *However, when comparing (3) to (4), there is no observable change in the behavior of MB1 as a result of enabling MB2, hence indicating that the memory controller is composable when composable patterns are used.* Table 3.5 shows slight differences in the number of traces that take the path with zero difference from the baseline trace when comparing (3) and (4). This can be attributed to the finite number of samples we took from what is essentially a random distribution of possible phase-offsets between the different clock domains.

The timestamps in $s_2^j(i)$ (graph (2)) are smaller than in $s_1^j(i)$ (graph (1)) for most iterations, which means that the execution time of MB1 in scenario 2 is smaller, even though the load on the memory controller has increased. This can be explained by considering that each write request by MB2 shifts the relative alignment between the TDM slots in the memory controller and the arrival time of requests from MB1, since writes take longer than the default 22 cycle scheduling interval. These changes could have a net positive effect on MB1 in this experiment, because it may cause its requests to capture a slot in an earlier TDM iteration, instead of having to wait for the next one. Another possible explanation originates from the interaction between the DMA and the MicroBlaze. The MicroBlaze polls a status register in the DMA for the completion of requests in a software loop. This loop has a certain length, and polling thus happens at fixed intervals. Changes in the alignment between this loop and the arrival of responses from the memory controller caused by MB2 might again positively or negatively influence the execution time. In short, *the actual-case behavior of the described system is not performance monotonic*, i.e., if one event within the system (like polling the status register) happens sooner compared to a reference run, it might cause another event (like the completion detection in the MicroBlaze) to happen later with respect to that same reference. Note that with a proper analysis model [20], the worst-case behavior is performance monotonic.

What is not visible in the graphs is that the total execution time of the first 100 iterations of the baseline trace, $S_4^1(100)$, is 467552 cycles. This means that even the largest visible deviations from the baseline are still relatively small compared to the absolute execution time. One might argue that there is hence no significant difference in the demonstrated results, and predictable performance is sufficient to be able to verify MB1's behavior is correct. However, there is a large difference between the conclusions that can be drawn from graphs (1) and (2) versus graph (3), because there is *no guarantee or mechanism* that limits the impact of interference on MB1 to the difference between (1) and (2). Changing application MB2 (or its input if was data dependent) or adding a third application to the system would require additional verification runs for MB1. The only conclusion that we can draw based on (1) and (2) is that in the currently tested conditions, the application behaves as shown. In contrast, (3) provides a snapshot of the behavior of MB1 that is independent of the behavior of MB2 or other potential applications, and is therefore a much more general and useful result.

Finally, one might expect the two observed execution traces in (3) or (4) to diverge after the initial 20 cycle difference happens. However, the presence of periodically triggered events, like for example the edges of the TDM slots and the polling loop in the MicroBlaze, can actually hide the visible effects of these (relatively small) timing differences in the starting time of an application loop iteration. This happens because progress halts until such events happen regardless of the precise arrival time of requests. Therefore, a request from MB1 *arriving* 20 cycles (200 ns) earlier could still be *processed* by the memory controller at the exact same time as a later request, considering there is a period of 10 slots in the TDM table (2133 ns) where no service is offered to MB1. Note that there is no guarantee that these two traces will not diverge when the experiment length is increased, nor that only two traces exist. However, the set of possible traces in scenarios (3) and (4) will always be the same, and given enough runs, they should have the same frequency, because the controller is composable.

3.5 Conclusion

The SDRAM controller that is the central theme of this book uses memory patterns, both for its worst-case performance analysis, and in its implementation. This chapter discussed the generation of predictable patterns sets and a conversion method which turns them into composable pattern sets. Two heuristic methods that generate patterns are described, applicable to a wide range of contemporary SDRAM types, including DDR4. Even though the generation of patterns can be seen as a special case of the (old) memory command scheduling problem, we innovated on multiple aspects of its solution in the context of real-time memory controllers. First, we introduce a notational abstraction that allows us to write down our scheduling algorithms in a general fashion, that is, without specifically having to target one SDRAM type. This improves portability, and makes it easier to compare scheduling algorithms across SDRAM generations. Second, we exploit the available degrees of freedom in the low-level memory map within the scheduling algorithm, which determines the distribution of bursts across banks. Both the number of banks a request is interleaved over (BI) and the number of bursts per bank (BC) are parameterized, generating a range of possible pattern configurations. By means of an ILP formulation, we evaluated the quality the pattern-generation heuristics for a range of 12 memory devices, and concluded that their output is close to optimal, while being orders of magnitude faster in terms of generation time.

To create a composable memory resource, we introduced a simple method that turns a predictable pattern set into a composable pattern set. We showed that this conversion has a negligible impact on the predictable performance bounds. Experimentally, we demonstrated the timing-isolating effect of using composable patterns in contrast to using predictable patterns on our FPGA memory controller instance, even in the presence of nondeterministic hardware components.

References

1. JEDEC (2009) Low power double data rate specification JESD209B
2. JEDEC (2010) DDR3 SDRAM specification JESD79-3E
3. JEDEC (2010) Low power double data rate 2 specification JESD209-2D
4. JEDEC (2012) DDR4 SDRAM specification JESD79-4
5. JEDEC (2013) Low power double data rate 3 specification JESD209-3B
6. JEDEC (2009) DDR2 SDRAM specification JESD79-2F
7. 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
8. 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
9. 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
10. 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
11. 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
12. 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 applications symposium (RTAS), pp 145–154

13. Akesson B, Hayes Jr W, Goossens K (2011) Automatic generation of efficient predictable memory patterns. In: Embedded and real-time computing systems and applications (RTCSA), pp 177–184
14. Goossens S (2014) Power/performance trade-offs in real-time SDRAM controllers—code and datasets. http://www.es.ele.tue.nl/~sgoossens/sdram_trade_offs
15. 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 systems and applications (RTCSA)
16. Zhang Z, Zhu Z, Zhang X (2000) A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In: International sympoisum on microarchitecture (MICRO), pp 32–41
17. Chandrasekar K, Weis C, Akesson B, Wehn N, Goossens K (2013) Towards variation-aware system-level power estimation of DRAMs: an empirical approach. In: Design automation conference (DAC), pp 23:1–23:8
18. Xilinx (2011) Microblaze processor reference guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/mb_ref_guide.pdf
19. Vermeulen B, Goossens K (2011) Interactive debugging of systems on chip with multiple clocks. IEEE Des Test Comput 5
20. Nelson A, Goossens K, Akesson B (2015) Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. J Syst Arch

Chapter 4
Cycle-Accurate SDRAM Power Modeling

SDRAM memories contribute significantly to the overall system power and energy consumption of a system and require effective power management for their energy-efficient use. The key prerequisite to their efficient power/energy management is to use accurate SDRAM power and energy consumption estimates. Hence, system designers require high-precision power models that accurately estimate power and energy consumption of the different SDRAM operations, state transitions, and power-saving modes.

All SDRAM vendors furnish a set of standard current measures corresponding to different combinations of memory operations specified by JEDEC. These measures are employed by high-level power models, which break them down into measures corresponding to individual SDRAM operations. However, existing high-level power models lack precision in their modeling of the different SDRAM operations, and hence do not report accurate power estimates.

Alternatively, circuit-level power models can be employed for power estimation, since they perform accurate modeling of these operations, transitions, and modes. However, the underlying SDRAM architectures employed by these circuit-level models are very detailed and specific. They require extensive adaptation to model different SDRAM architectures, i.e. it is not easy to reuse these models across devices or generations.

This chapter addresses this issue by proposing a high-level cycle-accurate SDRAM power model called *DRAMPower*, which employs JEDEC-specified current measures and performs high-precision modeling of SDRAM operations to obtain accurate power and energy estimates. We compare and contrast the state of the art in high-level SDRAM power models against ours, and show how we improve the precision of the modeling of the different SDRAM operations, state transitions and power-saving modes.

Section 4.1 starts with a high-level description of the approach we use in our SDRAM power model. Section 4.2 continues with background on the various standard current measures specified by JEDEC. We introduce the various states an SDRAM can be in with respect to its power consumption in Sect. 4.3, and show how to calculate the energy cost of each command in Sect. 4.4. We first present equa-

S. Goossens et al., *Memory Controllers for Mixed-Time-Criticality Systems*,
Embedded Systems, DOI 10.1007/978-3-319-32094-6_4

tions targeting DDR2/3/4 devices and later adapt them to reflect LPDDR/2/3 and WIDE IO SDRAMs in Sect. 4.5. Section 4.6 combines the equations into an algorithm that accurately calculates the total energy cost of a trace of SDRAM commands. We contrast our approach with related work in Sect. 4.7, demonstrate its accuracy in Sect. 4.8, before ending with conclusions in Sect. 4.9.

4.1 High-Level Description of the DRAMPower Model

SDRAM standards require memory vendors to specify the currents consumed by their memories when specific series of commands are executed by it. These measures are called *I_{DD} currents*. The executed command sequences are defined in such a way that there is at least one I_{DD} measure for each possible *power state* in which the memory can be.

DRAMPower models the energy usage of a command trace based on these high-level currents. We discuss what each of the currents represents in Sect. 4.2. From the I_{DD} currents measures and knowledge of the command sequences that were used to generate them, we then derive:

- The background energy consumed by the memory in each possible power state (Sect. 4.3).
- The energy costs of the individual SDRAM commands (Sect. 4.4), sometimes called the *active* energy components. The commands that are executed determine which power state transitions the SDRAM makes.

By adding the energy related to the executed commands to the contributions of the background energy for each cycle spent in a certain power state, we obtain an accurate representation of the consumed energy by a trace of SDRAM commands. Dividing this energy by the trace length gives us a measure for the average power consumption during the trace.

4.2 Background on SDRAM Currents

In this section, we describe how the different I_{DD} currents are measured. These currents are also described in detail in [1], and their numerical values for the memories we use for experiments in this book are given in Appendix B.

1. I_{DD0} (*One Bank Active-Precharge Current*): The command sequence consists of one ACT, and one PRE command to one bank. Other banks are retained in *precharged* state.
2. I_{DD1} (*One Bank Active-Read-Precharge Current*): The command sequence consists of an ACT, RD and PRE command to one bank, while other banks remain

precharged. This measurement is performed twice, targeting two different memory locations and toggling of all data bits.

3. I_{DD2N} (*Precharge Standby Current*): Measured when all banks are closed (*precharged* state).
4. I_{DD2P0} (*Precharge Power-Down Current—Slow-Exit*): Measured during power-down mode with *Clock Enable* (*CKE*) Low and the *Delay Locked Loop* (*DLL*) locked but off, while the external clock is on and all banks are closed (precharged).
5. I_{DD2P1} (*Precharge Power-Down Current—Fast-Exit*): Measured during power-down mode with CKE Low and the DLL locked and on, while the external clock is on and all banks are closed (precharged).
6. I_{DD3N} (*Active Standby Current*): Measured while executing NOP or DES commands, when at least one bank is open (activated).
7. I_{DD3P} (*Active Power-Down Current*): Measured during power-down mode with CKE Low and the DLL locked, while the external clock is on and at least one bank is open (activated).
8. I_{DD4R} (*Burst Read Current*): The command sequence consists of a continuous series of RD commands, assuming seamless read data bursts with all data bits toggling between bursts and all banks open, with the RD commands cycling through all the banks.
9. I_{DD4W} (*Burst Write Current*): The command sequence consists of a continuous series of WR commands, assuming seamless write data burst with all data bits toggling between bursts and all banks open, with the WR commands cycling through all the banks and the *On die termination* (*ODT*) pin stable at high.
10. I_{DD5B} (*Refresh Current*): The command sequence consists of a continuous series of REF commands, issued every RFC cycles.
11. I_{DD6} (*Self Refresh Current*): Measured during self-refresh mode with CKE Low and the DLL off and reset, while the external clock is off and all banks are closed (precharged).

The I_{DD} currents capture most, but not all of the power consumption of an SDRAM. Other auxiliary power components are associated with every read and write operation. When a write is issued, the external signal used to drive the data to the memory needs to be terminated using termination resistors in the memory module to avoid distortions of other signals in the memory. This termination power is consumed whenever a write is issued and also considers potential other (idle) SDRAM ranks.

Similarly, when a read is issued the power required to drive the data out through the device I/O must also be accounted for and is referred to as the I/O power. Note that the I_{DD} currents found in datasheets have to cover all SDRAM devices that are sold under that datasheet. This means that the given values are conservative with respect to process variation, i.e. they contain safety margins (σ) to account for the worst devices in the produced batch. To obtain accurate I_{DD} currents for a single device, the JEDEC tests should be run on that specific device.

4.3 SDRAM Power State Machine

In the previous section, we discussed how I_{DD} currents are measured. Here, we describe the state machine that models a DDR3 memory in terms of the background I_{DD} currents it consumes as a function of the commands it executes. This state machine is very similar for other SDRAM types. In Sect. 4.5 we explain the differences.

5 different power states can be distinguished for DDR3 memories (see Fig. 4.1):

1. *Precharged*: When all banks are closed, and the memory is not in any of the power-down states. This state corresponds to the I_{DD2N} test. The *precharged* state is entered from the *active* state, in the cycle where the last open bank is precharged, either explicitly with a PRE or PREA command, or when the auto-precharge command triggers. It can also be entered by a PDX command from the *precharged power-down*, or a SRX command from the *self-refresh* state. These are the commands to exit power-down and self-refresh, respectively. Finally, there are two time-triggered state transitions from an *active* state to the precharged state, one as a result of a REF command, and the other as a result of a self-refresh exit (SRX) command.
2. *Active*: When at least one bank is open, and the memory is not in any of the power-down states. The *active* state is entered in the cycle where the first open bank is activated. During a refresh, the memory also spends a fraction of its time

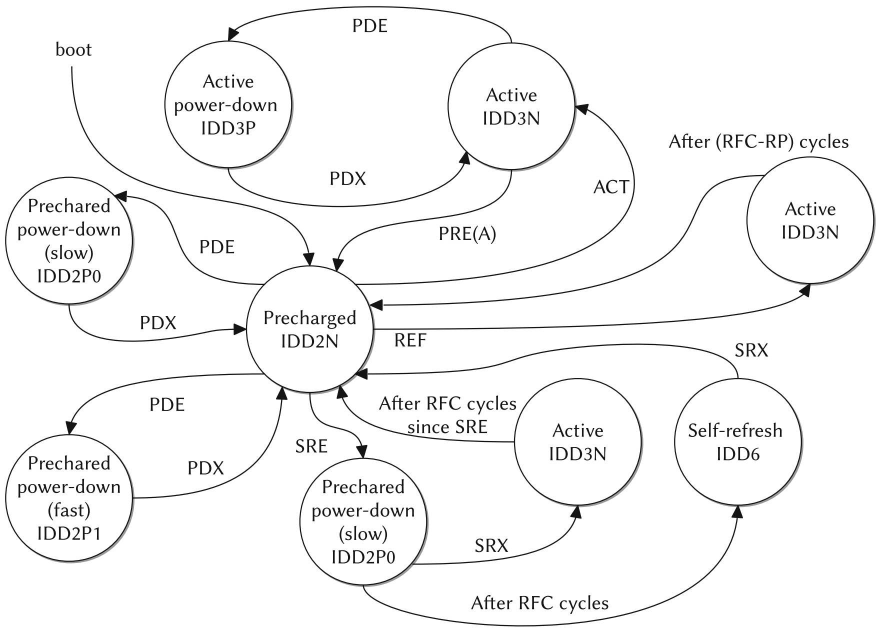


Fig. 4.1 DDR3 power state machine

in this state. Finally, it is also visited when an SRX command is given before RFC cycles have passed since the self-refresh entry (SRE command). This state corresponds to the I_{DD3N} test.

3. *Precharged power-down*: This state is entered if a power-down is issued by means of a PDE command, while the memory is in the *precharged* state. Based on a mode register setting, the memory enters
   - fast-exit mode, consuming I_{DD2P1}.
   - slow-exit mode, consuming I_{DD2P0}. This state is also visited as an intermediate state when entering self-refresh mode, after the SRE command is executed.
4. *Active power-down*: This state is entered if a power-down entry (PDE) is issued while the memory is in the *active* state. This state corresponds to the I_{DD3P} test.
5. *Self refresh*: The self-refresh state starts RFC cycles after a SRE command is executed, unless the SRX command is given before that time. The background current in this state corresponds to the I_{DD6} test.

With this description of the state machine, we now have the means to determine which background current is consumed at any given time by simply tracking the power states a memory visits. Given the duration (t) the SDRAM spends in a specific power state, the consumed background energy (E_{bg}) can be calculated by multiplying the current drawn in this power state (I_{bg}) with the operating voltage V_{DD}. So in total, if the number of cycles in each state is known, using this equation for each state computes the total background energy:

$$E_{bg} = I_{bg} \cdot V_{DD} \cdot t \tag{4.1}$$

4.4 Determining the Energy Cost of a Command

The previous section describes how to calculate the background energy. In this section, we calculate the active component of the energy by considering the energy cost of each individual command. The general procedure for doing this is as follows:

1. Find an I_{DD} test that executes the relevant command.
2. Determine the total energy cost (E) of one iteration of this test. This can straightforwardly be done by taking the current measure (I_{DD}) and multiplying by the duration of the test (t), and the operating voltage (V_{DD}):

$$E = I_{DD} \cdot V_{DD} \cdot t \tag{4.2}$$

3. The energy has an active component (E_a), caused by the logic that toggles as a result of the commands in the test, and a background component (E_{bg}). We are able to subtract the background component from the total energy, since there are other I_{DD} tests that exclusively measure the background current (I_{bg}) in each possible power state. For example, to find the cost of a WR command, we can

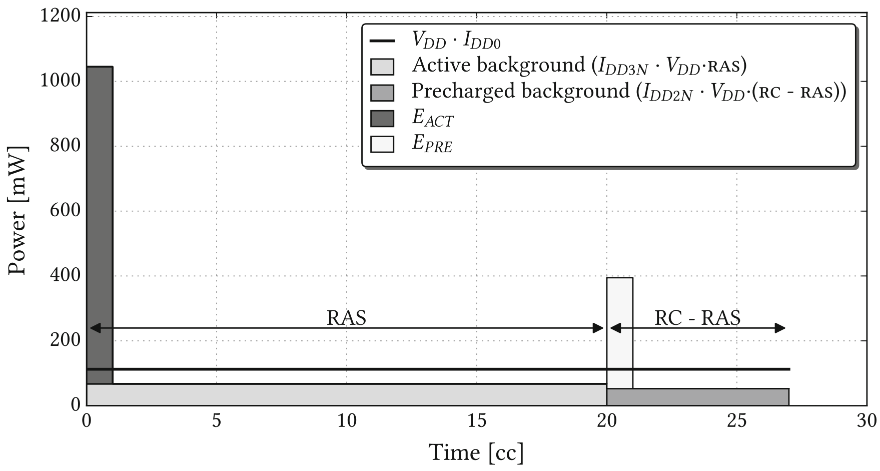


Fig. 4.2 The figure shows how E_{ACT} and E_{PRE} are determined. The *rectangles* represent the modeled distribution of energy during the I_{DD0} test for a DDR3-1066 (Appendix B), using the following parameters: V_{DD} = 1.5, I_{DD0} = 75 mA, I_{DD2N} = 35 mA, I_{DD3N} = 45 mA, RC = 27, RAS = 20. An ACT is executed in cycle 0, and a PRE in cycle 20. Note that the width of the E_{ACT} and E_{PRE} bars is arbitrary, but their combined surface area (representing energy) is not. In reality, the energy of these commands is distributed over multiple cycles

use take I_{DD4W} test, and subtract the active background current I_{DD3N}. In general, the equation looks as follows:

$$E_a = E - E_{bg} = I_{DD} \cdot V_{DD} \cdot t - I_{bg} \cdot V_{DD} \cdot t = V_{DD} \cdot t(I_{DD} - I_{bg}) \qquad (4.3)$$

E_a then represents the energy cost of the commands in the test. In the following sections, we derive E_a for each SDRAM command.

4.4.1 ACT, PRE, and PREA Commands

I_{DD0} specifies the average current consumed by the memory when it executes an ACT command (to transfer the data from the memory array to the row buffer) and a PRE command (to charge the bit lines and restore the row buffer contents back to the memory array), within the minimum timing constraints. We model the total energy spent in one iteration of the test as:

$$E_{I_{DD0}} = I_{DD0} \cdot V_{DD} \cdot t = E_{ACT} + E_{PRE} + E_{bg} \qquad (4.4)$$

E_{bg} consists of the *active* background current I_{DD3N} for the minimum period for which the row is active (RAS) and the *precharge* background current I_{DD2N} for the minimum period for which the row is precharged ($\text{RC} - \text{RAS}$), and we can hence describe the combined energy of an ACT and PRE command ($E_{ACT} + E_{PRE}$) as:

$$E_{ACT} + E_{PRE} = E_{I_{DD0}} - V_{DD} \cdot (I_{DD3N} \cdot \text{RAS} + I_{DD2N} \cdot (\text{RC} - \text{RAS})) \tag{4.5}$$

No I_{DD} test exists that individually captures the energy of an ACT or PRE command, respectively, and pairs of relevant tests are linearly dependent. Therefore, it is impossible to *exactly* calculate the energy per command (Fig. 4.2). To obtain an *estimate* for E_{ACT} and E_{PRE}, we split the right hand side of the equation in two, based on the fraction of time spent in the *active* and *precharge* state in the test.

$$E_{ACT} \approx \frac{\text{RAS}}{\text{RC}} \cdot (E_{ACT} + E_{PRE}) \tag{4.6}$$

$$E_{PRE} \approx \left(1 - \frac{\text{RAS}}{\text{RC}}\right) \cdot (E_{ACT} + E_{PRE}) \tag{4.7}$$

Note that a typical trace contains an equal number of activates and precharges, and is hence not affected by the potential inaccuracy of this split.

Precharge All (*PREA*) commands are often employed when more than one bank has an active row. The PREA command is more efficient in its latency and energy consumption compared to explicit PRE commands to different banks, since it avoids use of multiple commands and takes less time than a series of PRE commands. The energy cost of a PREA command (E_{PREA}) depends on the number of open banks (n_{open_banks}):

$$E_{PREA} = n_{open\_banks} \cdot E_{PRE} \tag{4.8}$$

4.4.2 RD and WR Commands

To determine the energy cost of a RD command (E_{RD}), we use the I_{DD4R} test, which continuously reads from the memory, each burst taking BL/2 cycles. During the test, the memory is in the *active* state, consuming a background current of I_{DD3N}.

In addition to the energy captured in the I_{DD4R} test, we need to consider the I/O energy, as discussed earlier in Sect. 4.2. For this, we rely on the power estimates given in Micron's power model [2]. To calculate the total power for data I/O during a read operation, the I/O power per data bit, P_M^{RDQ} from Table 4 in [2], the number of data lines, IW, and the number of data (byte) strobes IW/8 (Sect. 2.3.3), must be multiplied.

$$E_{RD} = (I_{DD4R} - I_{DD3N}) \cdot V_{DD} \cdot BL/2 + P_M^{RDQ} \cdot (IW + IW/8) \cdot BL/2 \tag{4.9}$$

Similarly, we use the I_{DD4W} test to derive the energy cost of a write command (E_{WR}). In order to calculate the total energy for termination during a write operation, the termination power per data bit, P_M^{WDQ} from Table 4 in [2] is used. In addition to the data lines and strobes, a write command also drives IW/4 mask lines.

$$E_{WR} = (I_{DD4W} - I_{DD3N}) \cdot V_{DD} \cdot BL/2 + P_M^{WDQ} \cdot (IW + IW/4) \cdot BL/2 \qquad (4.10)$$

4.4.3 REF Commands

To determine the energy of a REF command, we use the I_{DD5B} test. The total duration of the test is RFC, of which the first RFC – RP cycles are spent in the *active* state, and the final RP cycles are in the *precharged* state.

$$E_{REF} = I_{DD5B} \cdot V_{DD} \cdot \text{RFC} - V_{DD} \cdot \big(I_{DD3N} \cdot (\text{RFC} - \text{RP}) - I_{DD2N} \cdot \text{RP}\big) \qquad (4.11)$$

4.5 Adaptation to LPDDR and WIDE I/O Memories

The preceding sections used timings and terminology that is specific to DDR2/3/4 memories. LPDDR/2/3 and WIDE IO 3D-SDRAMs are slightly different, since they use multiple power supplies and hence have more than one *voltage domain*, and the power model has to account for this. Effectively this means that for each I_{DD} test defined in Sect. 4.2, there are multiple resulting currents in the SDRAM datasheet, one for each voltage domain. Table 4.1 gives an overview of the domains per memory type. Unfortunately, the naming scheme leads to confusing situations where for example I_{DD0} may refer to two different things, either the command sequence, or the resulting current measured during this test in the V_{DD} domain. Some domains are merged during I_{DD} tests if they use the same voltage value, meaning there is only 1 current measure available in the datasheet for the aggregate of both domains. The I_{DD} measures are suffixed with an identifier for the voltage domain they apply to in the datasheet. These suffixes are also mentioned in the table.

In Sect. 4.4.2, we used P_M^{RDQ} and P_M^{WDQ} to account for the I/O power that is used in the V_{DDQ} domain. For memories for which no I/O power numbers are readily available, the model should calculate the I/O power consumption directly from datasheets using V_{DDQ} domain current estimates. P_M^{RDQ} and P_M^{WDQ} can be set to 0 in those cases, and conversely, we can skip the V_{DDQ} domain for the memories for which we have direct I/O power numbers. In the case of LPDDRs (LPDDR1/2/3 SDRAMs), appropriate I/O circuitry must be employed as recommended by the SDRAM vendor [2]. Also note that the distinction between the slow-exit and fast-exit precharged power-down states does not exist for DDR4 and LPDDR1/2/3, and instead there is only 1 *precharged power-down* power state.

4.6 Trace-Level Energy and Power Calculation in DRAMPower

Sections 4.3 and 4.4 contain all components required to determine the energy and average power of a trace of SDRAM commands. The DRAMPower tool automates this process.

Algorithm 4 DRAMPower pseudo-code

```
1: function DRAMPOWERANALYSIS(T, memoryType)
2:     // N is a map from a command type to an integer (number of instances)
3:     // C is a map from a power state to an integer (number of cycles)
4:     N := COUNTCMDS(T)
5:     C := COUNTSTATECYCLES(T)
6:     $E := 0$
7:     for all dom ∈ VOLTAGEDOMAINS(memoryType) do
8:         $E_{dom} := 0$
9:         for all tp ∈ {$ACT, PRE, RD, WR, REF$} do
10:            $E_{dom} := E_{dom} + \mathbf{N}(tp) \times$ ENERGYPERCOMMANDINDOMAIN(tp, dom)
11:        for all state ∈ ALLSTATES(memoryType) do
12:            $E_{dom} := E_{dom} + \mathbf{C}(state) \times$ ENERGYPERCYCLEINDOMAIN(state, dom)
13:        $E := E + E_{dom}$
14:    return $E$
15: function COUNTCMDS(T)
16:    for all tp ∈ {$ACT, PRE, RD, WR, REF$} do
17:        // Find the cardinality (length) of the set of matching commands
18:        N(tp) := |{ cmd | cmd.type = tp ∀ cmd ∈ T }|
19:    return N
20: function COUNTSTATECYCLES(T, memoryType)
21:    // Assumes commands in trace are sorted by timestamp.
22:    powerState := precharged
23:    t := 0
24:    // Initialize state counters to 0
25:    for all tp ∈ {$ACT, PRE, RD, WR, REF$} do
26:        C(powerState) := 0
27:    for all cmd ∈ T do
28:        // Follow the power state machine from Figure 4.1:
29:        ns := NEXTSTATE(powerState, cmd.type, memoryType)
30:        if ns ≠ powerState then
31:            // Increment number of cycles spent in this power state
32:            C(powerState) := C(powerState) + (cmd.cc − t)
33:            powerState := ns
34:            t := cmd.cc
35:    return C
```

DRAMPower begins by identifying the different memory commands in the command trace (both implicit and explicit), their target bank and issued timestamp. It then inserts one explicit PRE command in place of the auto-precharges, and n_{open_banks} precharges for each PREA command.

Table 4.1 Voltage domains in various SDRAM types

Memory type	Voltage domains (current naming scheme)
DDR2	V_{DD} (IDDxx), V_{DDQ} *
DDR3	V_{DD} (IDDxx), V_{DDQ} *
DDR4	V_{DD} (IDDxx), V_{DDQ} (IDDQxx), V_{PP} (IPPxx)
LPDDR	V_{DD} (IDDxx), V_{DDQ} *
LPDDR2	V_{DD1} (IDDxx1), V_{DD2} (IDDxx2), V_{DDQ} + V_{DDCA} (IDDxxIn)
LPDDR3	V_{DD1} (IDDxx1), V_{DD2} (IDDxx2), V_{DDQ} + V_{DDCA} (IDDxxIn)

Domains annotated with (*) are not involved in all I_{DD} tests. The naming scheme of the current measures corresponding to the voltage domain is mentioned in brackets, where xx needs to be replaced by the identifier of the test, i.e. the *burst read current* for DDR4 in the V_{PP} domain is called I_{PP4R}

The resulting command list (**T**) is forwarded to the analysis phase of the tool, shown in Algorithm 4. Commands are encoded using the notation introduced in Sect. 3.2.1, i.e., each element of **T** is a 3-tuple representing the type, bank, and clock cycle (cc) of a command. The DRAMPowerAnalysis function first counts the number of times each type of command is executed (line 4), and how many cycles are spent in each power state (line 5) using two helper functions. It then simply iterates over the available voltage domains for the memory type of interest (line 7), and sums the contributions of the commands and background power. Both the energyPerCommandInDomain (line 10) and energyPerCycleInDomain functions (line 12) make sure the appropriate currents and voltages are substituted in the energy equations we specified earlier in Sects. 4.3 and 4.4 for the voltage domain of interest, using the substitution rules from Table 4.1.

Algorithm 4 only returns the aggregate energy to keep the algorithm small for the purpose of this book. However, our open-source DRAMPower tool [3] provides a detailed overview of all the components that contributed to this number. It is freely available for download, and has also been integrated into the gem5 simulator [4].

4.7 Related Work

Now we proceed by positioning our model with respect to related approaches. We first contrast our approach with that of Micron [2], which is the most popular (non-circuit level) SDRAM power model, in Sect. 4.7.1. Other models are then discussed in Sect. 4.7.2.

4.7.1 Micron's Approach

Micron derives power equations for different SDRAM operations using the JEDEC-specified datasheet current measures. Additionally, it determines the background power corresponding to the first four power states we showed earlier in Sect. 4.3, similarly to our approach. However, where DRAMPower uses the exact trace of executed commands to exactly track the executed commands and state transitions, Micron takes a more coarse-grained approach. As its input, it requires the page hit rate, and the number of cycles the SDRAM is outputting read data and accepting write data. From this, it estimates the average composition of commands the SDRAM executes. Additionally it takes the percentage of time the memory is in each power state to approximate the background power. DRAMPower is more accurate for several reasons including:

1. Micron does not consider the power consumed during the *state transitions* from an arbitrary SDRAM state to the power-down and self-refresh states (and back), reporting optimistic power-saving numbers for these modes.
2. They also do not include the power expended by the *mandatory precharges* required before a power-down or self-refresh states can be entered. Schmidt et al., empirically verified this shortcoming of Micron's power model in [5].
3. It does not take into account the *power consumed during the pre-refresh clock cycles used to precharge all banks before executing a Refresh*, as a part of Refresh power.
4. It employs the *minimal timing constraints* from the SDRAM datasheets [6, 7] between successive commands, and not the *actual duration* between them as issued by a SDRAM controller, which may well be greater than the minimum constraints. Direct scaling of the power estimates obtained from Micron's power model therefore gives pessimistic power consumption values for basic SDRAM operations, such as reads and writes, if the number of commands is small.
5. It cannot accurately provide power consumption values when an *open-page policy* or a *multi-bank-interleaved memory access policy* [8] is employed. This is because it assumes uniform behavior of all banks (i.e., the same hit ratio for each of them), which in reality is generally not the case.

Schmidt et al. empirically measured the power values from an SDRAM in [5, 9], and showed that Micron's power model provided approximate and worst-case power consumption numbers and over-estimated the actual savings of the self-refresh mode for SDRAMs. They also attributed these discrepancies to the fact that Micron's power model does not cover the state transitions to the self-refresh or the other modes and verified this using different benchmarks.

These critical issues with Micron's power model impact the accuracy and the validity of the power values reported by it. This chapter addresses all of the five aforementioned issues by proposing an improved SDRAM power model (DRAMPower) for all SDRAMs. The precision of the power model using the JEDEC-specified current measures is one of the factors that define the accuracy of the power estimates.

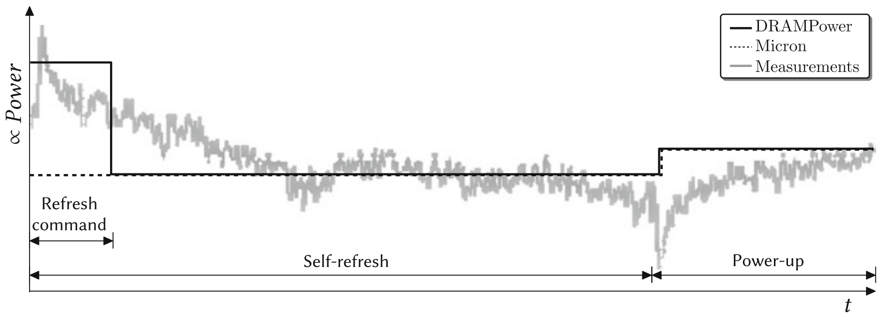


Fig. 4.3 Indication of the difference between Micron's and DRAMPower's way of modeling self-refresh

The proposed power model takes into account all possible state transitions from any arbitrary SDRAM state to the power-down and self-refresh states. Our generic power model accepts a cycle-accurate SDRAM command trace of any length (from a single transaction to an application trace) from any memory controller, supporting both open and close-page policies and any degree of bank-interleaving.

To highlight the improvement in the accuracy of modeling of SDRAM operations in DRAMPower, in Fig. 4.3, we present the difference in the modeling of the self-refresh power-saving mode between Micron's power model (indicated by the dashed line), DRAMPower (indicated by the solid black line) and measurements on a real SDRAM device. As can be noticed in the figure, Micron's power model ignores the internal implicit refresh at the beginning of the self-refresh period, which may prove critical (in terms of power consumption) for shorter self-refresh periods. This effect is captured by DRAMPower unlike Micron's model. Similarly, state transitions to power-down modes or auto-refreshes and use of dynamic command scheduling policies are captured more accurately by our model.

4.7.2 Other Power Models

Our proposed DRAMPower model employs the *actual duration* between commands obtained from any such SDRAM command trace together with the JEDEC-specified current and voltage values. Other existing SDRAM power models, such as Rawson [10], Joshi et al. [11] and Ji et al. [12], propose SDRAM power modeling similar to Micron, but none of them identified or addressed the issue with state transitions and hence, do not provide any improved power estimation numbers. Rawson [10] even simplified Micron's model further with approximate power equations, which were less precise compared to Micron's model. Joshi et al. [11] estimated energy per read/write transaction and assumed all transactions have a fixed timing behavior, ignoring that in reality the times between commands vary. Ji et al. [12] did not model the memory power-saving states or state transitions, and hence, do not provide better power consumption estimates.

4.8 Evaluation

In this section, we present a set of experiments to highlight the accuracy of the power and energy estimates of the DRAMPower model. We compare the output of the tool to measurements on our ML605 [13] FPGA development board, and to the output of Micron's model.

4.8.1 Experimental Setup

We use our FPGA-based experimentation platform as described earlier in Chap. 2, containing an instance of the Raptor controller, to obtain I_{DD} current measurements for a DDR3-1066 memory device (see Appendix B). The measurements are obtained by mounting the DDR3 DIMMs on a JEDEC MO-268 [14] standard-compatible, JET-5466 SO-DIMM extender board [15] equipped with a current-sensing shunt resistor of 100 mΩ. The measurements are done using a high-end Lecroy Wavesurfer 454 Oscilloscope (2 GS/s) reporting at 500 MHz. We employed two channels on the scope, with the two probes connected across the resistor and using a common ground. We used $1\times$ probes for minimal signal loss. The difference between the voltage measures of the two channels indicates the current flowing through the shunt resistor. For average current (and by extension, power) measurements, we take the mean of the voltage drop over more than 100 iterations of the analysis window.

Each experiment we run is effectively a micro-benchmark that executes a certain command or enters a certain power state. The combination of all experiments exercises a representative set of all possible combinations of SDRAM commands. We decided to employ these micro-benchmarks instead of a complete application trace, since this allows us to continuously loop them and obtain an accurate measure of the average current during each test. The power can be expected to be the same for the different operations under realistic workloads as well. The 17 tests we execute perform the following operations:

1. Activates two banks, and then precharges them.
2. Activates four banks, and then precharges them.
3. Activates eight banks, and then precharges them.
4. Activates a bank, precharge it, and then execute a REF command.
5. Activates a bank, precharges it, and then moves to the *self-Refresh* state.
6. Activates a bank, and then moves to the *active power-down* state.
7. Activates a bank, precharges it, and then moves to the *precharged power-down* state.
8. Execute a BI 1, BC 4 read pattern.
9. Execute a BI 1, BC 8 read pattern.
10. Execute a BI 2, BC 1 read pattern.
11. Execute a BI 4, BC 1 read pattern.
12. Execute a BI 8, BC 1 read pattern.

13. Execute a BI 1, BC 8 write pattern.
14. Execute a BI 1, BC 4 write pattern.
15. Execute a BI 2, BC 1 write pattern.
16. Execute a BI 4, BC 1 write pattern.
17. Execute a BI 8, BC 1 write pattern.

DRAMPower is used as described in Sect. 4.6 to derive power estimates. For Micron, we rely on the equations in [2].

4.8.2 Results

For each test, we present the power consumption estimates of DRAMPower and Micron's model when using the measured I_{DD} currents (from the tested DIMM) as inputs to both the models for a fair comparison in Table 4.2. We also show the obtained accuracy when the I_{DD} currents from the memory's datasheet are used as input for the models.[1] As mentioned earlier in Sect. 4.2, the datasheet currents contain safety margins to conservatively cover all devices the datasheet applies to, and hence using them leads to less accurate results if only a *single specific* device is considered. However, DRAMPower should be used with the datasheet numbers if its output has to be conservative for *any* manufactured device.

Figure 4.4 shows the relative differences of the results the two models provide with respect to the measured power. As evident from Table 4.2 and Fig. 4.4, DRAMPower performs much better compared to Micron's power model in all cases, with an average accuracy of 97 % (calculated as the arithmetic mean of the absolute error percentages). In comparison, Micron's model achieves around 82 % accuracy. This highlights the importance of high-precision cycle-accurate modeling of power state transitions and accurate scaling of power estimates based on actual observed (trace-level) timings between commands, as performed by DRAMPower.

In case of self-refresh test (#5), the accuracy of DRAMPower drops, and the deviation from the measurements is 8 %. This can be attributed to the fact that our model assumes a digital implementation of the DLL in the DIMM. However, the particular DIMM tested shows characteristics of an analog implementation of DLL [17], as evident from the measured gradual drop in power consumption when the clock is gated or turned-off. The power estimation accuracy improves with longer self-refresh periods.

[1]A more extensive set of test results can be found in [16].

Table 4.2 Comparison of DRAMPower and Micron against measurements

Test #	Measured	Based on measured I_{DD}s				Based on datasheet I_{DD}s			
		DRAMPower		Micron		DRAMPower		Micron	
1	490.0	500	3 %	659	35 %	689	41 %	970	98 %
2	646.5	647	1 %	874	36 %	859	33 %	1260	95 %
3	658.5	655	1 %	885	35 %	871	33 %	1278	94 %
4	687.0	693	1 %	705	3 %	1184	73 %	1206	76 %
5	66.5	72	8 %	58	13 %	135	104 %	111	67 %
6	73.0	78	7 %	91	25 %	177	142 %	197	170 %
6	151.5	154	2 %	155	3 %	273	80 %	275	82 %
8	1002.0	980	3 %	1050	5 %	1191	19 %	1316	32 %
9	1119.0	1108	2 %	1152	3 %	1325	19 %	1405	26 %
10	835.5	847	2 %	1000	20 %	1060	27 %	1330	60 %
11	1165.5	1182	2 %	1409	21 %	1435	24 %	1836	58 %
12	1446.0	1462	2 %	1751	22 %	1749	21 %	2260	57 %
13	631.5	631	1 %	669	6 %	958	52 %	1023	62 %
14	562.5	551	3 %	604	8 %	830	48 %	923	64 %
15	703.0	672	5 %	819	17 %	945	35 %	1205	72 %
16	877.5	870	1 %	1074	23 %	1197	37 %	1557	78 %
17	1104.8	1099	1 %	1368	24 %	1489	35 %	1965	78 %
Avg. accuracy (%)			97 %		82 %		52 %		26 %

The displayed numbers represent power (mW), and the absolute percentual deviation from measured power (%)

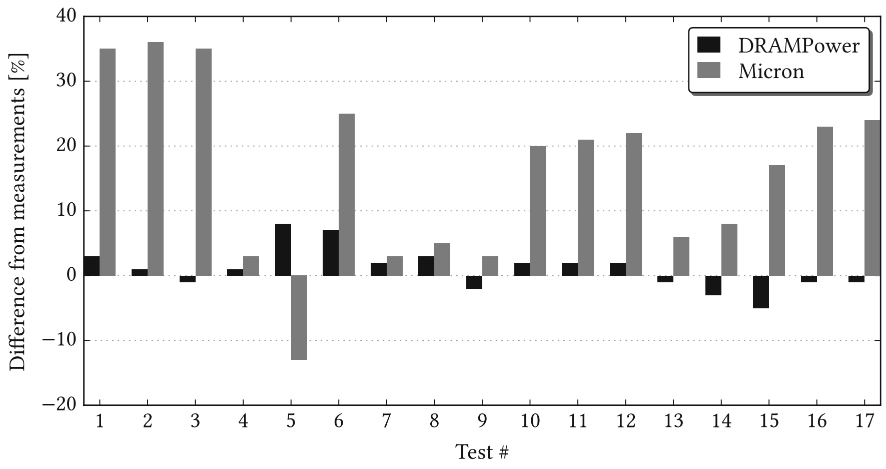


Fig. 4.4 Difference between measurements and the output of DRAMPower and Micron's model, respectively

4.9 Conclusion

In this chapter, we proposed DRAMPower, a high-level SDRAM power model that employs JEDEC-specified current metrics and performs high-precision modeling of the power consumption of different SDRAM operations, state transitions, and power-saving modes at the cycle-accurate level. In order to do this, we derived equations describing the background energy consumption in each power state, and the energy cost of each individual SDRAM command. We then described an algorithm that applies them to a trace of SDRAM commands and thusly calculates the energy expended by it.

We also highlighted the differences between DRAMPower and other existing high-level power models like Micron's and provided updated and new power equations to achieve more accurate power and energy estimates. Finally, we described the open-source DRAMPower tool, and how it employs the model for its analysis and computations. We experimentally showed DRAMPower provides accurate estimates of SDRAM power usage, and performs better than the de-facto standard model from Micron using a set of micro-benchmarks.

References

1. JEDEC (2010) DDR3 SDRAM specification JESD79-3E
2. Micron (2007) Calculating memory system power for DDR3. Technical report, Micron Technology Inc. TN-41-01
3. 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
4. Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA (2011) The gem5 simulator. SIGARCH Comput Arch News 39(2):1–7
5. Schmidt D, Wehn N (2009) DRAM power management and energy consumption: a critical assessment. In: Proceedings of the 22'nd annual sympoisum on integrated circuits and system design: chip on the dunes, pp 32:1–32:5
6. DDR2 SDRAM (2007) Micron, 1GbDDR2.pdf - Rev. Z 03/14 EN edition
7. DDR3 SDRAM (2006) Micron, 1Gb_DDR3_SDRAM.pdf - Rev. L 03/13 EN edition
8. Rixner S, Dally WJ, Kapasi UJ, Mattson P, Owens JD (2000) Memory access scheduling. In: Computer architecture, international sympoisum (ISCA), pp 128–138
9. Schmidt D, Wehn N (2009) A review of common belief on power management and power consumption. Technical report, Technische Universitt Kaiserslautern
10. Rawson F (2004) MEMPOWER: a simple memory power analysis tool set. Technical report, IBM Research Division. RC23068 (W0401-091)
11. Joshi A, Sambamurthy S, Kumar S, John L (2004) Power modeling of SDRAMs. Technical report, The University of Texas at Austin. TR-040126-02
12. Ji J, Wang C, Zhou X (2008) System-level early power estimation for memory subsystem in embedded systems. In: Fifth IEEE international symposium on embedded computing, SEC'08, pp 370–375
13. Xilinx (2011) ML605 documentation UG533. http://www.xilinx.com/support/documentation/boards_and_kits/ug533.pdf
14. JEDEC (2014) 240 pin DDR3 DIMM, 1.00 mm pitch MO-269J

15. JUJET (2015) JET-5466 extender. http://www.eztest.com.tw
16. Chandrasekar K (2014) High-level power estimation and optimization of DRAMs. PhD thesis, Delft University of Technology
17. Kim KY, Choi DM (2006) Delay stage-interweaved analog dll/pll. US Patent 7,149,145

Chapter 5
Power/Performance Trade-Offs

A range of possible memory patterns can be generated within the design space provided by the BI and BC parameters that were discussed in Chap. 3. This immediately raises a very obvious question: how should we choose which (BI, BC) to use? Unfortunately, there is no clear-cut answer to this question. This chapter tries to explain why this is so by means of a discussion of the influence of the (BI, BC) parameters on the worst-case power as calculated based on the model in Chap. 4, and the memory performance calculated with the model from Chap. 2. There is not much that can be said about these metrics analytically; simply looking at the power model or the command scheduling algorithms will provide one with very few hints on the sensitivity of their outputs to the two parameters in question. Once the models are concretized with the numerical values of the involved currents and timing constraints, it is possible to evaluate their output, but then generality is lost. Our approach is to evaluate a large number of different memory devices, and describe the observed trends.

This chapter first describes how the worst-case bandwidth, energy, and power metrics are calculated in Sect. 5.1. Section 5.2 applies these calculations to twelve memories from six SDRAM generations. The observed trends and trade-offs between different pattern configurations and memory modules, both within and across generations, are discussed here as well. Section 5.3 looks at the influence of (BI, BC) on the worst-case response time of an atom. Finally, in Sect. 5.4, we apply the worst-case bandwidth analysis to the Raptor instance of the memory controller, and we experimentally show that its behavior accurately matches the worst-case model.

5.1 Worst-Case Bandwidth, Energy, and Power Metrics

The worst-case analysis of memory patterns in terms of bandwidth has been extensively discussed in related work [1–3]. We apply the same procedure to derive our results as described in those works, as discussed earlier in Sect. 2.4.2.2. To determine the power and energy costs of a pattern set, we rely on the open-source DRAMPower tool [4], which implements the power model we explained in the previous chapter. Section 5.1.1 explains how it is used in detail.

5.1.1 Calculating Worst-Case Power and Energy Efficiency

To estimate the power and energy associated with each pattern configuration, we used the DRAMPower tool, as discussed earlier in Sect. 4.6. DRAMPower takes a trace of SDRAM commands and a description of the modeled memory device as input. For each pattern set, we generate a write and a read trace by concatenating 1000 read or write patterns, respectively, interleaved with a periodic refresh pattern according to the refresh period requirement (REFI) of the memory device, and then supply these traces to DRAMPower. It uses these traces to estimate the average power consumed when *continuously* serving read and write requests, denoted as p^r_{wc} and p^w_{wc}, respectively. This procedure generates an upper bound for the average power, under the assumption that alternating between read and write patterns never consumes more energy than sticking to a single pattern type. Alternating between read or write patterns could only introduce more NOPs into the command stream in the form of switching patterns, and since they do not incur an additional energy penalty (not larger than read or write commands), this assumption holds. The larger of the read and write power is selected as our *worst-case power* metric, p_{wc}:

$$p_{wc} = \max(p^r_{wc}, p^w_{wc}) \tag{5.1}$$

Dividing the energy consumption of a trace by the amount of data it transports yields a measure of the memory's (inverse) energy efficiency for that trace. The energy efficiency can be arbitrarily small; the more idle time a trace contains, the lower its utilization is, and the smaller its energy efficiency will be, since there is a static background power component that is not traffic dependent. To attach any meaning to an energy efficiency number, we need to exclude, or at least constrain the amount of idle time, similar to the worst-case bandwidth, where idle patterns were excluded from the worst-case analysis. Dividing p_{wc} (J/s) by b_{wc} (B/s) yields a number that also has $joule/byte$ as unit, although the interpretation for it is less straight-forward. In general, the worst case for bandwidth is not the same as for power, because the command sequences in the worst case for bandwidth have a relatively low command and data bus utilization, while the worst case for power has high utilization. If p' represents the power consumed by a worst-case bandwidth trace, and b' is the bandwidth delivered by a worst-case power trace, the following relations must hold:

$$\text{assuming } p_{wc},\ b_{wc},\ p',\ b' > 0 \qquad \left\{ \begin{array}{c} p_{wc} \geq p' \wedge b_{wc} \leq b' \Rightarrow \\ \dfrac{p_{wc}}{b_{wc}} \geq \dfrac{p'}{b_{wc}} \wedge \dfrac{p_{wc}}{b_{wc}} \geq \dfrac{p_{wc}}{b'} \end{array} \right. \tag{5.2}$$

This means that the value p_{wc}/b_{wc} can underestimate (but not overestimate) the energy efficiency of both a worst-case power and worst-case bandwidth trace, and hence we interpret it as a lower bound on the energy efficiency for these two modes of operation.

In the following sections, we refer to the efficiency metric from Sect. 2.4.2.2 as (memory) efficiency, while energy efficiency is always explicitly called *energy efficiency*.

5.2 Worst-Case Bandwidth/Power Trends

This section uses the pattern-generation heuristics from Chap. 3 to generate pattern sets for the 12 memory devices we used earlier (Table 3.3). Appendix B contains their detailed specifications. Based on the generated pattern sets, we determine b_{wc} and p_{wc}, and plot the results in Fig. 5.1a, b. The vertical axis displays b_{wc}, expressed in GB/s, with each vertical tick representing a 20 % increase in efficiency, such that the graph covers a range from 0 to 100 % memory efficiency. The horizontal axis represents p_{wc}, expressed in mW, starting at 50 mW for each graph.

Each pattern configuration is identified by two numbers, BI and BC. The data points in Fig. 5.1 are annotated with these two numbers. Configurations are grouped by access granularity (using the marker shape), ranging from 8 to 256 bytes per pattern. The minimum access granularity differs based on the width of the data bus, and hence certain access granularities are not available for all memories.

The diagonal isolines in Fig. 5.1 connect points with equal p_{wc}/b_{wc} quotients, indicative of the energy efficiency of a pattern set (with the caveats mentioned in the previous section). Labels at the top and right of the graphs are associated with the closest isoline, showing the energy cost per bit in [pJ] (125 divided by these labels yields gigabytes per joule). Note that this is the only numerical value in the graphs that can be fairly compared across all memories, since it removes the dependence on the clock frequency and the data bus width.

There are many ways to read this graph. Ideally, a configuration should be as close to the upper-left corner as possible, i.e., have high bandwidth and energy efficiency, and low power. Within one graph, comparing configurations with the same access granularity (marker) shows the effect of trading BI for BC. In the DDR3-1066 graph, for example, (4, 2) is objectively better than the (8, 1), since they are transparently interchangeable from the client's point of view, but (4, 2) is better in the three plotted performance metrics.

Pairs of graphs that belong to the same SDRAM type have their data points in approximately the same relative position, but both the power axis and bandwidth axis are scaled up with frequency. Comparing DDR2-1066 with DDR3-1066 shows a significant drop in power usage on a configuration-by-configuration basis, while the bandwidth remains almost constant, indicating that their timing constraints are very similar.

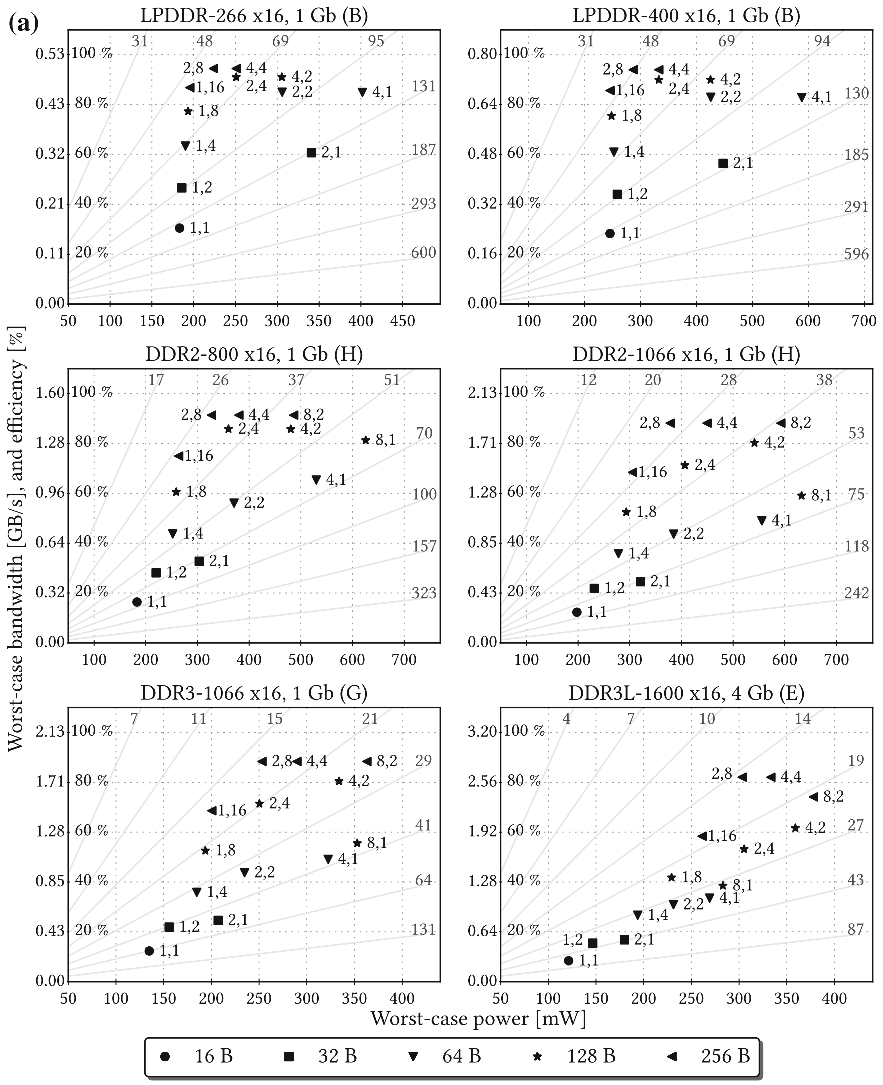


Fig. 5.1 Worst-case bandwidth versus worst-case power (part 1). Graph titles contain the type, data bus width in bits, capacity, and die revision (Appendix B). Labels at the *top* and *right* of the graphs are associated with the closest isoline, showing the energy cost per bit in [pJ] (125 divided by these labels yields gigabytes per joule)

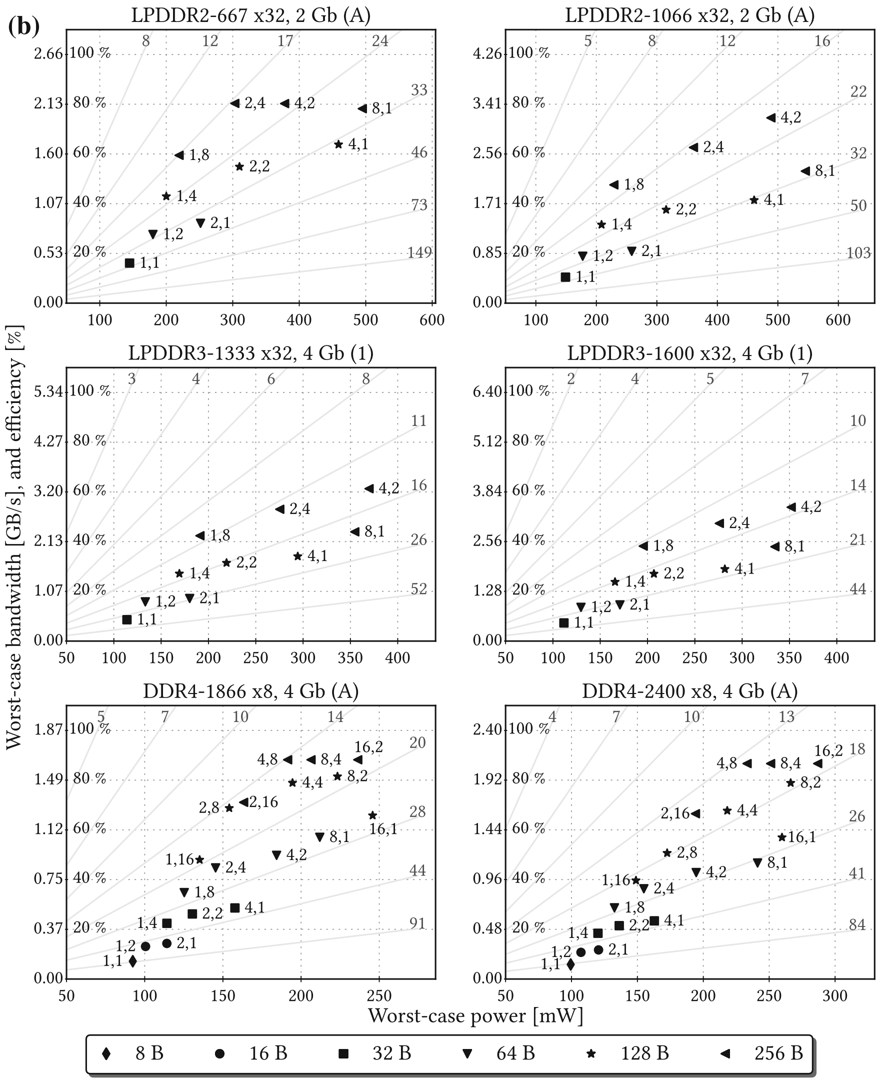


Fig. 5.1 (continued)

The graphs primarily show the trade-offs between worst-case power and worst-case bandwidth (as will be discussed in the next sections), but can also be interpreted differently, outside of the worst-case context. Doing so requires considering what a pattern consists of at the burst level: BI is a measure for the amount of bank parallelism that is exploited, while BC is a measure for the page hit/miss ratio: there are (BC − 1) hits per BC bursts even in the worst case. Each configuration can thus be interpreted as an operating point of the memory as a function of the burst-level bank parallelism and page hit/miss ratio, for which the graphs (coarsely) estimate the delivered bandwidth and power consumption.

5.2.1 Comparing Pattern Configurations of a Single Memory Device

In this initial evaluation, we compare the relative performance of the configurations in Fig. 5.1 on a per-memory basis. Four trends are identified:

1. *For all SDRAM types except DDR4: configurations interleaving over more than four banks ($BI > 4$) are always worse in terms of bandwidth and energy efficiency than another configuration with a similar access granularity, and hence $BI > 4$ should not be used.* The inefficiency is caused by the relatively large four activate window constraints having to be resolved within the pattern instead of across patterns where it overlaps with other constraints. If BI $\geq$ 8, then a pattern contains 8 or more ACT commands. Consequently, it needs to be at least $2 \cdot$ FAW long to be valid, which is always larger than RC for all defined speed bins, and thus more restrictive and unnecessarily expensive compared to using smaller BI with a larger BC instead.
2. For DDR4, a similar effect is visible if BI > 8. This can be explained using similar reasoning, considering that a pattern with 16 ACT commands is at least $4 \cdot$ FAW long, which is at least twice as large as RC for the currently defined speed bins. The FAW timing (in nanoseconds) is slowly reducing as SDRAM technology progresses [5]. Since DDR4 sits on the modern end of the spectrum, it is possible to successfully interleave over more banks in DDR4 compared to the other SDRAM types.
3. *For a constant BI, increasing the access granularity (by increasing BC) improves the worst-case bandwidth and the energy efficiency*, since the energy cost of opening and closing a page is amortized over a larger number of bytes.
4. *For a constant access granularity, interleaving over more banks improves the worst-case bandwidth at the cost of more power. In most cases, the energy efficiency reduces as a result.* The reuse distance per bank increases as bank parallelism is exploited, improving memory efficiency because fewer NOPs need to be added between or within patterns to resolve intra-bank constraints (see Sect. 3.2.1). Energy efficiency reduces in all but 10 pairs of configurations, because the relative number of ACT and PRE commands increases, and they consume energy. In the 10 exceptional cases (DDR3L-1600 (1, 16) and (2, 8) for example), the bandwidth gain is high enough to compensate for the power growth. This trend is also overruled by trends 1 and 2.

A consequence of trends 3 and 4 is that *for a given access granularity, BI can be traded for BC, which corresponds to trading off worst-case bandwidth for energy efficiency.*

5.2.2 Comparing Multiple Speed Bins and SDRAM Types

By comparing the same configurations across speed bins, we can see that *for the same SDRAM type and access granularity, the faster bins generally have a higher energy efficiency*, because the proportional growth of the worst-case bandwidth when switching to a higher speed bin is generally bigger than the proportional growth of the worst-case power within the observed frequency ranges. The $(1, 8)$ configurations for LPDDR2-667 and LPDDR2-1066 demonstrate this (with data points conveniently left and right of an isoline in Fig. 5.1), for example, where the slower device requires more than 17 pJ/bit and the faster device uses less than 16 pJ/bit.

There are three reasons the worst-case bandwidth tends to grow when the frequency increases. The first reason is the most obvious: at a higher frequency, each data burst requires less time, thus potentially reducing the pattern length if the data bursts are on the "critical path" through the pattern. The second reason is that manufacturers design the devices in the higher speed bins to run at the higher clock frequency of that bin, and as a result their analog timings in nanoseconds are also smaller, reducing the pattern lengths in cycles. The third reason arises from the conservative discretization of memory timings that are specified in nanoseconds in the datasheet into clock cycles that the controller can use. Even though the maximum error in the cycle-level approximation of the timing monotonically decreases with an increasing clock frequency (the maximum deviation from the intended delay is always less than one clock period), the actual error does not, such that a higher frequency might occasionally result in a bigger approximation error compared to a smaller frequency that just happens to fit better. The net effect of increasing the clock frequency on the approximation error rarely impacts the worst-case bandwidth negatively.[1] However, in the exceptional cases where it does, it makes no sense to run at these higher frequencies from a worst-case performance point of view.

The worst-case power generally increases as well when a higher clock frequency is used, but because a significant fraction of it is static (related to leakage) and unaffected by the clock frequency, the energy efficiency generally improves.

Increasing the clock frequency has diminishing returns in terms of memory efficiency. The clock period shrinks faster than the pattern lengths, which implies a smaller fraction of the time is spent actually transferring data, assuming the number of bursts ($BI \cdot BC$) in the pattern remains constant. The fraction of time spent waiting for nanosecond-based constraints, for example related to activating and precharging, increases. *This means that the required (BI, BC) product (equivalent to number the of bursts in a pattern) to reach a certain memory efficiency grows with the clock frequency*; this effect is visible in Fig. 5.1, where the same configuration per SDRAM type has a higher efficiency in the slower speed bin than in the faster speed bin.

[1] In the set of experiments presented in Fig. 5.1, only the pair of DDR2 memories shows a bandwidth reduction in a higher speed bin, when $BC = 1$ and $BI \in \{1, 2, 4, 8\}$. The maximum reduction is less than 4 %.

The effects of increasing the width of the data bus on efficiency mirror those of increasing the clock frequency, since it also reduces fraction of time spent transferring data. *Increasing the data bus width thus also has diminishing returns in terms of memory efficiency*. If the pin and wiring costs are of key importance in a particular design, then it may make sense to prefer devices with a smaller data bus width if they can sustain the required bandwidth and are sufficiently energy efficient.

The SDRAM types that are shown in Fig. 5.1 are not all used in the same application area, and some are older than others. Mostly driven by power constraints, the supply voltage has come down over the years, and specific low-power standards (LPDDRX) that typically use a wide data bus have emerged. This move is clearly visible when the memories are ranked by maximum energy efficiency per access granularity, as shown in Fig. 5.2. For the commonly used access granularity of 64 bytes for example, the old LPDDR memories perform the worst, followed by DDR2, DDR3, LPDDR2, DDR4, and LPDDR3. The last four types from this series, but especially LPDDR3, can theoretically do significantly better both in terms of energy efficiency and worst-case bandwidth when the access granularity is increased even further. Without restrictions on the granularity, refresh eventually becomes the only remaining limitation for efficiency. However, it is questionable if these configurations have any practical application. It makes more sense to consider only a limited range of granularities, based on realistic request sizes that a memory client like a cache, *Direct Memory Access* (*DMA*), or accelerator, may generate.

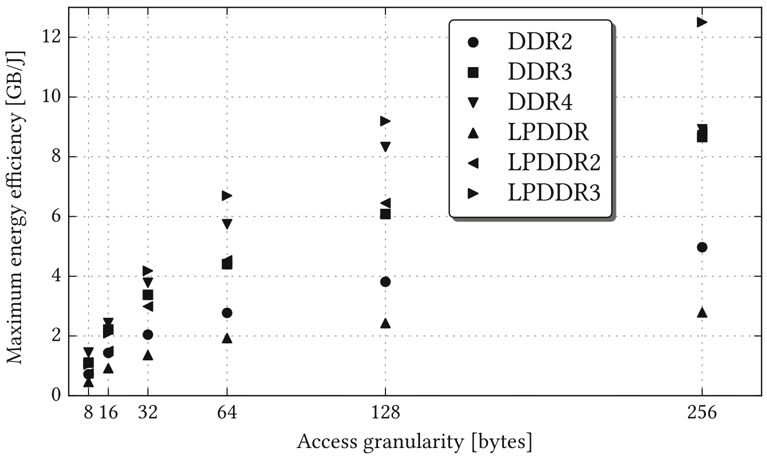


Fig. 5.2 Maximum energy efficiency achieved by the considered pattern sets and memories in Fig. 5.1 at different access granularities

5.3 Worst-Case Response Time of an Atom

The WCSI of an atom, i.e., the maximum time it occupies the SDRAM command bus, is solely dependent on the length of the patterns (assuming the size of the request is not larger than the access granularity of the pattern). The WCRT, i.e., the difference between the arrival of a request in the controller and the departure of the response, depends on more factors, as discussed in Sect. 2.4. It is influenced by the arbitration policy that selects the order in which clients get to use the memory, the number of interfering refreshes, and the number of outstanding (posted) requests the client already has, which is unknown in the general case. An accurate WCRT analysis is thus always a system-specific point solution, and since this chapter focuses on the general trends across memory configurations and types, it is out of its scope. The reader can refer to [1, 6–9] or Sect. 2.4 for a more detailed look into WCRT analysis, while here we focus on the ingredients that the memory command scheduler inserts into that analysis in the form of memory patterns.

Figure 5.3 shows the execution time of the memory patterns for various SDRAM types and configurations (ordered by access granularity and BI). The first two groups of bars represent the entire configuration space for access granularities of 32, 64, and 128 bytes for the two LPDDR2 memories. The other groups show the 64-byte configurations of the fastest memories in our set for the remaining SDRAM types. The *offset* bar shows the time it takes from the start of a read pattern until the final data word is put on the data bus by the memory ($\Delta_r + BI \cdot BC \cdot BL/2$). This may happen after the end of the read pattern, because commands are pipelined in the memory, and thus the offset bar is sometimes larger than the read bar. The total

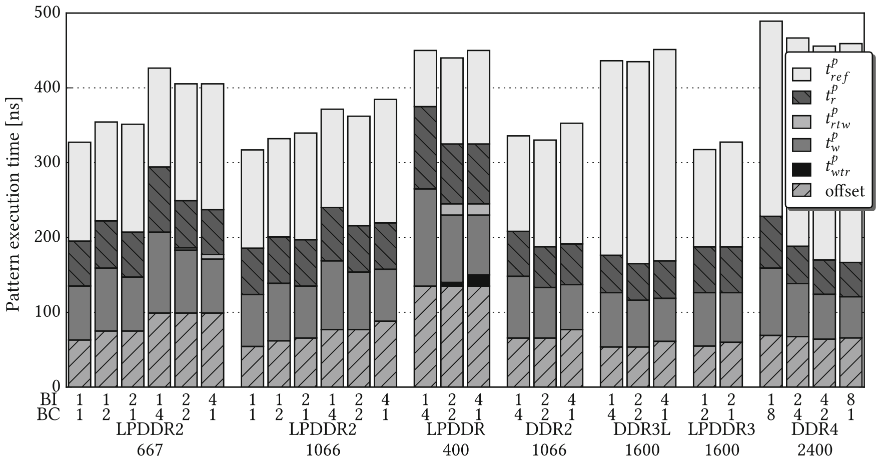


Fig. 5.3 Request WCRT components. From *bottom* to *top*, the stacked bar order is offset, t^p_{wtr} (zero in most configurations), t^p_w, t^p_{rtw} (zero in most configurations), t^p_r, and t^p_{ref}

stacked length of the bars per configuration can be interpreted as the WCRT[2] of a read request that has to wait for a refresh (t^p_{ref}), an interfering read and write pattern (t^p_r and t^p_w) and the associated switching patterns (t^p_{rtw} and t^p_{wtr}), and finally its own data offset.

Comparing the configurations for LPDDR2-667, we can see that the pattern execution times grow as expected (Fig. 3.3) when the access granularity grows. The length of the refresh pattern increases as patterns become more efficient. This happens because a refresh command can only be issued once *all* banks have been precharged, and NOPs are inserted at the start of the refresh pattern to ensure this. Efficient pattern configurations exploit bank parallelism and have their final read or write burst relatively close to the end of the associated pattern, thus increasing the required number of NOPs before the refresh command. The switching patterns grow with efficiency for similar reasons; they insert NOPs between the bursts of patterns of opposing types where they would otherwise be too near each other. At high efficiencies, bursts are scheduled relatively close to the pattern edges, and the switching patterns grow as a result. Note that exchanging read and write pattern duration for longer refresh and switching patterns is not a zero-sum game, because refresh patterns have to be issued infrequently relative to read or write patterns, and the switching patterns are not always in the worst-case sequence of patterns that determines the efficiency.[3]

Comparing similar configurations (same (BI, BC)) across the LPDDR2 memories reveals that increasing the clock frequency reduces the execution time (in seconds) of the patterns, in line with the bandwidth trends. Switching patterns disappear, because the data bus timings that dictate their length are specified in clock cycles and thus shrink in comparison to the analog timings that are based on nanoseconds, which dominate the read and write pattern length at higher clock frequencies.

The refresh pattern length for DDR3 and DDR4 is roughly twice as big as that of the LPDDR3 memory, but LPDDR2/3 memories need to be refreshed twice as often (REFI is half as long, as shown in Appendix B), and hence still have approximately the same refresh efficiency.

Finally, it is interesting to look at the global picture, considering the sensitivity of the pattern execution times to the clock frequency. For example, when comparing the LPDDR2-677 and LPDDR3-1600 (2, 2) configuration, the frequency grows with 140 %, while the duration of a read–write-offset sequence (t^p_r + t^p_w + offset) reduces by only 49 ns or 20 %. This highlights that both new memory technologies and higher clock frequencies do not give much benefit yet in terms of reducing the WCRT of a request.

[2]The pipeline latency of the controller hardware, including the PHY calibration offset (see Sects. 2.3.3) would also have to be added for completion.

[3]70 % of the tested configurations have only write (and refresh) patterns in their worst-case sequence, and 30 % alternate between read and write (and refresh) patterns.

5.4 Evaluation

This chapter relies on the correctness of the worst-case bandwidth and power analysis to predict the performance of various memory modules. Experimentally showing that this analysis model is accurate for *all* these modules is impractical, due to the large variety of specialized (PHY) hardware that would be involved. However, we can at least show that the analysis holds for our VHDL instance. Power measurement requires a relatively complex setup, and has been extensively done in [10] with the same memory modules and FPGA board we use, so we will not repeat that effort. Instead, we focus on the worst-case bandwidth.

Since this chapter primarily considers back-end performance, this experiment will do the same: the resource front-end is omitted in the experimental setup, and all measurements are done directly on the back-end input port. A memory client is connected to this port, it transmits a fixed workload of requests into the controller, and we measure the amount of time required to process them. If the workload is large enough, then the artifacts of starting and stopping the experiment are negligible, and an accurate approximation of the bandwidth can be obtained.

A complicating factor for the experiment is that we can only measure an *actual-case* bandwidth to compare with the outcome of a *worst-case* analysis. We hence try to approach the assumed worst-case conditions as closely as possible by making sure that

1. The memory is fully utilized. Not doing so would yield a measured bandwidth that is too low, simply because the client is not requesting fast enough.
2. The worst-case sequence of requests is executed continuously. This implies switching between read and write requests for mix-dominant pattern sets, or continuously reading or writing for read-dominant or write-dominant pattern sets, respectively.

Satisfying the first condition using a MicroBlaze processor as a client is not viable, since it does not have the capacity to generate traffic in the required large volumes. Instead, we use a configurable traffic generator hardware module, similar to the one that was used in our experiments of [11]. The traffic generator keeps track of the number of cycles it takes to complete a configurable number of requests. The types of the generated requests can be configured to exercise the controller with the three different required sequences of patterns, that is, (1) only writes, (2) only reads, or (3) alternating writes and reads.

The Raptor memory controller instance in this experiment uses a 400 MHz command clock, and has a 32-bit data bus. We generate memory patterns for its memory module in this configuration, and run the worst-case bandwidth analysis. The access granularity that is used by the traffic generator is restricted between its minimum value of 32 bytes (1 burst per pattern), and 256 bytes, corresponding to 8 bursts per pattern, to capture a reasonable range of possible request sizes. The controller is always configured to match this access granularity. Refresh patterns are automatically issued by the controller.

Table 5.1 Raptor worst-case bandwidth (b_{wc}) [MB/s] for an MT4JSF6464H DIMM [12] with f = 400 MHz and IW = 4 bytes for access granularities up to 256 bytes

↓ BC, BI →	1	2	4	8
1	467	933	1862	2360
2	814	1624	2639	–
4	1294	2575	–	–
8	1835	–	–	–

Table 5.1 shows the results of the worst-case bandwidth analysis. The numbers in the table should be a lower bound on the bandwidth that we measure for all request-type sequences in the associated experiment, and should match exactly when the worst-case sequence of requests for a particular pattern set is used. In each experiment, we transfer 128 MB worth of data, such that even the fastest configuration experiences close to 6000 refresh periods, and hence the effect of refresh patterns is sufficiently present in the measurements.

Figure 5.4 shows the results of the experiment as a collection of bar graphs for each (BI, BC) configuration. The first bar in each group represents the (analytical) worst-case bandwidth (b_{wc}), the second, third, and fourth show the measured bandwidth when continuously reading ($b_r^{measured}$), writing ($b_w^{measured}$), or alternating read and write requests ($b_{rw}^{measured}$), respectively. *The figure shows that the worst-case bounds are valid*: the measured bandwidth is always slightly larger than predicted (although this is difficult to see in the graph), while the largest deviation from the bound when a worst-case request sequence is executed is only 0.09 %. Intentionally misconfiguring the memory controller such that a pattern is one cycle longer than expected leads to

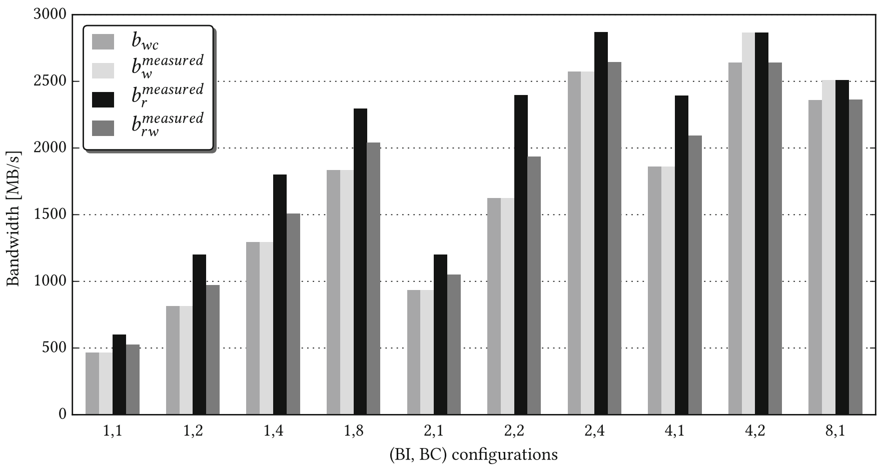


Fig. 5.4 Worst-case and measured bandwidth for different pattern configurations

a violation of the worst-case bound that is at least an order of magnitude larger than this deviation, supporting this conclusion. The dominance class of each configuration can also clearly be observed in the graph by finding the bar of the request sequence that best matches the worst-case bar (the last two sets are mix dominant, the others are all write dominant).

5.5 Conclusion

This chapter discussed the influence of memory pattern configurations ((BI, BC) combinations) on the worst-case performance of 12 memory modules from six different memory generations. We have shown that memory efficiency scales with the access granularity by which it is used. The maximum granularity is limited by the request size of the clients that use the memory, so there is only a limited range of configurations that is practically useful.

For a fixed access granularity, we observed that the BI and BC parameters can be used to trade worst-case bandwidth for energy efficiency. Additionally, we showed that interleaving over more than four banks (or eight banks for DDR4) is never a good idea, since there is always a better configuration available in terms of worst-case bandwidth or energy efficiency at the same granularity. These observations can be used as rudimentary guidelines for the configuration of a scheduling algorithm or the selection of a pattern set, although the fine-grained decision will always depend on the mix of requirements from the memory clients.

Comparing speed bins and different SDRAM types showed that both faster bins and (not unexpectedly) newer memory generations tend to be more energy efficient. We also see that modern memories with wider interfaces and/or higher clock frequencies require larger access granularities to reach the same level of memory efficiency (data bus utilization) as slower/narrower memories, respectively. This trend may be a reason for concern, since it implies that worst-case efficiency will fall even further as memories get wider and faster, unless structural changes are made to reduce SDRAM timings, or clients increase their access granularity to compensate.

Finally, we validated the bound on worst-case bandwidth for our memory controller implementation, and showed that it is tight in the worst case.

References

1. 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
2. Akesson B (2010) Predictable and composable system-on-chip memory controllers. Ph.D. thesis, Eindhoven University of Technology
3. Akesson B, Goossens K (2011) Memory controllers for real-time embedded systems. Springer, Embedded Systems Series

4. 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
5. Micron Technology Inc. (2014) DDR4 networking design guide introduction. TN-40-03
6. 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
7. 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 applications symposium (RTAS), pp 145–154
8. 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
9. 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 applications (RTCSA)
10. Chandrasekar K (2014) High-level power estimation and optimization of DRAMs. Ph.D. thesis, Delft University of Technology
11. Chandrasekar K, Goossens S, Weis C, Koedam M, Akesson B, Wehn N, Goossens K (2014) Exploiting expendable process-margins in DRAMs for run-time performance optimization. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1–6
12. DDR3 SDRAM SODIMM - MT4JSF6464H - 512MB JSF4C64_64x64HY.fm - Rev. B 3/08 EN (2007) Micron

Chapter 6
Conservative Open-Page Policy

The overarching theme in this book is mixed time-criticality memory controllers. So far, the focus has been mostly on the worst-case aspects of such a controller. This chapter changes that by introducing a mechanism that improves its average-case performance.

Memory controllers that use a close-page policy immediately precharge (close) a page when a request is completed. The alternative to a close-page policy is an open-page policy that attempts to exploit locality of reference. It keeps the page open after a request, speculating that a following request wants to access the same page again. If this bet pays off, then no time is spent precharging and subsequently activating the same page again (as would be done in a close-page policy), and instead data can be accessed immediately, improving efficiency. If it does not, then the request experiences the full precharge and activate penalty once it arrives. A close-page policy could have avoided this penalty at least partially, because it can start precharging the page before the request arrives. In systems where multiple independent processors share a single memory resource, requests from different clients are interleaved in an unpredictable manner, making it impossible to guarantee that any locality remains to be exploited (without resorting to bank privatization, discussed in Sect. 8.1.2.3). This is why worst-case oriented controllers in such systems often use a close-page policy.

In this chapter, we will introduce a version of the open-page policy that does not compromise on worst-case guarantees, and can hence be freely used by controllers that care about both worst-case and average-case performance. Section 6.1 explains the intuition behind it, and introduces terminology. Section 6.2 discusses the impact of the policy on the previously introduced controller architecture. The implementation of the policy is refined in Sect. 6.3, such that the average-case performance gains are improved. Section 6.4 evaluates the effectiveness of the policy through experiments with the SystemC instance of the memory controller, followed by conclusions in Sect. 6.5.

S. Goossens et al., *Memory Controllers for Mixed-Time-Criticality Systems*,
Embedded Systems, DOI 10.1007/978-3-319-32094-6_6

6.1 Conservative Open-Page Policy

Exploiting locality is beneficial for the average-case performance of a memory controller, but speculative open-page policies increase worst-case response times. Figure 6.1 shows exactly why this happens: it can take longer to read all the data of a request in a speculative open-page policy than in a close-page policy. In a mixed-time criticality system, certain applications may be bound by tight real-time requirements, and therefore it is undesirable to tamper with the worst-case guarantees. We hence propose a *conservative open-page policy* that exploits part of the available locality without sacrificing worst-case performance. The core idea is to remove the speculation that is normally inherent to the policy by only allowing a page to remain open after an access *when it is certain* that the next request targets the same page. This is conservative with respect to the worst-case guarantees if this decision is made *before* the page would traditionally be precharged by the controller in a close-page policy. The policy can be adopted by all real-time close-page command schedulers, although the implementation effort may vary based on the (original) scheduler.

When the conservative open-page policy is used, the controller executes the following four steps while it serves a request:

1. Commands are executed as normal, as if it were using a close-page policy, until the point where a close-page controller would commit to precharging. Assuming the command schedule is generated by an algorithm similar to those from Chap. 3, this commitment is made when the auto-precharge flag is attached to the last read or write command of the first bank in the pattern.
2. The target address for the next request is now inspected. If the next request is not available yet, or if it targets a different row or set of banks, then the execution of the commands continues as if a close-page policy was used.

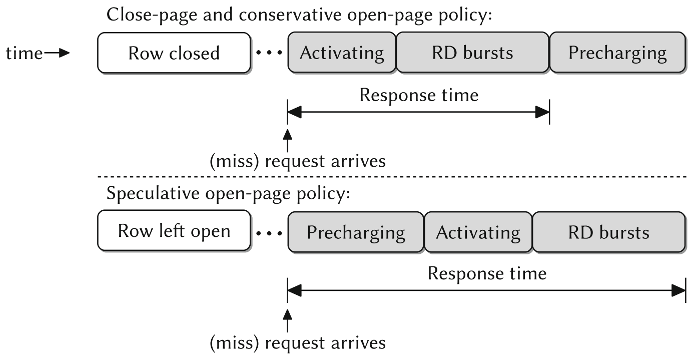


Fig. 6.1 Response time of a hit versus a miss. A miss may have a longer response time in a speculative policy, while the conservative policy behaves similar to a close-page policy

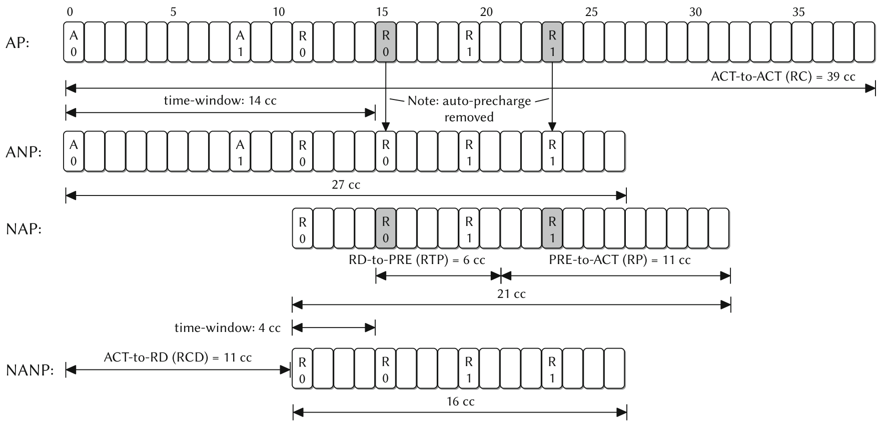


Fig. 6.2 Read schedules for the DDR3-1600 memory in four different modes, for BI 2, BC 2. Each block represents a command, empty blocks represent NOPs. The tinted commands have auto-precharge flags. The timing constraints that dictate the length of the schedule are shown on the *arrows*

3. If the next request targets the same row in the same set of banks, then a hit is detected. The auto-precharge flags are omitted, as are the NOPs that are normally scheduled after the RD or WR commands. These NOPs are normally there to satisfy the PRE-to-ACT and ACT-to-ACT constraints.
4. The command schedule for the next memory access does not incorporate any activate commands, and the NOPs required to satisfy the ACT-to-RD/WR constraint are also omitted.

As a motivating example, consider the first schedule in Fig. 6.2, targeted at a 16-bit DDR3-1600 (Appendix B), the memory that will be used as the running example throughout this chapter. It shows that for BI 2, BC 2, a read atom needs 39 cycles in the close-page policy, of which only 16 cycles are used to transfer data. 59 % of the cycles is spent waiting for ACT and PRE related constraints. The average cost of opening and closing a page (i.e., the time not spent transferring data in the pattern) in all configurations up to a granularity of 64 bytes for this memory is 71 and 77 % for reads and writes, respectively. This shows that our policy (and open-page policies in general) can have a significant impact on the (average-case) efficiency with which the memory is used. Note that the proposed mechanism merely creates the *opportunity* for locality exploitation, but it *does not guarantee* that it will actually happen, because the client is not considered anywhere in our approach, and is hence an average-case optimization.

The conservative open-page policy can use four different *modes* to process an atom. Figure 6.2 shows an example of the read schedules in each possible mode, applied to the DDR3-1600 memory we used before:

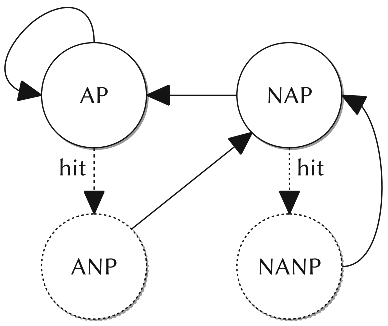


Fig. 6.3 Allowed mode transitions. Schedules in *dotted* modes are not always executed start to finish, but instead begin where the connected mode on their incoming vertex left off when the hit was detected

1. AP: Activates and Precharges a page. This mode is used if a closed page is accessed, and the next atom needs different page. A close-page policy *always* uses this mode.
2. ANP: contains an Activate, but No-Precharge. A transition from AP to this mode is made *while* the schedule is being executed, if the next access is a page hit and it is detected in time.
3. NAP: No-Activate, and Precharge. This mode is used if the current atom was a hit, but the next atom is not known to be a hit.
4. NANP: No-Activate, No-Precharge. A mode that contains only RD or WR commands. A transition from NAP to this mode is made *while* the schedule is being executed, if the previous and next atom are both page hits.

Figure 6.3 shows the relation between the modes graphically. A transition to a dotted mode can be regarded as the memory controller equivalent of a conditional jump instruction to a different schedule, based on the detection of a page hit. The transition is made instantly when the transition condition is satisfied. For example, if the nth command from a pattern is executed in AP mode and a hit is detected in cycle x, then the $n + 1$th command of the same type of pattern in ANP mode is executed in cycle $x + 1$.

The memory controller has to inspect the address of the next atom that is selected by the resource arbiter to detect hits. This address has to be known *before the first precharge* would be executed in the AP or NAP schedules. If the address for the next access is not known by that time, then the controller has to assume a miss to prevent sacrificing worst-case guarantees. This implies that there is a limited time-window in which locality can be exploited. In a naive implementation, based on a close-page schedule with auto-precharge commands, the size of this window depends on the time required to activate a row, plus the time required to access all bursts from the first bank in the access. The greater the BC is, the more time exists between the start of an access and the decision moment. Section 6.3 describes a method to increase the size of the window by exchanging auto-precharge commands for explicit precharges that are scheduled as late as possible.

Similarly, there also exists an address window in which locality can be exploited, the size of which depends on BI. Each bank has its own row buffer, so the number of bits that is activated by a pattern is BI times the size of the row buffer per bank. The number of distinct atoms that might be a hit is equal to the total number of activated bits, divided by the access granularity, which for a constant access granularity grows linearly with BI.

6.2 Impact on Pattern-Based Controller

The conservative open-page policy impacts both the memory patterns and the supporting architecture of our controller. At first sight, the introduction of different modes for each of the patterns leads to four-fold expansion of the number of patterns that need to be stored. Fortunately, the differences between patterns in each mode are small by definition: to allow seamless transitions from AP to ANP, and from NAP to NANP, the respective schedule-pairs have to be the same at least until the precharge decision is made. After the decision-point, they differ in terms of length, and in whether or not they execute precharge commands. The differences compared to the template architecture in Chap. 2 (Fig. 2.8) are limited. We take advantage of this in the pattern memory encoding, limiting the required space at the cost of a slightly more complex controller.

Two versions of each original pattern are stored in the pattern memory instead of one. The first version contains commands shared by AP and ANP, while the second version is shared by NAP and NANP. Figure 6.4 shows what the pattern memory and its entry and exit points look like when the conservative open-page policy is used. When deciding on which pattern to execute next, the pattern selector indexes an entry in the pattern LUT. In addition to the incoming atom type and the previously executed pattern type, it now also needs to consider the mode in which the *previous* pattern was executed (the LUT gets more entries). For misses, the previous mode was either AP or NAP. In that case, it selects the entry containing a base address in the

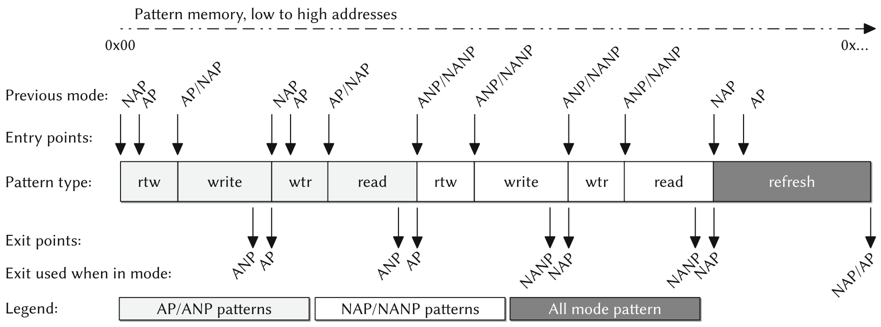


Fig. 6.4 Mapping of patterns to the pattern memory

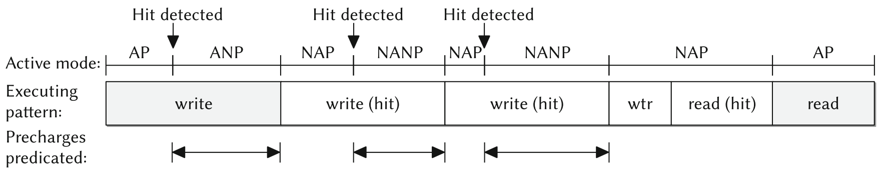


Fig. 6.5 Example of the relation between modes, executed patterns and the predication of precharge commands. Detected hits only change the mode if they are detected before the time-window closes

(gray) AP/ANP range for the next atom. For hits, the previous mode was either ANP or NANP, which means the next atom needs a pattern from the (white) NAP/NANP base address range.

If a hit is detected during the execution of a pattern and before the first precharge, then its active mode changes. The command player is notified, such that it can update the pattern's exit point to reflect the length of the pattern in the appropriate (NP) mode. The possible exit points per active mode are shown in Fig. 6.4. The execution of precharge commands, either explicitly (Sect. 6.3) or in the form of auto-precharge flags, is predicated by the active mode in which the pattern is executed: in ANP or NANP mode, they are not forwarded to the SDRAM. Figure 6.5 shows the transitions between modes for an example with three write and two read atoms. The example starts off with all banks closed. A write atom arrives, and is serviced by the gray write pattern in AP mode from Fig. 6.4. While it is executing, but before the first precharge, the next write atom arrives. It is a hit, so the active mode changes to ANP. The exit point changes correspondingly to ANP, allowing for an earlier termination of the pattern, as shown in Fig. 6.4. Since the mode at the end of the first write was ANP, the second write pattern uses the ANP/NANP entry point in the white pattern range.

The refresh pattern always follows a mode that precharges (AP or NAP). The number of required leading NOPs before the REF command may vary depending on the preceding mode, and hence the entry point changes based on it.

Figures 6.2 and 6.4 suggest that patterns in modes containing no precharges are shorter than those that do, but this is not necessarily the case, as shown in Fig. 6.6. Read and write patterns in ANP or NANP mode have to contain trailing NOPs to resolve the RD-to-RD or WR-to-WR (data bus) constraints across patterns. In AP or ANP mode, these constraints overlap with the start of the following patterns, which generally contain no additional data bus activity, and hence no NOPs are required. Note that for the presented storage scheme and pattern transition mechanism, it does not matter which of the modes contains the longest pattern, as long as the entry and exit points in the pattern LUT are configured accordingly.

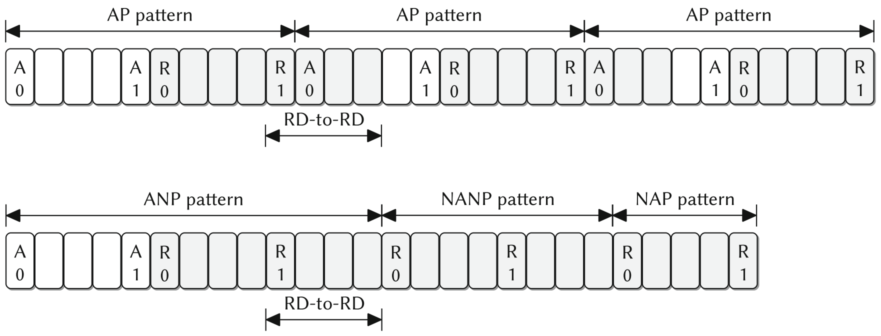


Fig. 6.6 Example where an ANP pattern is longer than an AP pattern. Note that each individual read burst still completes at the same time or earlier when the NP patterns are used

6.3 Using Explicit Precharge Commands

The conservative open-page patterns have a limited time window in which they can transition to a NP mode. The time-window size has to be as large as possible to maximize the exploited locality. To maximize the window size, the *decision to precharge* must be made as late as possible. For this purpose, we propose to *replace the auto-precharge flags with explicit precharges* that happen later in the schedule, effectively postponing the decision. To maintain the same worst-case guarantees, we do not allow the read and write schedule lengths to increase as a result of the replacement. This section presents a greedy heuristic that generates schedules that use this principle for our pattern-based memory controller, based on the existing patterns that use auto-precharges. After applying the heuristic, the size of the time-window is larger than or equal to the original window size, with no influence on the read or write pattern length. Therefore, it is always recommended to apply this heuristic when using the policy.

Algorithm 5 shows the heuristic, reusing the semantics and some of the functions that were used earlier to describe Algorithm 2 in Chap. 3. The top-level function in the heuristic is INCREASEWINDOWSIZE. As its inputs, it needs the read or write pattern for which the time-window should be extended (**P**), its length including trailing NOPs (pattLen), and the pattern that should be schedulable after this pattern (**nextP**). The AP and NAP pattern should be processed separately, and hence **P** can be either of these patterns. **nextP** can either be an AP or ANP pattern (it has to start with an activate). In practice, running the heuristic with either of these patterns as **nextP** yields the same result, since they both contain the same commands except for the precharges, (as can be seen in Fig. 6.2), and no PRE-to-PRE-command constraints exist for the considered SDRAM types.

Algorithm 5 Heuristic to increase the time window size

```
1: function IncreaseWindowSize(pattLen, nextP, P)
2:     while true do
3:         P' := P // A copy, to restore the pattern if this iteration fails
4:         firstPre := getFirstPre(P) // Returned by reference
5:         if firstPre.autoPrechargeFlag == true then
6:             startAt := earliest((type: PRE, bank: firstPre.bank, cc: 0), P)
7:             preCc := firstFreeCycle(startAt, P)
8:             P := P $\cup$ { (type: PRE, bank: firstPre.bank, cc: preCc) }
9:             firstPre.autoPrechargeFlag := false
10:        else
11:            preCc := firstFreeCycle(firstPre.cc + 1, P)
12:            firstPre.cc := preCc
13:        if preCc >= pattLen or minPatternDistance(pattLen, nextP, P) $> 0$ then
14:            return P'
15: function firstFreeCycle(startAt, P)
16:     // Determine the first free cycle in P starting at and including startAt.
17:     while { cmd $\in$ P | cmd.cc == startAt } $\neq \varnothing$ do
18:         startAt := startAt + 1
19:     return startAt
```

The heuristics iteratively increases the size of the window, considering one precharge at a time. The first auto or explicit precharge in the schedule is greedily selected as a conversion or move candidate, since it is the critical command that determines the window size. The getFirstPre function returns a reference to this command (line 4). Note that it is always possible to uniquely identify the first precharge, since there is only one command per cycle, which could either be a RD or WR with an auto-precharge flag, or an explicit precharge.

In each iteration, the heuristic can either (1) *convert* an auto-precharge into an explicit precharge (lines 5–9), or (2) *move* an earlier converted explicit precharge command to a later cycle in the schedule (lines 10–12). The initial conversion results in a relatively large jump of the precharge decision for a bank, since it needs to satisfy the RD/WR-to-PRE constraint (Table 3.2). We attempt to move explicit precharge commands one cycle per iteration of the heuristic.

Adding or moving a PRE command is only possible if the cycle we try to place it in is not already taken, which is ensured by the firstFreeCycle function. If a cycle is already occupied, then this function will consider all following cycles, until it finds an empty one. We do not consider the option of moving other commands (ACTs, RDs or WRs) to make room for the PRE command, since this would require reevaluation of all other constraints related to these commands, and the result is likely to affect the pattern length and worst-case performance negatively.

Lines 13 and 14 evaluate the result of a conversion or move. The minPatternDistance function, defined earlier in Algorithm 1 (Sect. 3.2.1), is used to determine the minimal distance between **nextP** and **P**. If the modified precharge command is not scheduled *within* the pattern, or if the next pattern cannot be scheduled *immediately* after the current pattern anymore as a result of the modification, then the heuristic

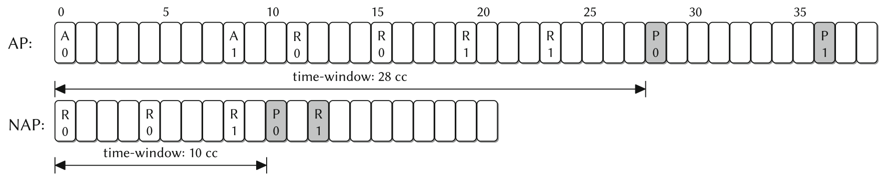


Fig. 6.7 Resulting patterns after converting auto-precharges to explicit precharges (DDR3-1600, (2,2))

terminates returning $\mathbf{P}'$, the last successfully modified copy of the pattern. In all other cases, we select a new candidate precharge (line 4), and repeat the procedure.

Algorithm 5 is greedy in its selection of which precharge to move first. It might place this precharge in a cycle that makes future moves of other precharges impossible, terminating at a non-optimal solution, in the sense that a larger window would have been possible if another order was chosen. In Sect. 6.4.1, we bound the difference between the heuristic's output and the optimal window size.

Figure 6.7 shows the results of the heuristic when applied to the example from Fig. 6.2. A more extensive evaluation of the heuristic can be found in Sect. 6.4.1. The precharge decision is postponed by 14 and 6 cycles in the AP and NAP modes, respectively. Another way to put this is to say the window size increased by 100 and 150 %, although that probably paints a more dramatic picture than warranted by the absolute numbers. The windows in NAP patterns are relatively small compared to the AP pattern, since they immediately start with RD or WR commands, leaving out the initial ACT-to-RD/WR cycles that add to the window size in an AP pattern.

Note that a time-window of 10 cycles does not imply that a client's atoms have to arrive at 10 cycle intervals to successfully exploit locality. Instead, it means that the arbiter in the resource bus has 10 *additional* cycles before it needs to settle on the next atom to send to the back-end, while the atoms themselves are can already be queued up in the client's atom queue in the front-end.

The heuristic ensures that the sizes of the read and write patterns do not increase as a result of moving the precharge commands, but it has no guards preventing the auxiliary patterns from growing. Switching patterns are not impacted by the location of PRE commands, since there are no PRE-to-RD/WR constraints (see Tables 3.1 and 3.2). Refresh commands, however, need to respect a non-zero PRE-to-REF constraint, and their patterns may thus be impacted. There are multiple ways to approach this issue

1. The refresh pattern can be regenerated based on the modified read and write patterns. After evaluating the impact on the worst-case guarantees, a decision to accept the change or to use option 6.2 or 6.3 can be made.
2. A fairly trivial modification to line 13 of the heuristic, adding the refresh pattern as an additional MINPATTERNDISTANCE check, could terminate it once the refresh pattern starts to be affected.
3. Manual refresh schemes like shown in [1, 2] can avoid refresh patterns and the associated issue completely.

6.4 Evaluation

The evaluation of the conservative open-page policy is done in two steps. First, we generate conservative open-page patterns for our test memories, apply Algorithm 5, and discuss its effectiveness in Sect. 6.4.1. Second, Sect. 6.4.2 shows the average-case performance improvement the policy offers in various scenarios.

6.4.1 Time-Window Size

Table 6.1 focuses on the 16-bit DDR3-1600 device from before, containing results for pattern configurations up to an access granularity of 64 bytes. Read and write pattern related numbers are shown on separate lines, in both the AP and NAP mode. For each configuration, it shows three columns per pattern:

1. *PS*, the pattern size.
2. *WS*, representing the time-window size after applying Algorithm 5.
3. *A*, representing the number of cycles added to the time-window by Algorithm 5 relative to the naive solution that only uses auto-precharge flags.

Postponing the precharge-decision by increasing BC increases the size of the time-window as predicted earlier in Sect. 6.1, and hence the largest window sizes at a specific access granularity are found at the largest BC value. The table also shows Algorithm 5 usually increases the window size by 50 % in this result set. The time-window in a write pattern is always at least as large as in the corresponding read pattern, because the WR-to-PRE constraints are greater than the RD-to-PRE constraints for this memory (and for all other memories in Appendix B as well). This causes write patterns in general to be larger than read patterns, and hence there are

Table 6.1 Time-window sizes using the conservative open-page policy and the number of cycles contributed by the heuristic for the schedules containing precharges (DDR3-1600)

BI	1			1			2			1			2			4		
BC	1			2			1			4			2			1		
AG (bytes)	16			32			32			64			64			64		
	PS	WS	A	PS	WS	A	PS	WS	A	PS	WS	A	PS	WS	A	PS	WS	A
AP-RD [cc]	39	28	17	39	28	13	39	28	17	40	29	6	39	28	13	40	23	12
AP-WR [cc]	46	35	24	50	39	24	46	35	24	58	47	24	50	39	24	46	23	12
NAP-RD [cc]	24	13	13	21	10	6	18	7	7	29	18	6	21	10	6	17	6	6
NAP-WR [cc]	35	24	24	39	28	24	35	24	24	47	36	24	39	28	24	35	12	12

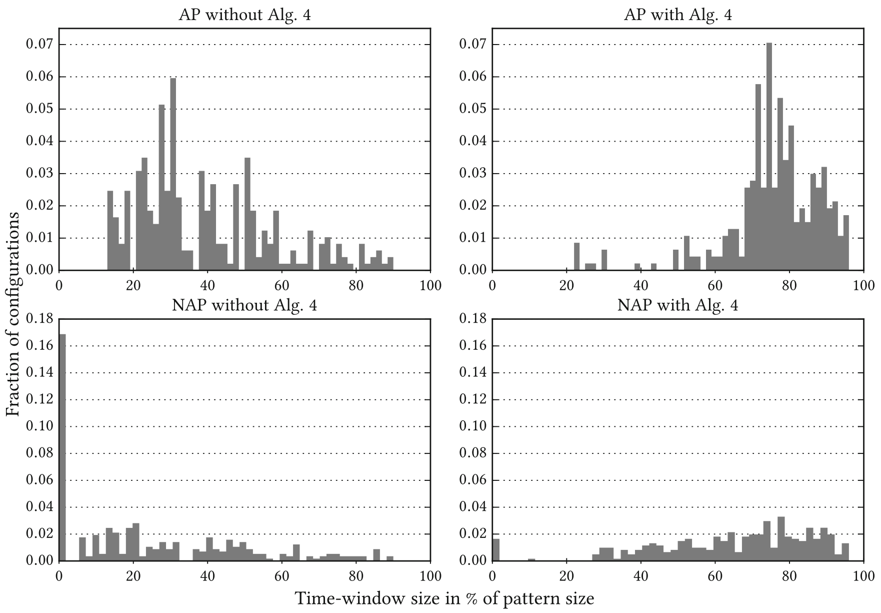


Fig. 6.8 Relative time-window size histogram (the height of a bar represents the fraction of patterns in the result set having a time-window in the bin corresponding to the value on the x-axis)

more potential locations for explicit PRE commands, while at the same time they are forced to be scheduled relatively late due to the same constraint. NAP patterns have smaller time windows than AP patterns in the same configuration, since they are simply smaller (which is the whole point of creating them), and immediately start with read or write bursts. The accessed banks in NAP patterns can typically be precharged earlier, because the ACT-to-PRE constraints are (partially) resolved during the preceding ANP pattern.

Next, the scope is expanded to configurations up to an access granularity of 256 bytes, and we also include all the other test memories (Appendix B). In this result set (visualized in Fig. 6.8), the time-window spans on average 39 % of the AP pattern and 35 % of the NAP pattern before applying Algorithm 5, versus 76 and 65 % after, respectively. We also observe that none of the refresh patterns are affected by the moved precharge commands, so we do not have to select a method to deal with this.

Figure 6.8 contains histograms of the relative time-window size with respect to the pattern size for the same result set, both before (left graphs) and after (right graphs) Algorithm 5 is used, for patterns in AP (top graphs) and NAP (bottom graphs) mode. The y-axis is normalized and represents the fraction of patterns that fit in a certain bin (there are 316 patterns and 50 bins in each graph). The graphs shows that the heuristic shifts the distribution significantly to the right, in line with the growth of the average window size we saw earlier.

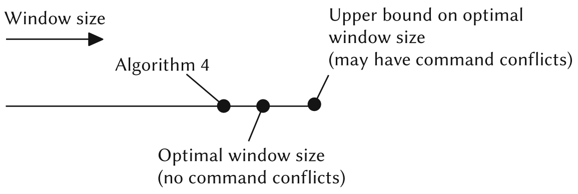

Window size
Upper bound on optimal window size (may have command conflicts)
Algorithm 4
Optimal window size (no command conflicts)

Fig. 6.9 We use an upper bound on the optimal window size to determine how far Algorithm 5 can maximally be from the real optimum

To bound the *difference* between the heuristic output and the global optimum, we sabotage the FIRSTFREECYCLE function such that it regards cycles containing explicit precharge commands as empty. As a result, multiple precharge commands are allowed to happen within the same cycle, which eliminates the effect of the greediness of the heuristic, although it does potentially introduce command conflicts. The window size produced by this modified heuristic is an upper bound for the optimal window size (Fig. 6.9).

We apply the modified heuristic to our set of test memories (Appendix B) and take note of the differences in window sizes compared to our previous result. The results show that Algorithm 5 generates the optimal result for at least 90 % of the tested patterns. On average, the optimal window size is not more than 1 % larger than the size produced by the heuristic. The largest percentual difference from the upper bound in a single configuration is 29 %, although a manual (not generally applicable) inspection shows that in this case the bound is not tight, and Algorithm 5 actually produced the optimal window size. Given these results, we conclude that the heuristic is effective in achieving its goal.

6.4.2 Stall Time Reduction

The conservative open-page policy theoretically improves the average-case performance of memory clients. To evaluate this claim, we set up an experiment with the SystemC model of our memory controller, connected to a model of the DDR3-1600 SDRAM (Appendix B). We compare the performance of a set of memory traces, first using a close-page policy, and later using the conservative open-page policy. The corresponding applications are drawn from the CHStone benchmark set [3], and traces are generated using the SimpleScalar 3.0 processor simulator [4].

The selection of a benchmark set is often a compromise between (arguably) conflicting requirements, this experiment being no exception. Code and the associated input sets need to be available, the effort involved in porting and compilation should be limited, the mix of applications has to be relevant with respect to the application area of the research, run-time on the target architecture needs to be reasonable, and the functional correctness of the applications should be maintained, while the overarching goal is to prove a point based on the results of the experiments. CHStone meets most of these requirements, as it comes with integrated input data and reference output, while being mostly compatible with the target processor. Eight out of

Table 6.2 CHStone trace characteristics

Trace	ADPCM	AES	BF	GSM	JPEG	MIPS	MOTION	SHA
Avg. bandwidth (MB/s)	846	878	253	1910	100	1577	2426	236
Number of requests	645	742	873	644	1685	541	617	791

the twelve applications in the benchmark are used, the four applications that are left out use 64-bit floating-point operations that are not supported by the SimpleScalar compiler. What remains is the set shown in Table 6.2. It consists of various audio and image coding/decoding applications (ADPCM, GSM, JPEG, MOTION), cryptographic algorithms (AES, BF, SHA) and a small MIPS simulator, all of which (except for perhaps the MIPS simulator) could plausibly run on a mixed time-criticality embedded system.

For each application in the benchmark, a memory-trace file is generated using a slightly modified version of the SimpleScalar simulator. It records the time and address of each L2 cache miss, resulting in a trace file containing all requests that go to the SDRAM. We use the out-of-order execution engine (sim-outorder) with default settings except for the cache setup, for which we select half the size compared to its defaults. We use a unified 128 KB L2 cache with 64-byte cache lines, 512 sets and an associativity of 4. Each request in the trace thus corresponds to a cache miss of 64 bytes. There are two reasons to reduce the cache size: (1) the traces are later used to model traffic in a 4 core system with partitioned L2 caches, and hence we limit the total cache size to more realistic proportions for an embedded system. (2) The smaller cache size increases the load on the SDRAM, and biases the applications' performance toward being more memory bound. Applications that are not influenced by the memory performance, do not benefit from improvements made on the SDRAM side, and would generate valid but trivial results, and hence we try to avoid them. The reader should keep this in mind when evaluating the results of the experiment.

The memory controller front-end is included such that the SDRAM can be shared amongst multiple clients. Its arbiter configuration only matters in the multi-application experiment Sect. 6.4.2.3, and will be introduced there. We use a setup with four input ports, each of which is connected to a trace-based traffic player by means of a (composable) NoC [5], as shown in Fig. 6.10. A traffic player generates requests at the times (clock cycles) indicated by the trace file assigned to it. Each player emulates a processor running at 1400 MHz, which means that a clock cycle in the trace corresponds to 0.71 ns. Table 6.2 shows the resulting average requested bandwidth, i.e., the total amount of data requested in the trace, divided by the time of the last request. Note that the actual traffic intensity varies over time during the trace execution. Each application has its own memory range, such that they do not share rows in the memory. The controller is configured with pattern sets that offers a 64-byte granularity (atom size) to match the request size of the traces.

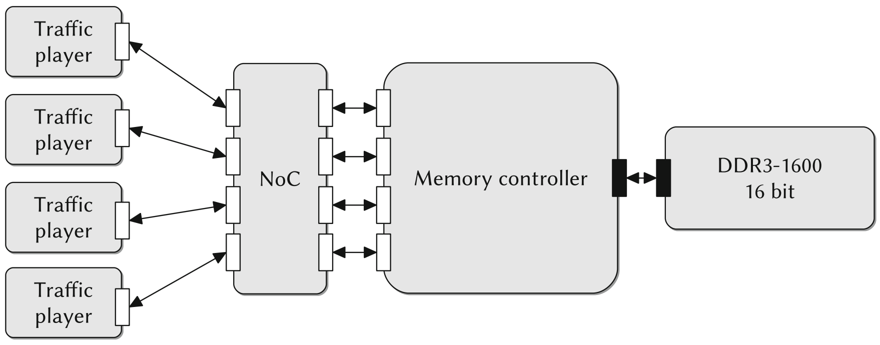


Fig. 6.10 Setup of the conservative open-page experiments

A traffic player allows a maximum of four outstanding read-requests before it stalls. When this happens, it stops executing, i.e., halts its cycle counter, and does not issue any more requests until a response arrives. It further assumes that all requests are independent. Note that a real processor could potentially stall due to dependencies, but that it is not uncommon for multiple cache misses to arrive at a memory controller in a relatively short interval [6], and that techniques exist to improve the available memory parallelism [7, 8]. Support for multiple outstanding requests is a feature found in the higher-end embedded processors, for example, the PowerPC e500v2 [9] supports five outstanding load misses.

Three factors determine the number of page hits for an application. In the following sections, the consequences of each of these factors are evaluated for the proposed policy.

1. *Spatial locality* has to be present within an application (Sect. 6.4.2.1). If a request targets the same row and bank as its predecessor, then it is a potential hit, otherwise it is a guaranteed miss.
2. *Temporal locality* (Sect. 6.4.2.2): a request containing spatial locality has to arrive at the memory controller before the time window closes.
3. *Interference* from requests by other clients (Sect. 6.4.2.3). Requests streams from different clients might be interleaved, destroying their locality.

6.4.2.1 Spatial Locality

The spatial locality of a trace, i.e., the fraction of consecutive requests that target the same page, can be determined by analyzing the sequence of accessed addresses. Which bits from the address are used to determine the bank and row addresses depends on the address decoder configuration (Sect. 3.2.5), which is a result of the selected pattern configuration. The likelihood of two consecutive requests targeting the same banks increases with a growing BI, as discussed earlier in Sect. 6.1, resulting in more hits.

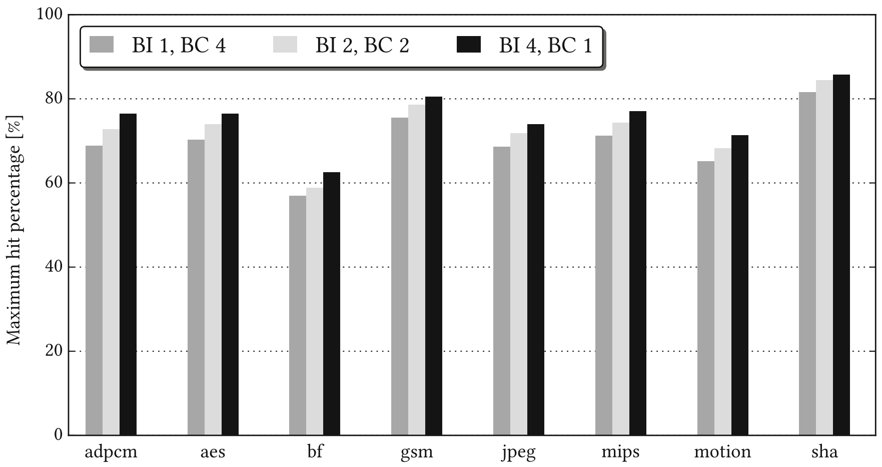


Fig. 6.11 Available spatial locality per trace for three pattern configurations, from left to right: (BI 1, BC 4), (BI 2, BC 2) and (BI 4, BC 1)

We analyze the traces for the three different pattern configurations that offer a 64-byte access granularity. Figure 6.11 shows the spatial locality per trace. At least 57 % of the requests in each trace can potentially benefit from an open-page policy, with a variation of at most 8 % points caused by the different configurations.

6.4.2.2 Single-Application Performance

In the single-application experiment, only one of the four traffic players is active, running each of the application traces independently. This implies that both spatial and temporal locality play a role, but interference is still left out of the equation.

We run each trace two times, first using a close-page policy and then using the conservative open-page policy in the BI 2, BC 2 configuration. The results of these experiments are shown in Fig. 6.12, through 4 bars per experiment, which we now discuss one by one.

First, we determine the fraction of requests containing spatial locality (identified in Fig. 6.11) that is captured within the time-window by simply counting the number of hits in the memory controller. This is plotted on the first bar in the graph as a percentage of the maximum number of hits identified in Fig. 6.11. 70 % of the requests which contained spatial locality actually result in a hit on average.

The execution time of a trace consists of two parts, (1) the time spent on *computation*, which is memory performance independent, and (2) the time spent *stalled* waiting for a response from the SDRAM. Optimizations on the memory side can only reduce the stall time. The percentage by which the stall time is reduced by the conservative open-page policy is plotted on the second bar in Fig. 6.12. Results vary between 58 % (BF) and 86 % (SHA) reduction. These numbers may appear relatively

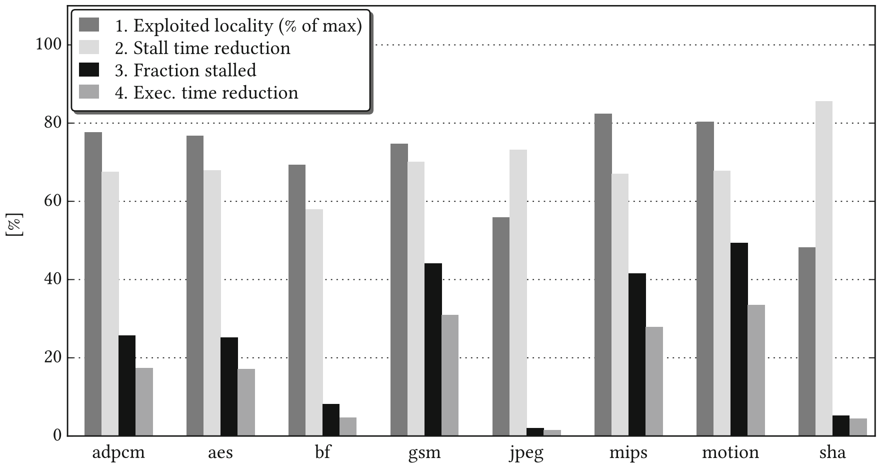


Fig. 6.12 Single-application experiment results. Bar 1 represents the exploited locality during the conservative open-page run, bars 2 and 4 are relative numbers given the close-page and conservative open-page runs, and 3 is the fraction of time the traffic generator was stalled during the close-page run. All runs use (BI 2, BC 2)

high at first, considering that in the best case, the conservative open-page policy replaces an AP pattern by an NANP pattern compared to the close-page policy, resulting in "only" a 59 % time saving (see Fig. 6.2). However, the traffic generator only stalls once it reaches its maximum outstanding request limit, and hence there is not a one-to-one translation of memory response time to stall time. A small reduction of the memory response time can thus reduce the stall time by a relatively larger factor.

Bar three in Fig. 6.12 shows the ratio between the stall time and the computation time for the trace while using the close-page policy. The higher this ratio is, the more the execution time of an trace will reduce as a result of a stall-time reduction. This number is directly correlated to the average requested bandwidth for a trace, previously shown in Table 6.2. Based on this statistic, MOTION is expected to receive most benefit, while JPEG benefits the least.

Next, we look at the execution-time difference, shown in the fourth bar. Here, results vary wildly based on the used application: JPEG's execution time is reduced by only 1 %, while that of MOTION is reduced by 33 %, as expected based on the previously discussed ratio. The average execution-time reduction across all applications is 17 %, so we conclude that our proposed technique works well given the stall-time reduction, and that the benefit for an application scales with how memory intensive it is.

Table 6.3 Pattern configuration influence on single application performance when using the conservative open-page policy

Banks interleaving (BI)	1	2	4
Burst count (BC)	4	2	1
Worst-case bandwidth (b_{wc}) (MB/s)	901	1050	1144
Average exploited locality (%)	78.7	70.6	70.1
Average exec. time reduction (%)	16.1	17.2	17.0

Pattern Configuration Influence

The pattern configuration has a large impact on the worst-case guarantees, as discussed in Chap. 5. To quantify its interaction with the conservative open-page policy, we repeat the single-application experiment for different configurations.

The considered configurations in this experiment all have a granularity of 64 bytes: interleaving over either 1, 2 or 4 banks, while doing 4, 2 or 1 burst per bank, respectively. Table 6.3 shows the worst-case bandwidth delivered by those configurations, calculated as described in Sect. 2.4.2.2. It increases with BI, with a 21 % difference between BI 1 and BI 4. Figure 6.11 shows a trend in the same direction; the higher BI, the higher the spatial locality. A trend in the opposite direction is visible for the window size (see Table 6.1). This leads to the observation that for an increasing BI, both worst-case bandwidth and spatial locality increase, but the size of the time-window decreases.

Table 6.3 also shows the (measured) average fraction of exploited locality and the average execution time for the eight benchmark applications. The conservative open-page policy captures the largest fraction of potential locality in the configuration with the largest window-size (BI 1, BC 4). However, the execution-time reduction is largest for the (BI 2, BC 2) configuration. In absolute numbers, the average execution time for (BI 2, BC 2) is only 0.3 % smaller than that of (BI 1, BC 4). We conclude that *the (open-page related) performance differences across configurations are so small that they are insignificant, so the selection of a configuration can be made based on the real-time guarantees (and power consumption) that it offers, without significantly impacting the effectiveness of the conservative open-page policy.*

6.4.2.3 Multi-application Performance

To show the effect of multi-application interference on locality exploitation, we run an experiment with four simultaneously active applications on four separate traffic players. Two high-load and two low-load applications (MIPS, MOTION, JPEG and BF) compete for the memory resource. *Work-conserving TDM* is used as the arbitration scheme in the front-end, which means that unclaimed slots from one application can be used by another application. If a slot is not used by its designated owner, then the arbiter simply selects the next application in the table that has a request available. A

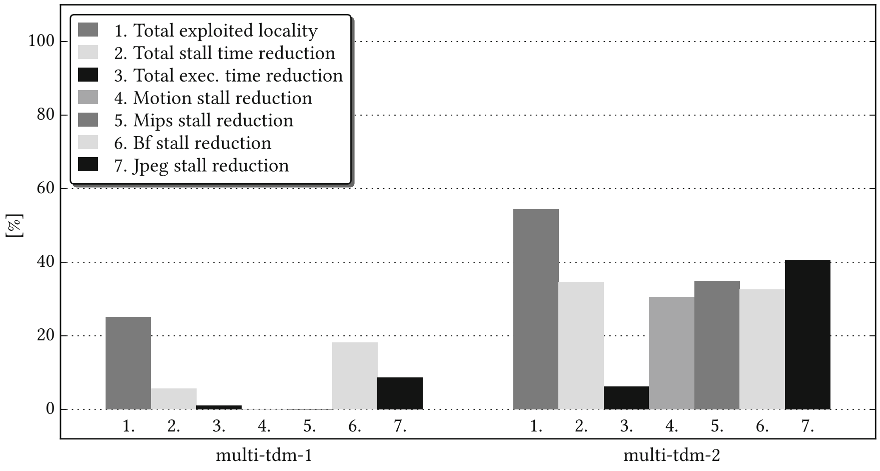


Fig. 6.13 Multi-application experiment results

(BI 2, BC 2) configuration is used. Two variations of this experiment with different arbiter settings are performed, annotated with MULTI-TDM-1 and MULTI-TDM-2.

In the MULTI-TDM-1 experiment, each application is allocated one out of four slots in the TDM table (effectively creating a round-robin arbiter). This means each application can get at least a quarter of the memory bandwidth. The downside of this scheme is that it allows for fine-grained interleaving of requests from *different* applications, which destroys locality that was present in the original memory trace. This effect is visible in Fig. 6.13: only 25 % of the potential locality is captured (bar 1), which is significantly lower than the average captured locality in the single application case. The reduction of the sum of the stall cycles of all applications when the conservative open-page policy is switched on, shown in the second bar, is 6 %, and consequently the total execution-time reduction (third bar) is also small.

Looking at the stall-time reduction for the individual applications (bar 4–7), MOTION and MIPS hardly benefit at all, with MIPS even showing a small increase of its stall time. An explanation for this is that both of these traces are short with respect to the other two, and require a relatively large amount of bandwidth (Table 6.2). As a result, they interfere with *all* other applications and with each other during the short period where they are active, almost nullifying the benefits of the conservative open-page policy. BF and JPEG run significantly longer, and produce fewer requests while doing so, which means they rarely actually try to use the memory at the same time, and hence manage to obtain some benefit from the policy once MOTION and MIPS are done.

To retain more of the locality in the request stream, the arbiter in the MULTI-TDM-2 experiment is modified: each application gets two consecutive slots in a TDM table of eight slots in total. Note that this has implications for the worst-case response time; in the first TDM-schedule there were at most three interfering slots for an application, while in the second there are at most six interfering slots, and hence Θ_{arb} grows.

Figure 6.13 shows that giving each application two consecutive slots has a large impact on the fraction of exploited locality: 54 % of it is captured, more than 2 times as much as in the MULTI-TDM-1 experiment, resulting in a total stall-time reduction of 35 %. The individual stall time drop is between 31 % (MOTION) and 40 % (JPEG). We can conclude that *successfully applying the conservative open-page policy in a multi-application use-case is possible, under the condition that the arbiter allows at least part of the consecutive requests from an application to be scheduled consecutively, potentially at the cost of a larger WCRT, but equal throughput.*

MRT Performance

The previous experiments used a setup in which the tested applications themselves had an inherent degree of memory parallelism: each could issue 4 requests before stalling. In this experiment, we demonstrate that *all* applications benefit from the conservative open-page policy even if they do *not all* have this property, based on the notion that the policy improves the *overall* memory performance by exploiting locality. This is a relevant scenario, especially in a mixed-time criticality context, where real-time streaming applications co-run with best-effort applications.

Two traffic players are active in the experiment: the first one runs one of the benchmark applications and is configured to block immediately when a request is issued, and unblock once the response arrives. As a result, the application cannot benefit from any of its own locality. This models what happens in case an in-order processor is used to execute the application. The second traffic player generates a synthetic traffic stream that models a (fictional) real-time video IP. It requires a bandwidth of 270 MB/s, which is the combined read and write rate required to transport 60 frames per second of $1024 \cdot 786$ pixels from and to the memory, assuming 3 bytes/pixel. We assume all of its requests are independent, such that a high degree of locality is available in this stream, i.e., the IP running the application is fully pipelined.

16 runs are performed in total: two for each benchmark applications, first with the close-page policy, and later with the conservative open-page policy. The objective of this experiment is twofold: (1) it allows us to experimentally verify that the real-time bandwidth constraint of the video IP is always satisfied, regardless of the used page policy, and (2) it quantifies the impact of the locality exploitation by the pipelined video application on the execution time of the non-pipelined application.

Compared to a close-page policy, the average execution time of the benchmark applications is reduced by 7.9 % when using the conservative open-page, while still satisfying the constraints of the video IP. Using the policy, the controller manages to serve the video application faster, allowing more time to be spent on the benchmark application which hence benefits indirectly. The MOTION benchmark again gains most in terms of execution-time reduction (13.4 %), while JPEG shows the smallest improvement (1.9 %). Based on these results, we conclude that *if there is at least one application that exploits locality, then all the other applications that share the memory can benefit and the overall average-case performance increases.*

6.5 Conclusion

This chapter deals with the problem of mixed time-criticality workloads for SDRAM controllers. Existing controllers typically optimize for either worst-case or average-case performance, but not for the combination of the two. We proposed a conservative open-page policy that improves the average-case performance without sacrificing real-time guarantees. It exploits a portion of the locality in the request stream, reducing the average-case response time. We showed how existing close-page patterns can be converted to their conservative open-page counterparts, and presented an algorithm that replaces auto-precharge flags by explicit precharge commands, which improves the effectiveness of the policy.

The stall-time and execution-time reduction in single- and multi-application use-cases are quantified for a set of benchmark traces. The average-case performance is improved by the conservative open-page policy in both of these scenarios. Interference between applications largely determines the degree to which locality is successfully exploited. Arbiter configurations that encourage requests of the same application to be scheduled consecutively by the back-end are significantly more effective than those that do not. It therefore makes sense to try and use those if the (unavoidable) impact on the worst-case performance can be tolerated. Fortunately, the average-case performance benefits obtained by using the policy is not controller configuration, and hence BI and BC can be selected based on their effect on the worst-case properties of the memory only. Finally, we showed that as long as least one of the memory clients benefits from the policy, the overall memory performance for all clients improves.

References

1. Bhat B, Mueller F (2011) Making DRAM refresh predictable. Real-Time Syst 47(5):430–453
2. Reineke J, Liu I, Patel HD, Kim S, Lee EA (2011) PRET DRAM controller: bank privatization for predictability and temporal isolation. In: Proceedings of the CODES+ISSS, pp 99–108
3. Hara Y, Tomiyama H, Honda S, Takada H (2009) Proposal and quantitative analysis of the chstone benchmark program suite for practical C-based high-level synthesis. J Inf Process 17:242–254
4. Austin T, Larson E, Ernst D (2002) SimpleScalar: an infrastructure for computer system modeling. Computer 35(2):59–67
5. Goossens K, Hansson A (2010) The Aethereal network on chip after ten years: goals, evolution, lessons, and future. In: Design automation conference (DAC)
6. Zhu Z, Zhang Z (2005) A performance comparison of DRAM memory system optimizations for SMT processors. In: 11th International symposium on high-performance computer architecture, 2005. HPCA-11, pp 213–224
7. Pai V, Adve S (1999) Code transformations to improve memory parallelism. In: International symposium on micro-architecture, pp 147–155
8. Ding W, Guttman D, Kandemir M (2014) Compiler support for optimizing memory bank-level parallelism. In: International symposium on microarchitecture (MICRO), pp 571–582
9. PowerPC e500 Core Family Reference Manual Reference Manual (2005) Freescale Semiconductor

Chapter 7
Reconfiguration

This chapter deals with the topic of *reconfiguration*. It has been alluded to in several places already, most prominently in Chap. 2, where the reconfiguration infrastructure was shown from an architectural point of view. The focus in this current chapter lies more on the process of memory controller reconfiguration, its merits and limitations in the context of a predictable and composable SoC, and the effects it has on the associated performance guarantees.

So far, the controller runs we have shown used the same configuration from boot time until finish, merely using the reconfiguration infrastructure for initialization. The value of this software-based programmability is the flexibility it offers: the same hardware may be deployed with multiple different configurations. This enables customization of its behavior to (1) the memory device, (2) the power/performance trade-off provided by the patterns, and (3) different sets of clients. However, systems generally operate in dynamic environments where the mix of active applications or *use-case* is not constant during the execution. As a result, the requirements of the controller's clients change, and it is unlikely that a single configuration fits all use-cases perfectly, creating a need for reconfiguration during operation.

This chapter starts with an overview of the reconfiguration options offered by our architecture in Sect. 7.1. Section 7.2 describes how predictable and composable performance guarantees are defined for clients that remain active during reconfiguration, and Sect. 7.3 discusses the implication this has on the reconfiguration options we can safely use for these clients. Reconfiguring an arbiter while retaining predictable performance guarantees is not trivial: Sect. 7.4 shows how to construct and use a TDM arbiter that has this property. Section 7.5 evaluates the contributions in this chapter through experiments with our SystemC model and the VHDL instance of our controller.

7.1 Reconfiguration Options

The controller architecture template from Chap. 2 contains multiple components that are reconfigurable at run time. They are configured at least once, at boot time, to setup the controller for its first use-case, i.e., the initial set of clients. The memory controller's configuration determines how the SDRAM is used (i.e., which commands it executes), and what the memory performance looks like from the viewpoint of a client. Later reconfiguration might be desired as a result of a *use-case switch*: clients may be started or stopped, or the application behind a client might change. This leads to a change in the system state that is typically coordinated by a *resource manager* [1, 2], which could be either part of an operating system running on a processor in the SoC, or a dedicated hardware module, depending on the required flexibility. The requirements that the memory controller has to satisfy in this new state may be different from what they were earlier. A use-case switch leads to reconfiguration if the required configuration changes with the use-case.

The individual reconfigurable components of the controller are connected by dependencies, limiting how they may be configured. Figure 7.1 visualizes these dependencies. A continuous arrow between two nodes means that if the source node changes its value, then the destination node *has* to change accordingly. A dotted arrow means a change in the source node *might* warrant a change in the destination node, although this is not always necessary. The controller front-end contains the following reconfigurable components:

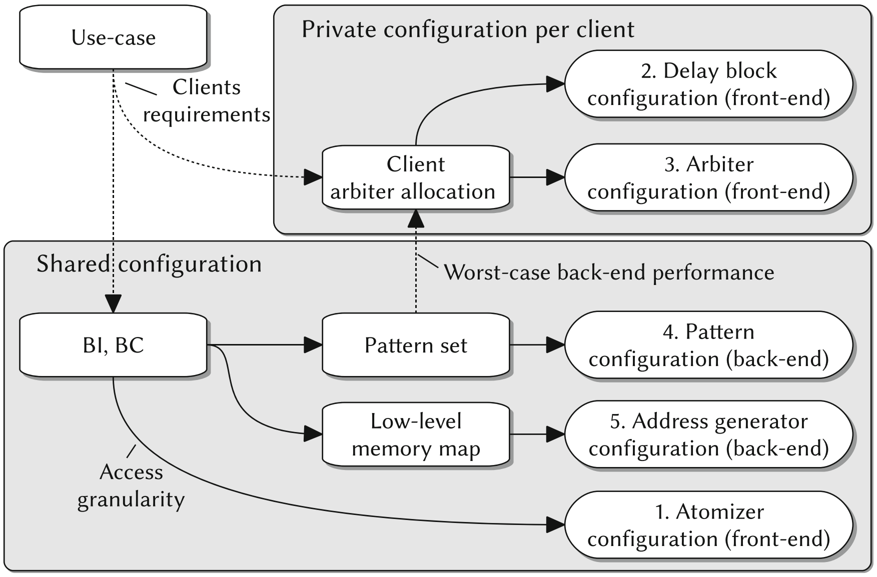

Use-case
Clients requirements
Private configuration per client
2. Delay block configuration (front-end)
Client arbiter allocation
3. Arbiter configuration (front-end)
Shared configuration
Worst-case back-end performance
BI, BC
Pattern set
4. Pattern configuration (back-end)
Low-level memory map
5. Address generator configuration (back-end)
Access granularity
1. Atomizer configuration (front-end)

Fig. 7.1 Overview of reconfigurable components and their interdependencies

1. Atomizers: the atom size should be set equal to access granularity of the back-end.
2. Delay blocks: when composable performance for the client is required, a set of timing registers has to be configured in the delay block, such that can emulate a client's worst-case latency-rate curve (Sect. 2.4.1).
3. Arbiter: the arbiter has to be programmed with the allocated resource budget for an application.

Even though the delay blocks and the arbiter may use different configurations per client, the atomizer may not, since the atom size *has* to match with the back-end. There is hence a distinction between configuration parameters that are *shared* by (and the same for) all clients, and those that are *private*. The back-end solely contains shared configurable components:

4. Patterns: the contents of the pattern memory, the pattern LUT, refresh timer and the pattern set offset need to be configured.
5. Address generator: the masks and shift amounts should be set, such that memory mapping corresponds to the pattern configuration, as discussed in Sect. 3.2.5.

When reconfiguring, two scenarios can be distinguished. (1) In the first scenario, no clients remain active during reconfiguration, and no data has to be retained. This is essentially the same as the initial configuration after reset, and we can hence trivially change all settings in this scenario. (2) In the second scenario, at least one client remains active while the controller is reconfigured. We will call such clients *persistent*.

If we consider changing BI and BC while persistent clients continue to use the controller, then two structural problems appear:

- Modifications of BI and BC that retain a constant access granularity permute the relative burst order before and after reconfiguration, both within and across atoms, as shown in Fig. 7.2. Bursts that were written by a single atom before

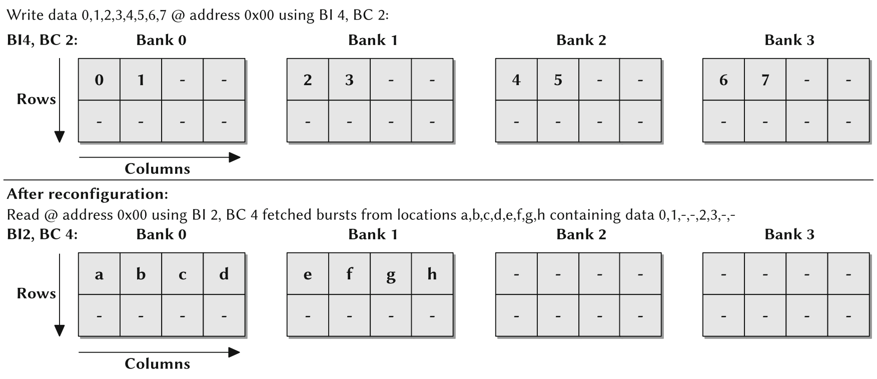

Write data 0,1,2,3,4,5,6,7 @ address 0x00 using BI 4, BC 2:
BI4, BC 2:
Bank 0
Bank 1
Bank 2
Bank 3
Rows
Columns
0
1
2
3
4
5
6
7
After reconfiguration:
Read @ address 0x00 using BI 2, BC 4 fetched bursts from locations a,b,c,d,e,f,g,h containing data 0,1,-,-,2,3,-,-
BI2, BC 4:
a
b
c
d
e
f
g
h

Fig. 7.2 An example of the placement of bursts in the memory using two different pattern sets with the same access granularity. Consecutive bursts have consecutive numbers/characters, and each cell contains a burst. Retrieving the data that was written using (BI 4, BC 2) would require two atoms and reordering when using (BI 2, BC 4)

reconfiguration are spread across multiple atoms after BI and BC are changed.[1] This makes it impossible to (transparently) retrieve the original data for persistent clients.
- Changing the access granularity requires cooperation of the atomizers of persistent clients, since they all have to start using the new atom size before the patterns can actually be changed. This cooperation generally cannot be guaranteed, a client may for example, delay the completion of one of its old-sized atoms by not delivering the required data to its atom buffer, potentially delaying the reconfiguration indefinitely.

These issues make it infeasible to change BI and BC in this scenario, and thus *the entire shared configuration cannot be reconfigured in the presence of persistent applications*, because all its components are dependent on these parameters. However, reconfiguration of the private configuration per client is not hindered by structural issues, and hence, modifying the delay block or arbiter configuration is possible. The next section bounds the extent to which we can use these options, depending on the performance guarantee of the associated client.

7.2 Performance Guarantees During a Use-Case Switch

In this section, we describe how predictable and composable performance for persistent clients is defined, and we outline how the use-case switching process is orchestrated for the different types of clients in the system (Fig. 7.3).

Nonpersistent clients receive no performance guarantees during reconfiguration, because they are switched off before it happens. During a use-case switch, this type of client is handled first. The resource manager stops the flow of requests for these clients at the source side (processor or peripheral). This can be done forcefully, or in cooperation with the source. It then triggers a final dummy-read request on the client's front-end port and waits for the response to ensure that no requests for the client are left in the controller. Alternatively, the client's WCRT bound could be used to determine when the final request that the client made is fully processed. Since requests are never reordered per client, quiescence is thus ensured [3] before reconfiguration is initiated.

Next, we consider persistent clients that require predictable or composable performance, and have the same requirements before and after the use-case switch. Their configuration might have to change as a side-effect of other reconfiguration actions. If a TDM arbiter is used, for example, then we might want to move their slots to a different position in the TDM table to make room for another (new) client that requires a large contiguous allocation. Reconfiguration of this type of client can

[1]We recognize that it may be beneficial to apply such permutations for certain applications with regular but nonlinear addressing strides, but in general the data that a client reads should match the data that it wrote earlier.

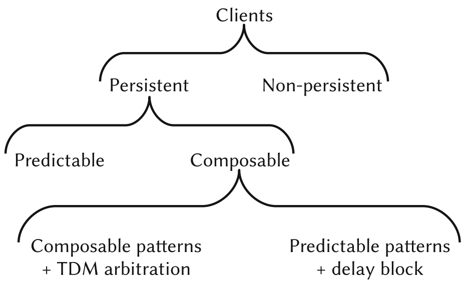

Clients
Persistent
Non-persistent
Predictable
Composable
Composable patterns + TDM arbitration
Predictable patterns + delay block

Fig. 7.3 Client type hierarchy

start once all nonpersistent clients have been stopped. Our goal is to not change the guarantees that are given to persistent clients as a result of reconfiguration, so we define them as follows:

- *The worst-case requirements of a persistent predictable client should always be satisfied, i.e., its guaranteed performance before, during and after a use-case switch is always equal to or higher than its required performance.*
- *The (actual-case) behavior of a persistent composable client should not be influenced by the use-case switch in any way, i.e., for the client, it is impossible to determine whether or not reconfiguration took place by observing data, timings or a combination of the two.*

These definitions allow the verification process of these types of clients to remain unchanged, regardless of the possible reconfigurations they might experience during their lifetime. Note that it is possible to combine the two guarantees for clients that need to be both predictable and composable. However, the composable guarantee is stronger and enforces predictability during reconfiguration, assuming the client was receiving predictable performance before the use-case switch.

Finally, the newly starting clients in the use-case we transitioned to can be enabled to complete the reconfiguration process.

7.3 Delay Block/Arbiter Reconfiguration with Persistent Clients

Delay blocks and the arbiter are reconfigurable in the presence of persistent clients, as discussed in Sect. 7.1. We now evaluate under which conditions we actually *can* reconfigure, given the effect it has on the performance guarantees defined in Sect. 7.2.

Clients receiving predictable performance do not require a delay block. The memory controller may use composable patterns and TDM arbitration to provide composable performance, or it can use predictable patterns and delay blocks. In the latter case, the delay block may not be reconfigured by definition, since that alters the actual-case behavior of the client. Instead, their delay blocks should always be configured to emulate the worst-case performance across all use-cases in which the client

Table 7.1 Components we can reconfigure for persistent clients

Client type	Delay block	Arbiter
Persistent predictable clients		
	Not used	Yes, if transition is safe (shown for TDM in Sect. 7.4)
Persistent composable clients		
Composable patterns + TDM	Not used	No (composability)
Predictable patterns + delay block	No (composability)	Yes, if transition is safe (shown for TDM in Sect. 7.4)

is active. We conclude that *even though reconfiguring the delay blocks for persistent clients is not structurally impossible, it is never allowed or useful.*

A client's predictable performance guarantees depend on the configuration of the arbiter. The arbiter may therefore only be reconfigured if the guaranteed performance for all persistent clients is higher than or equal to their respective requirement before, during and after reconfiguration. *If we assume the arbiter is only reconfigured to switch between valid configurations that individually satisfy the client's requirements, then we only have to ensure that the transition, i.e., the process of reconfiguration itself does not cause requirement violations.* If this is the case, then we call the reconfiguration process *safe*. Composable clients that use delay blocks need the same assertion to assure their (single) delay block configuration is conservative. Proving this assertion holds for an arbiter, even if we restrict ourselves to predictable arbiters, is not trivial. Section 7.4 discusses the challenges, and how it can be done for a TDM arbiter.

Finally, we consider the options for persistent composable clients using composable patterns with TDM arbitration. Their private arbiter configuration, i.e., their slots in the slot-table, may not be changed, since they rely on these to be constant for composability (Sect. 3.3.1). However, the *other* slots that are not owned by clients of this kind are still reconfigurable.
The results of this section are summarized in Table 7.1.

7.4 Reconfigurable TDM Arbiter

Arbiters make run-time scheduling decisions based on their state variables, their configuration, and the availability of new requests from clients. When the configuration is changed through reconfiguration, it has to remain *consistent* with the state variables at the time of reconfiguration. What consistent means in this context depends on the arbiter type: for example, if a TDM arbiter is reconfigured from a slot-table size of 10 to a size of 5, then the state is only consistent if the current slot-table pointer lies within the new slot-table range. The more of these variables there are to consider, the tougher this becomes, and hence, it is easier to safely reconfigure for example a TDM arbiter, which only holds its slot-table position as state variable, compared to a CCSP arbiter [4], which has individual credit counters for each client.

In this section, we focus on (work-conserving and non-work-conserving) TDM arbiters, and limit reconfiguration to the allocation of slots to clients, leaving the table size untouched to avoid consistency issues. Reconfiguration might change a predictable persistent client's allocation by adding, removing, or moving its slots. *Adding slots* is always safe, since it can only improve the performance of a client. *Removing slots* is also safe within our performance guarantee definitions (Sect. 7.2): a transition to a configuration with fewer slots is only allowed if this configuration also satisfies the client's requirements, and hence the slots that are removed can be considered overallocation. This only leaves the case where slots are moved within the table.

Section 7.4.1 first describes how a slot allocation can be translated into a (latency-rate) performance guarantee. Section 7.4.2 explores the slot-moving scenario further by first showing how an (atomic) move operation leads to temporarily reduced performance. It then derives a reconfiguration protocol for TDM arbiters that preserves the guaranteed performance of persistent predictable clients during reconfiguration. Section 7.4.3 shows the architecture of a TDM arbiter that satisfies the protocol constraints. Section 7.4.4 formalizes our approach to prove its correctness.

7.4.1 Latency-Rate Parameters for TDM Arbiters

A TDM arbiter divides the resource time into *slots* that are distributed to multiple clients. Each slot represents a time slice in which one client can use the resource. A slot is nonpreemptive, but its duration can be bounded by the WCIAT (Sect. 2.4.3). We assume allocation of slots to clients is done at design time, yielding a slot table that maps each slot to a certain client. The length of the slot table (or frame) $\mathcal{T}$, defines the period of the arbiter in number of slots. Each slot corresponds to a fraction $1/\mathcal{T}$ of the worst-case bandwidth (b_{wc}), such that a client c that receives ϕ^c slots has a (normalized) allocated rate of:

$$\rho^c_{tdm} = \frac{\phi^c}{\mathcal{T}} \tag{7.1}$$

Intuitively, the service latency for a client c using a TDM arbiter expressed in slots $\left(\Theta^c_{tdm}\right)$ is the worst-case number of slots this client has to wait until the arbiter reaches one of its slots. If a TDM arbiter uses contiguous (greedy) allocation, this is equal to $\mathcal{T}$ times the rate not allocated to this client $\left(1 - \rho^c_{tdm}\right)$, plus one, as shown in Eq. (7.2). The plus one accounts for the misalignment of the arrival of a atom with the arbitration moments. In the worst case, a decision has been made one cycle before the arrival, and the client is too late to claim its slot.

$$\Theta^c_{tdm} = \mathcal{T} \cdot (1 - \rho^c_{tdm}) + 1 = \mathcal{T} - \phi^c + 1 \tag{7.2}$$

We only discuss the arbiter's $\mathcal{LR}$ abstraction in the remainder of this chapter. To simplify our notation, we use (Θ, ρ) to represent $(\Theta^c_{tdm}, \rho^c_{tdm})$.

7.4.2 Safe TDM Arbiter Reconfiguration protocol

Each memory controller client receives a $\mathcal{LR}$ performance guarantee based on its allocated slots in the TDM arbiter. When the arbiter is reconfigured to switch between different slot allocations, the number of slots the client receives over a period of $\mathcal{T}$slots may reduce temporarily, even if the number of allocated slots within each table iteration remains constant, as shown by example in Fig. 7.4. Three of such table iterations are shown in the figure. A letter in a slot indicates the slots belongs to the client corresponding to that letter, and that it can claim that slot, if it has a request available. In this example, we consider the case where the request from A arrives just *after* its slot in the first iteration has started, which means it is too late to claim it. If the same slot allocation had been used in the second iteration of the table, then A's response time would have been 6 slots. However, the arbiter is reconfigured, and A's slot is moved to the end of the table. Instead of 6 slots, it now sees a response time of 10 slots. This *could* mean its $\mathcal{LR}$ guarantees are violated, depending on the tightness of its service bound. This section discusses a reconfiguration protocol that prevents bound violations.

Coffee Machine Analogy

Consider a simple filter coffee machine in a university kitchen. The first Ph.D. student who arrives in the morning finds the machine in a nonoperational state. This is not unexpected, and the student knows how to deal with this problem. After a certain initial service latency (adding the filter, water, and ground coffee), the machine produces a stream of coffee at a steady rate. Once the water is depleted, the machine no longer provides the service until someone replenishes its resources, reconfiguring it for continued operation. A caffeine seeker arriving right after this reconfiguration takes place will be unexpectedly disappointed by the provided service of the coffee machine. Not being the first one in the office, he or she would expect to get coffee immediately, but this is not the case now. Clearly something must be done to fix this issue.

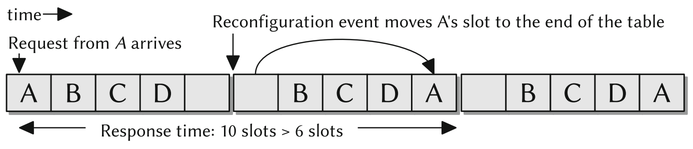

time
Request from A arrives
Reconfiguration event moves A's slot to the end of the table
A
B
C
D
B
C
D
A
B
C
D
A
Response time: 10 slots > 6 slots

Fig. 7.4 Example of potentially violated $\mathcal{LR}$ guarantees for client A during reconfiguration. The figure shows 3 TDM-table iterations of 5 slots each. A letter in a slot indicates the slots belongs to the client corresponding to that letter

Adding a second coffee machine intuitively provides a solution to this problem, since the additional capacity can compensate for the temporary loss of service. A reconfiguration protocol is quickly established: a set time before the first machine runs out of resources, the second machine needs to be prepared for operation. Once the first runs out of coffee, the second is ready to take over, such that a steady rate of coffee is guaranteed.

Our TDM arbiter uses the same principle that the coffee drinkers applied in the analogy to ensure that moving slots does not lead to a reduction of the service guarantee. It knows that slots will be removed at some time in the future due to reconfiguration, and compensates by activating the slots that will replace them early enough. The key insights are that *moving slots should not be an atomic action, but should instead be broken up in the removal and subsequent addition of slots, and that these operations can be reversed.* Section 7.4.4 derives what the minimum amount of time between these operations has to be in order to retain the original $\mathcal{LR}$ guarantees during reconfiguration.

7.4.3 Arbiter Architecture

A schematic representation of the TDM arbiter architecture we propose is shown in Fig. 7.5. It contains of a set of registers that represents the active TDM *slot table*. Each slot contains the bus-port id of the client to which the slot belongs. An incrementing wrapping *index counter* selects the next client to be scheduled from the slot table. The wrap-around value is configurable, such that multiple table lengths can be implemented by the same hardware.

A copy of the slot table is kept in the *shadow table*, which can be reprogrammed through a (DTL) configuration port. All slot reconfigurations are first applied to the shadow table. One configuration message can reassign a contiguous slot range in the shadow table to a different client. The shadow table is locked from further updates after each configuration message until its contents is copied to the slot table. While it is locked, the reconfiguration module does not accept new reconfiguration messages.

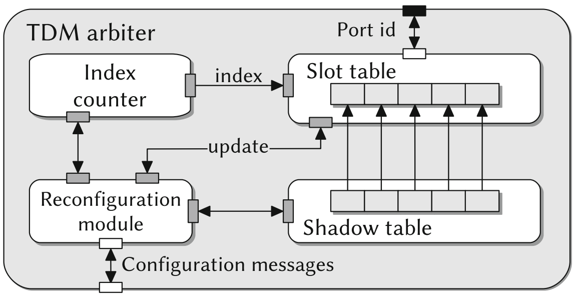


Fig. 7.5 Reconfigurable TDM arbiter architecture

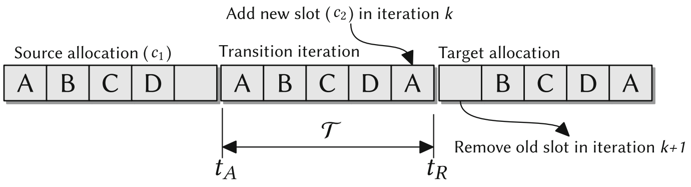

Add new slot (c_2) in iteration k
Source allocation (c_1)
Transition iteration
Target allocation
A B C D
A B C D A
B C D A
$\mathcal{T}$
Remove old slot in iteration k+1
t_A
t_R

Fig. 7.6 Splitting the reconfiguration in two steps that take place in separate table iterations guarantees that the provided service is always greater than the guaranteed service

The purpose of the *reconfiguration module* is to implement our safe reconfiguration protocol. It delays the actual reconfiguration of the slot table until the index counter wraps around. Only then is the content of the shadow table copied to the slot table, and hence, the new configuration immediately takes effect. If a predictable client is reconfigured to a different set of contiguous slots, then two configuration messages are used: the first one enables the new slots, while the second one disables the old slots. As a result, the module forces the transition phase where both the new and old allocation are given to be at least one table iteration. Figure 7.6 shows what this mechanism looks like in an example. For arbitrary allocations, two configuration messages are required to add and remove each contiguous block of slots.

7.4.4 Latency-Rate Guarantees During Reconfiguration

We evaluate reconfiguration effects at the slot granularity. We formally prove that the $\mathcal{LR}$ guarantees at this level of abstraction are not invalidated if our reconfiguration protocol is used. This is a sufficient condition to guarantee that the $\mathcal{LR}$ bound expressed in clock cycles is also valid, since the transformation function from slots to clock cycles is monotonically increasing [5].

A $\mathcal{LR}$ server offers a linear lower bound on the provided service within a busy period [6], and is defined as follows:

Definition 7.1 ($\mathcal{LR}$ *server*) Let τ be the starting time of a busy period $[\tau, \tau']$ for server s_i with a service latency Θ_i and allocated rate ρ_i. For all times t during this busy period, a lower bound on the provided service by s_i is given by:

$$w_i(t) = \max\left(0, \rho_i \cdot (t - \tau - \Theta_i)\right) \qquad \forall t \in [\tau, \tau'] \tag{7.3}$$

The lower bound on provided service by a $\mathcal{LR}$ server is maximal if the client keeps the server continuously busy. If reconfiguration does not lead to a violation of the bound under this condition, then it is also safe in all other cases. If a $\mathcal{LR}$ server is reconfigured, for example by changing the underlying TDM slot allocation, its Θ and ρ may change. We assume the allocations before and after reconfiguration are chosen such that they satisfy the $\mathcal{LR}$ requirements of the client, as mentioned earlier in Sect. 7.3.

Definition 7.2 The required $\mathcal{LR}$ service bound of the client , $w_r(t)$, in a busy period $[\tau, \tau']$ is given by:

$$w_r(t) = \max(0, \rho_r \cdot (t - \tau - \Theta_r)) \qquad \forall t \in [\tau, \tau'] \tag{7.4}$$

where (Θ_r, ρ_r) represent the client's $\mathcal{LR}$ requirements.

We model reconfiguration as the handover of a client between two independent $\mathcal{LR}$ servers. The first and second server are characterized by the allocation before and after reconfiguration, respectively.

Definition 7.3 Let c_1 and c_2 be two independent and distinct allocations for a client. The corresponding $\mathcal{LR}$ parameters for allocation c_1 and c_2 are denoted by (Θ_1, ρ_1) and (Θ_2, ρ_2), respectively. Both of these parameter sets satisfy the client's $\mathcal{LR}$ requirements, such that $\rho_r \leq \min(\rho_1, \rho_2)$ and $\Theta_r \geq \max(\Theta_1, \Theta_2)$.

The important words in this definition are *independent and distinct*: it should be possible to enable or disable one of the $\mathcal{LR}$ servers corresponding to these allocations without affecting the other's service bound (their respective worst-case performance guarantees should be independent). For a TDM arbiter this implies that c_1 and c_2 may not have overlapping slots. We will relax this requirement later.

We assume that c_1 is initially active, and through reconfiguration, the client is handed over to c_2. To model this behavior, we define two time instances: t_A is the time at which allocation c_2 is fully enabled in the slot table, and t_R is the time at which allocation c_1 is fully disabled in the slot table.

The total service guaranteed to a TDM client is conservatively bounded by the sum of the service provided by each slot that is allocated to it, because ρ and Θ monotonically increase and decrease, respectively, with the number of allocated slots (see Eqs. (7.1) and (7.2)). This property allows us to (lower) bound the guaranteed service during reconfiguration as the sum of the service provided by allocations c_1 and c_2 (because there are no overlapping slots). Combining Definitions 7.1 and 7.3 yields:

Definition 7.4 For a time t during a busy period $[\tau, \tau']$, the service guarantee $w_g(t)$ of a server that is reconfigured from c_1 to c_2 is given by:

$$\begin{aligned} w_g(t) \geq\ & \max(0, \rho_1 \cdot (\min(t, t_R) - \tau - \Theta_1)) \\ & + \max(0, \rho_2 \cdot (t - \max(\tau, t_A) - \Theta_2)) \qquad \forall t \in [\tau, \tau'] \end{aligned}$$

The required $\mathcal{LR}$ service bound may not be violated before, during or after reconfiguration. In other words, $w_g(t)$ has to be larger than or equal to $w_r(t)$ for all $t \in [\tau, \tau']$. This is formally proven in Theorem 7.1.

Theorem 7.1 *If* $t_R - t_A \geq \max(\Theta_1, \Theta_2)$ *then* $\forall t \in [\tau, \tau']$, $w_g(t) \geq w_r(t)$.

Proof We have to conservatively assume there is no overallocation, such that $\rho_1 = \rho_2 = \rho_r$. We can also conservatively substitute Θ_1 and Θ_2 by $\Theta' = \max(\Theta_1, \Theta_2)$. This means that:

$$\max\left(0, \rho_r \cdot \left(\min(t, t_R) - \tau - \Theta'\right)\right) + \tag{7.5}$$
$$\max\left(0, \rho_r \cdot \left(t - \max(\tau, t_A) - \Theta'\right)\right) \geq \tag{7.6}$$
$$\max\left(0, \rho_r \cdot \left(t - \tau - \Theta'\right)\right) \tag{7.7}$$

has to hold for all t. There are four cases that can be distinguished. Case 1, 3, and 4 are visualized in Fig. 7.7:

1. $\tau \leq t \leq t_R$:
   As long as c_1 is not disabled, then Eqs. (7.5–7.7) is satisfied by c_1's contribution to the total service, expressed by Eq. (7.5). In Fig. 7.7, the requirement line $w_r(t)$ overlaps with c_1's contribution while $t \leq t_R$.
2. $t_A \leq \tau$:
   Similarly, if the busy period starts after allocation c_2 is enabled, then Eqs. (7.5–7.7) is satisfied by c_2's contribution, expressed by Eq. (7.6).
3. $\tau \leq t \leq \tau + \Theta'$:
   As long as $w_r(t)$ is 0, i.e., before the rate phase of the server begins, then Eqs. (7.5–7.7) is also trivially satisfied. In Fig. 7.7, this corresponds to the flat line portion of $w_r(t)$ marked with Θ_1.
4. $t > \tau$ and $t > t_R$ and $t_A > \tau$ and $t > \tau + \Theta'$:
   This only leaves the complement of the union of the previous three cases: c_1 has been removed, c_2 is activated after the start of the busy period, and the rate phase of the server has started. Applying these case constraints to Eq. (7.5–7.7), and dividing by ρ_r yields:

$$\max\left(0, t_R - \tau - \Theta'\right) + \max\left(0, t - t_A - \Theta'\right) \geq t - \tau - \Theta' \tag{7.8}$$

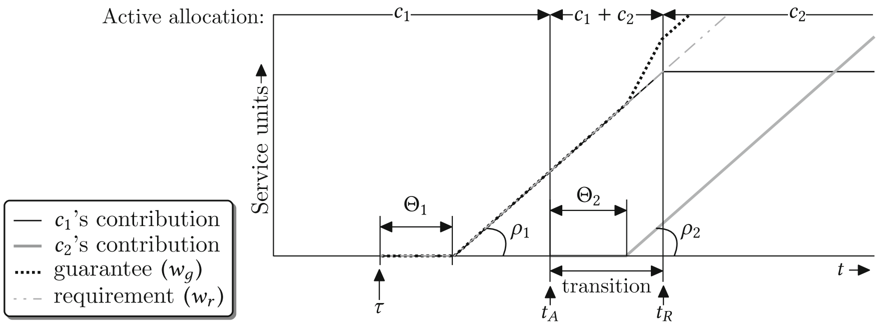


Fig. 7.7 Example of the latency-rate guarantees during reconfiguration

Because $t > t_R$ in this case, the first max-term alone does not satisfy the inequality, and hence, the second max-term needs to contribute, so we additionally require:

$$t > t_A + \Theta' \tag{7.9}$$

to hold. Eliminating common terms in Eq. (7.8) given Eq. (7.9) yields:

$$\max\left(0, t_R - \tau - \Theta'\right) \geq t_A - \tau \tag{7.10}$$

Because $t_A > \tau$ in this case, we now know the left-hand side should yield a nonzero result and hence:

$$t_R > \tau + \Theta' \tag{7.11}$$

has to hold. Removing the common terms from Eq. (7.10) and rearranging leaves:

$$t_R - t_A \geq \Theta' = \max(\Theta_1, \Theta_2) \tag{7.12}$$

If we assert that Eq. (7.12) holds, then $w_g(t) \geq w_r(t)$ holds in Case 4 given that (7.9) and (7.11) are true. Combining Eq. (7.12) with case constraint $t > t_R$ yields (7.9), and combining Eq. (7.12) with case constraint $t_A > \tau$ yields (7.11), confirming these assumptions.

This means that if Eq. (7.12) holds, then $w_g(t) \geq w_r(t)$ holds for all $t \in [\tau, \tau']$ which concludes the proof. □

Equation (7.12) enforces a minimum interval of $\max(\Theta_1, \Theta_2)$ where both the c_1 and c_2 have to be provided by the server. During that transition period, the server temporarily assigns both slot allocations to the client. Figures 7.6 and 7.7 illustrate this. Considering what happens when t_R moves further to the left (i.e., removing the original allocation earlier), we see the interval in which w_g is larger than w_r get smaller until $t_R - t_A = \max(\Theta_1, \Theta_2)$, where the guarantee completely overlaps the requirement.

7.4.4.1 Overlapping Slots

Definition 7.3 stated that c_1 and c_2 had to be distinct allocations, and reusing slots in the TDM table across the corresponding configurations was hence not allowed. The reason for this limitation is that we sum the contribution of c_1 (Eq. (7.5)) and c_2 (Eq. (7.6)) in Eq. (7.7). If we relax this requirement, and allow c_1 and c_2 to have overlapping slots, then we need to make sure we only count each slot once in all cases. At t_R, we now only remove the nonoverlapping slots of c_1.

Definition 7.5 Let ϕ_{ol} be the number of overlapping slots in allocation c_1 and c_2. The corresponding rate of those slots is denoted with ρ_{ol}.

Theorem 7.2 *If c_1 and c_2 contain ϕ_{ol} overlapping slots, then $t_R - t_A \geq \Theta' + \frac{\rho_{ol}}{\rho_r} \cdot (t_R - t_A - \Theta')$ should be satisfied to assert that $\forall t \in [\tau, \tau']$, $w_g(t) \geq w_r(t)$.*

Proof The first three cases of Theorem 7.1 remain unmodified, since only c_1 or c_2 is active in those cases, leaving only Case 4. We can narrow down Case 4 further by using $t_R > t_A$, as required by Theorem 7.1 for safe reconfiguration when there are no overlapping slots. Case 4 then turns into: $t > t_R > t_A > \tau$ and $t > \tau + \Theta'$. The contribution of the overlapping slots is counted twice in the interval $[t_A, t_R]$, and for that we need to compensate by subtracting $\rho_{ol} \cdot (t_R - t_A - \Theta')$ service units from $w_g(t)$. Adding this term to Eqs. (7.5–7.7) yields:

$$\rho_r \cdot \left(t_R - \tau - \Theta'\right) \ + \rho_r \cdot \left(t - t_A - \Theta'\right) - \rho_{ol} \cdot \left(t_R - t_A - \Theta'\right) \geq \rho_r \cdot \left(t - \tau - \Theta'\right) \tag{7.13}$$

Dividing both sides of Eq. (7.13) by ρ_r and removing common terms yields:

$$t_R - t_A \geq \Theta' + \frac{\rho_{ol}}{\rho_r} \cdot \left(t_R - t_A - \Theta'\right) \tag{7.14}$$

□

This result is similar to Eq. (7.12), although it constrains $t_R - t_A$ further, since there is an extra term on the right-hand side accounting for the overlap.

7.4.4.2 Application to Our Arbiter

The arbiter we introduced in Sect. 7.4.3 forces the transition phase where both the new and old allocation are given to be at least one table iteration, so $t_R - t_A \geq \mathcal{T}$.

Theorem 7.3 *If $t_R - t_A \geq \mathcal{T}$, then reconfiguration is guaranteed not to violate the latency-rate guarantees of the client for our TDM arbiter.*

Proof Reconfiguration is guaranteed to not violate the latency-rate guarantees of the client if $t_R - t_A \geq \Theta' + \frac{\rho_{ol}}{\rho_r} \cdot (t_R - t_A - \Theta')$, satisfying Eq. (7.14). Substituting Θ' with the service latency equation for TDM arbiters (Eq. (7.2)), and $t_R - t_A$ with $\mathcal{T}$ in this equation yields:

$$\mathcal{T} \geq \mathcal{T} - \min\left(\phi_1, \phi_2\right) + 1 + \frac{\rho_{ol}}{\min\left(\rho_1, \rho_2\right)} \cdot \left(\mathcal{T} - \left(\mathcal{T} - \min\left(\phi_1, \phi_2\right) + 1\right)\right) \tag{7.15}$$

After removing the common terms and rearranging, we have:

$$\min\left(\phi_1, \phi_2\right) \geq 1 + \frac{\rho_{ol}}{\min\left(\rho_1, \rho_2\right)} \cdot \left(\min\left(\phi_1, \phi_2\right) - 1\right) \tag{7.16}$$

The allocated rate in a slot-based TDM arbiter is equal to the number of allocated slots divided by the table length, $\rho_{tdm}^{c} = \frac{\phi^c}{\mathcal{T}}$ (Eq. (7.1)). Applying this to Eq. (7.16):

$$\min(\phi_1, \phi_2) \geq 1 + \frac{\phi_{ol}}{\min(\phi_1, \phi_2)} \cdot (\min(\phi_1, \phi_2) - 1) = 1 + \phi_{ol} - \frac{\phi_{ol}}{\min(\phi_1, \phi_2)} \tag{7.17}$$

We distinguish three cases:

1. $\phi_{ol} = 0$:
   In cases where $\phi_{ol} = 0$, we use the fact that the minimum allocation is 1 slot, so $\min(\phi_1, \phi_2) \geq 1$, which satisfies Eq. (7.17).
2. $\phi_{ol} = \min(\phi_1, \phi_2)$:
   In this case, all slots from the first configuration are also used in the second, meaning we are removing or adding *over-allocated* slots, since $\rho_r \leq \min(\rho_1, \rho_2)$ (Definition 7.3). This can not violate the latency-rate guarantees of the client. Applying this case constraint in Eq. (7.17) confirms this.
3. $\min(\phi_1, \phi_2) > \phi_{ol} > 0$:
   The number of allocated slots cannot be negative, and the overlap can at most be as large as the minimum number of allocated slots, so for the rightmost term of Eq. (7.17) we know:

   $$0 \leq \frac{\phi_{ol}}{\min(\phi_1, \phi_2)} \leq 1 \tag{7.18}$$

   In cases not captured by case 1 and 2, $\min(\phi_1, \phi_2)$ is at least one slot larger than ϕ_{ol}, satisfying Eq. (7.17).

□

With this proof, we have shown that the $\mathcal{LR}$ guarantees of the arbiter are unaffected when it is reconfigured, and therefore, the WCRT guarantees derived in Sect. 2.4 that are based on these guarantees remain valid.

7.5 Evaluation

This chapter presented the various reconfiguration options offered by our memory controller and showed in which context they can be used. It also presented a TDM reconfiguration protocol and associated arbiter architecture that enable to moving the slots of persistent predictable clients, while asserting their performance guarantees are not violated. The cost of the reconfiguration-related hardware has been evaluated in Sect. 2.6, which showed the relative overhead is negligible. In the current section, we evaluate the performance guarantees offered during reconfiguration by means of two experiments. The first experiment, Sect. 7.5.1, uses the SystemC model of the controller, while the second experiment, Sect. 7.5.2, uses the VHDL instance.

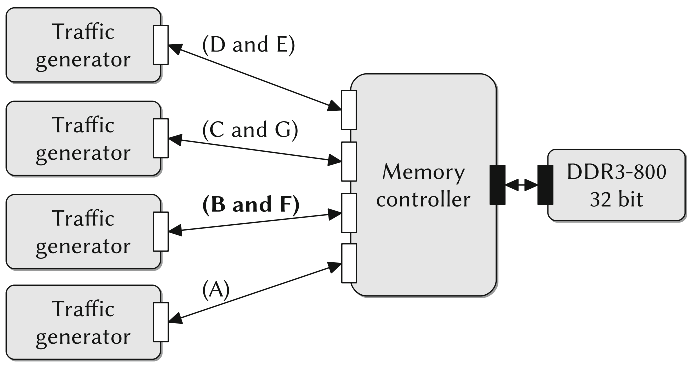


Fig. 7.8 Experimental setup for Sect. 7.5.1. Labels on the *arrows* correspond to the client name(s) that use the connection

7.5.1 Predictable Performance During Reconfiguration

In this experiment, we demonstrate that the controller offers predictable performance to its clients during reconfiguration through simulations with the SystemC model. The experimental setup is shown in Fig. 7.8. A four-port instance of the controller is used, connected to a model of a 32-bit DDR3-800 device using a (default) 400 MHz command clock.

Seven synthetic clients (denoted A–G) share the SDRAM resource in this experiment. Figure 7.9 shows the compositions of active clients and their properties over the course of the experiment. Each bar represents a client and is annotated with its name, bandwidth requirement, and the type of performance it requires (either predictable, or predictable and composable). The controller uses composable patterns and TDM arbitration to provide composable performance. For simplicity of the example, we assume all clients have a relaxed WCRT requirement of 2000 ns. Clients generate requests of 128 bytes, half of these are reads, and half are writes.

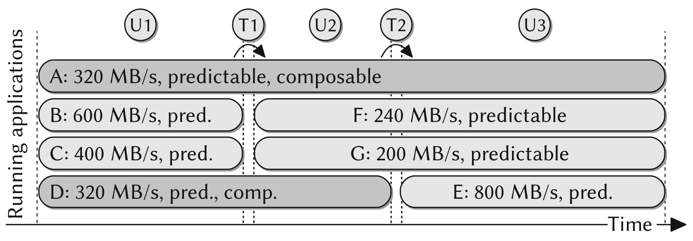


Fig. 7.9 Active clients over time. Three use-cases are visited: $U1$ (A, B, C, D), $U2$ (A, D, F, G), and $U3$ (A, E, F, G)

Four synthetic traffic generators represent the clients. Each is connected to separate ports on the memory controller. The generators can have an unlimited number of outstanding requests, and will hence issue requests at the rate dictated by their bandwidth requirement, as long as no back pressure is applied by the memory controller. The atom buffers in the controller are over-dimensioned, such that they do not cause back-pressure during normal operation, and hence, the arrival times of requests are independent from the memory controller's behavior. This allows us to focus purely on the variations in the response time of each individual atom, without having to take the effect on the arrival times of later requests into account.

Clients on the same horizontal line in Fig. 7.9 are mutually exclusive and share a port and traffic generator. Clients on the same vertical line are active simultaneously, resulting in three distinct use-cases, annotated with $U1$ (A, B, C, D), $U2$ (A, D, F, G) and $U3$ (A, E, F, G). At $T1$ and $T2$, use-case transitions take place.

The controller uses a (BI 1, BC 4) pattern configuration with a worst-case bandwidth (b_{wc}) of 1862 MB/s. The conversion to composable patterns does not impact the memory efficiency for this configuration, because it is write-dominant (Sect. 3.3.2). The slot-table size $\mathcal{T}$ in the arbiter is set to 20 slots, such that each slot corresponds to $1862/20 = 93$ MB/s.

Slots are assigned to clients in contiguous blocks using a greedy allocation algorithm, considering one use-case at a time. Because clients A and D require composable performance, they require the same slot allocation in all use-cases where they are active, and are allocated first, similar to [7]. This is reflected in the allocation algorithms' output, which is shown in Fig. 7.10. Note that without reconfiguration support, the slots for client F and G would not be movable and client E would be unmappable due to fragmentation, although one could argue this is a limitation of the allocation algorithm. Several more advanced slot allocation strategies that consider real-time constraints exist [8–10]. The flexibility to move slots, as offered by our arbiter, enables the use of such algorithms in case they are not capable of considering multiple use-cases at once. In general, it reduces the number of constraints they have to take into account, which may lead to more successful allocations. However, for the purpose of this experiment, we stick to our basic greedy algorithm.

Figure 7.11 shows the temporal behavior of read requests at the atom buffers for clients B and F. The position on the x-axis of the bars corresponds to the time at which a read atom arrives in the atom buffer of the client. The height of the bar corresponds to the measured response time, i.e., the time until the data corresponding to the atom is fully received by the atom buffer. Two runs of the experiment are shown, drawn

	0					5					10					15				
U1:	A	A	A	A	D	D	D	D	B	B	B	B	B	B	B	C	C	C	C	C
U2:	A	A	A	A	D	D	D	D	F	F	F	G	G	G						
U3:	A	A	A	A	F	F	F	G	G	G	E	E	E	E	E	E	E	E	E	

Fig. 7.10 Slot allocation results

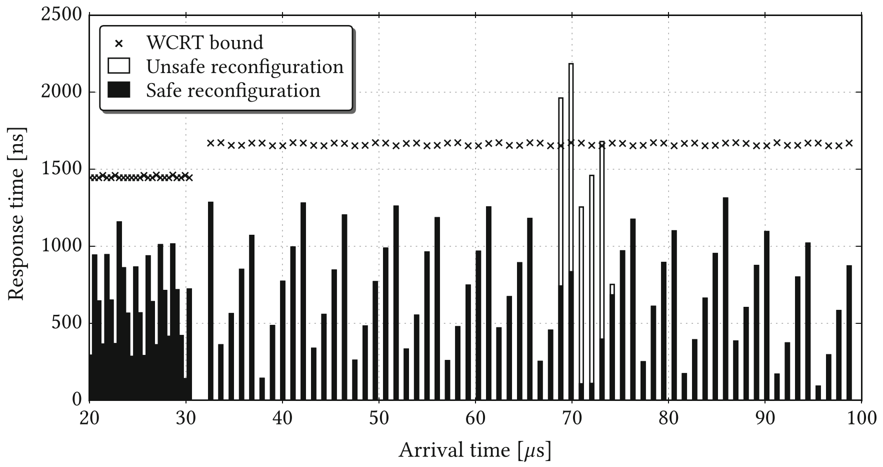


Fig. 7.11 Response times with and without predictable reconfiguration, generated by the SystemC simulation

with black and white bars, respectively. The white bars are mostly hidden behind the black bars, and equal in size in those cases. Based on the $\mathcal{LR}$ guarantee resulting from the slot allocation (Eqs. (7.1) and (7.2)), we determine a WCRT bound per request, which is drawn as x-markers in the graph. It varies slightly due to self-interference (i.e., the client having to wait on its own previous request(s)).

Initially, client B is active, shown in the interval from 20 to 30 μs. At approximately 31 μs ($T1$), the first use-case transition and associated reconfiguration takes place. Client F, becomes active now, which requests at a lower rate, and hence, the bars are spaced further apart from here on.

In the first experiment (white bars), the safe reconfiguration mechanism in the TDM arbiter is switched off, and transitions between different configurations happen instantaneously, i.e., new slots are added and old slots are removed at the same time. At 68 μs ($T2$), the transition from $U2$ to $U3$ takes place. The WCRT bounds of some requests are violated as a consequence of reconfiguring the arbiter, which is unacceptable. It shows that unconstrained reconfiguration is not safe.

A second run (black bars) is performed with the safe reconfiguration mechanism switched on. Here the WCRT bound is valid during reconfiguration, and the measured response times are slightly lower, since the client temporarily gets more slots. *This experiment suggests that our reconfiguration protocol is safe.*

7.5.2 Composable Performance During Reconfiguration

The second experiment demonstrates that the controller offers composable performance using composable patterns and TDM arbitration to clients that require this, even while the arbiter is reconfigured. This is contrasted to a run where predictable patterns are used, where we demonstrate that the interapplication interference changes as a result of reconfiguration.

We use a two-port VHDL instance of the (Raptor) controller, and hence, perform the experiments on our FPGA. A pattern set with (BI 1, BC 2) is used, which guarantees a worst-case bandwidth (b_{wc}) of 933 MB/s. The TDM slot-table size is set to 8 slots.

Two MicroBlaze processors (MB1 and MB2) are connected to our memory controller through a DMA. Each MicroBlaze runs one application, referred to by the name of the MicroBlaze. The applications consist of a simple loop that generates bursts of memory requests at an average rate of 90 MB/s. Each application maps to a single client and thus single port on the memory controller.

The atom buffers are instrumented with timers that keep track of the arrival and response times of the requests, similar to the SystemC setup from the previous experiment. These timestamps are recorded and read out after the experiment. For each experiment, we wait until the PHY finishes its initialization, and then program the initial configuration in the memory controller. For the purpose of this experiment, the start of the refresh timer and the first arbiter iteration are synchronized, making behavior across multiple runs more likely to be repeatable, although some nondeterminism remains, as discussed earlier in detail in Sect. 3.4.3.2. We plot the most commonly observed timestamp series per run.

Six different runs are performed, divided into two groups, one using predictable patterns and the other using composable patterns, doing three runs per group. In all runs, MB1 gets 4 slots in the table.

1. Reference run: Only MB1 runs its application, while MB2 remains idle.
2. Interference run: Both MB1 and MB2 are active. MB2 generates an interfering stream of write requests and gets 4 slots in the TDM table.
3. Reconfiguration run: Both MB1 and MB2 are active. MB2 initially has 1 slot in the TDM table, but is reconfigured to 2 slots after 32 μs.

Figure 7.12 shows the measured arrival and response times of the MicroBlazes in the first three runs. Predictable patterns are used, i.e., the slot length varies with the request that is executed. Even though application MB1 is not changed across the three runs, practically none of its timestamps overlap. Its behavior is hence affected by the interference from MB2. In the reconfiguration run, there is a difference in the behavior of MB1, even though we only change the allocation of MB2.

The second group of runs uses composable patterns (Fig. 7.13) to effectively eliminate all interference across the two applications in a use-case, and during use-case switches. The figure illustrates that MB1 is not affected by any of the actions of MB2, nor by the reconfiguration of the arbiter, and its behavior is constant. The reference run is thus representative for the behavior after integration, thus enabling independent verification of the application in isolation.

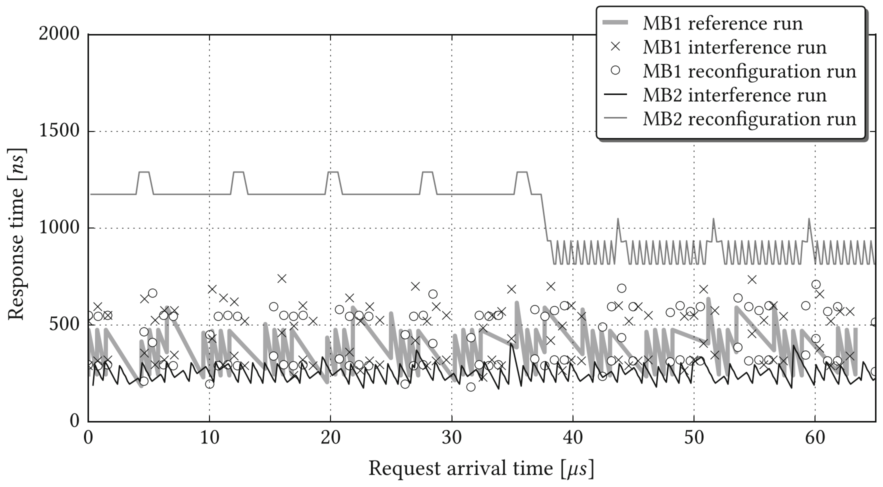


Fig. 7.12 Predictable patterns runs. Note how the response times in the MB1 interference and reconfiguration runs are different with respect to the reference run, indicating MB2 influences the (actual-case) performance of MB1

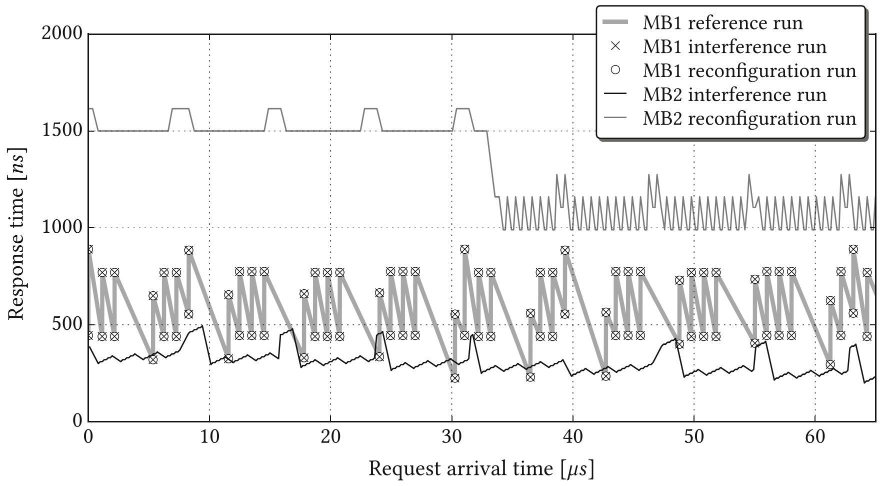


Fig. 7.13 Composable patterns runs

7.6 Conclusion

This chapter showed that our memory controller has a flexible architecture with various reconfigurable components. There is, however, a crucial difference between its configuration options at boot time, and its reconfigurability at run time. In the latter case, we need to make sure that state that was built up in the SDRAM and controller by persistent clients is retained, and that the new configuration is consistent with it, which limits the options. Resisting timing variations is an inherent part of predictable and composable performance guarantees. Reconfiguring the very components that facilitate these guarantees is hence often not possible in the presence of persistent clients, although the resource arbiter remains reconfigurable under certain conditions. We showed how a TDM arbiter can satisfy these conditions, and prove that safely reconfiguring it between valid configurations without degrading its $\mathcal{LR}$ guarantees is possible. An implementation of this arbiter and reconfiguration scheme, both in SystemC and VHDL, demonstrated its effectiveness.

References

1. Sinha S, Koedam M, Breaban G, Nelson A, Nejad AB, Geilen M, Goossens K (2015) Composable and predictable dynamic loading for time-critical partitioned systems on multiprocessor architectures. Microprocess Microsyst
2. Goossens K, Koedam M, Sinha S, Nelson A, Geilen M (2015) Run-time middleware to support real-time system scenarios. In: Proceedings of the European conference on circuit theory and design (ECCTD)
3. Kramer J, Magee J (1990) The evolving philosophers problem: dynamic change management. IEEE Trans Softw Eng 16(11):1293–1306
4. Akesson B, Steffens L, Strooisma S, Goossens K (2008) Real-time scheduling using credit-controlled static-priority arbitration. In: Embedded and real-time computing systems and applications (RTCSA)
5. 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
6. Stiliadis D, Varma A (1998) Latency-rate servers: a general model for analysis of traffic scheduling algorithms. IEEE/ACM Trans Netw 6(5)
7. Hansson A, Coenen M, Goossens K (2007) Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In: Design, automation and test in Europe conference and exhibition (DATE)
8. 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 applications symposium (RTAS)
9. Stuijk S, Basten T, Geilen M, Ghamarian AH, Theelen B (2008) Resource-efficient routing and scheduling of time-constrained streaming communication on networks-on-chip. J Syst Architect 54(3):411–426
10. Stefan R, Goossens K (2011) An improved algorithm for slot selection in the æthereal network-on-chip. In: Proceedings of the fifth international workshop on interconnection network architecture: on-chip, multi-chip (INA-OCMC), pp 7–10

Chapter 8
Related Work

This related work chapter is split into three main sections. The first one, Sect. 8.1, discusses various approaches toward the construction and analysis of SDRAM controllers. Section 8.2 considers performance-overviews for SDRAM memories, while Sect. 8.3 discusses other approaches towards run-time reconfiguration of (shared) resources under real-time constraints.

8.1 SDRAM Controllers

Many SDRAM command schedulers and/or controllers have been proposed in related work, employing a range of methods to improve different performance aspects. First, we discuss the works that focus on average-case performance in Sect. 8.1.1. We distinguish papers that introduce new techniques or hardware, and those that analyze existing (COTS) memory controllers. Later, in Sect. 8.1.2, we show an extensive overview of controllers that are specifically constructed to be analyzable, or are otherwise meant to be used in a (mixed-)real-time application area.

8.1.1 Average-Case-Oriented Controllers

Methods that improve average-case performance of SDRAM are abundant, exploiting locality [1, 2], grouping requests per thread [3], exchanging more information with the cache [4] and even using reinforcement learning to adapt the scheduling policy at run-time [5]. These techniques interact with the command scheduling in complex ways, relying on unpredictable request reordering schemes that are effectively impossible to analyze, which means no useful bounds on the real-time performance can be derived. This makes it very challenging to use them in a real-time context.

Certain works analyze COTS memory controllers, in order to derive the worst-case performance estimates for them. These approaches are by design fairly coarse-grained in terms of the guarantees they can satisfy, since the visibility on the cycle-by-cycle hardware behavior is limited. Shah et al. [6] shows a technique to bound the WCET of applications that use the SDRAM. The technique is tested on an Altera FPGA platform, where the worst-case parameters of the associated memory controller are empirically determined. Each request is interleaved over all banks in the memory device. Yun et al. [7] first measures the worst-case bandwidth, and then applies an OS-based bandwidth reservation system to distribute it amongst different cores. The focus of [8] is *First-Ready First-Come First-Served* (*FR-FCFS*) arbitration, where the authors perform a WCRT analysis based on a parameterizable system model. The model requires assumptions on the per-bank arbitration policy, the command scheduler, and a maximum number of outstanding requests per core. It furthermore assumes each request maps to only a single memory burst, and that the number of re-orderings by the "first ready" mechanism is known and finite. If the used COTS cores are timing compositional [9], and used for sporadic tasks with constrained deadlines, then it may be possible to calculate conservative WCRT bounds. As a necessary requirement, the right numbers should be fed into the system model such that it corresponds to the actual hardware implementation in the investigated COTS system. Often this may not possible, since the exact properties of a COTS component are generally not disclosed by the manufacturers.

8.1.2 Real-Time-Oriented Controllers

In this section, we take a closer look at the works on real-time and mixed-time-criticality SDRAM controllers. First, we focus mostly on the back-ends of these controller. We identify the Sect. 8.1.2 basic properties that are shared by practically all of them. These traits are then used as a coarse-grained characterization system, shown in Table 8.1. We consider the following properties in its columns:

1. *The target memory or target-SDRAM type of the work*: The authors of the publication usually have a certain type of memory in mind when they design or evaluate a memory controller. Although the differences between SDRAM generations are usually small, they are relevant when it comes to the quantification of the performance of the proposed controllers. For example, the designer of a controller that is focused on slow devices is less concerned about write-read switching overhead compared to someone considering fast devices. The reason being that the relative size of the WR-to-RD timing constraint compared to one data burst is much smaller.[1] These kinds of differences influence design decisions which are reasonable.

[1]WR-to-RD in the same bank requires 11 cycles for DDR2-800, or the equivalent time of 2.75 bursts, versus 29 cycles or 7.25 bursts for a DDR4-2400 (Sect. 3.1, Appendix B), assuming a data burst takes 4 cycles.

2. *Page policy*: A memory controller may use a close-page policy, an open-page policy, or a hybrid policy, like a conservative open-page policy (Chap. 6) for example. The values in this column of Table 8.1 refer to the implementation of the controllers, but not necessarily their worst-case analysis. Jalle et al. [10] for example uses an open-page policy, but assumes all requests are misses in its analysis.
3. *Device or DIMM focused*: Some works on memory controllers use individual SDRAM devices. The interface width of the SDRAM is then relatively small, equal to the data bus width of a one or two devices, as discussed in Sect. 2.1.4. As a result, the amount of data per burst is also relatively small compared to the size of the requests they process. This allows them to work at access granularities larger than one burst. Works that focus on DIMMs assume that the interface width is much wider. As a result, they can usually fit an entire request into just one burst. This makes it practically impossible to exploit bank-level parallelism within requests for such controllers in the worst case, which impacts the design of their command and request-level schedulers.
4. *Command scheduler granularity*: A memory controller has a component that schedules commands. In our work, the smallest granularity at which commands are scheduled at run-time is a pattern. Some related controllers also use prescheduled sequences in the implementation or analysis of their scheduler. Other controllers make all scheduling decisions at run-time, on a per-command basis. We refer to such controllers as being *dynamically scheduled*.
5. *Burst order*: Some controllers limit or enforce to which banks consecutive bursts are directed to improve the memory efficiency, as discussed conceptually in Sect. 2.2.1. Other controllers direct bursts to distinct ranks [11–13], improving efficiency when read and write bursts are interleaved at a small granularity.
6. *Refresh mechanism*: SDRAM controllers need a way to handle refresh. In this book, refresh is assumed to be triggered automatically by an internal refresh timer, which inserts a refresh pattern in the schedule when required. Variations on this scheme [11, 14] aimed at reducing the costs of refresh within a worst-case analysis are used by some of the mentioned controllers.
7. *Predictable and/or composable performance*: A memory controller can be built to offer predictable and/or composable performance. Note that the addition of a delay block [15] can make most predictable controllers composable under the assumption that a suitable latency-rate abstraction of their performance can be derived.

   Note that when we label a memory controller "non-composable", it sometimes contradict what is stated in the related paper, due to differences in the interpretation of what composability means. In particular, many papers use isolation of WCETs (compositionality) rather than the definition we described in Sect. 1.4.4 [16], which implies isolation of actual-case execution times.
8. (*Smallest*) *spatial mapping granularity*: When the SDRAM is shared amongst multiple clients, they are each assigned a certain fraction of the memory space. The memory controller design influences the minimum granularity at which this space can be distributed. We assume assignment of space is done in multiples of

memory rows (1–2 KiB), i.e., column-level distribution is not considered. Taking our Raptor controller as an example, we interleave a request over BI banks, and hence the smallest possible mapping granularity is a single row in BI banks.

The following sections discuss the controllers in mentioned in Table 8.1 in more detail. We categorize these controllers further by the command scheduler granularity they use, starting with very coarse-grained static command schedulers, and ending with dynamically scheduled solutions.

8.1.2.1 Static Command Schedulers

Bayliss

Within the group of real-time oriented controllers we consider, the amount of a priori information that is assumed to be available and exploitable by the command scheduler varies. Bayliss [20] requires every single request to be known at design time to compute a static command schedule at design time. It is hence completely analyzable, but has limited flexibility, since obtaining this information in a multi-core system is not possible in the general case due to non-determinism (Sect. 3.4.3.2) and dynamism in use-cases, leading to unpredictable interleaving of request from independent applications.

SMC

The *Streaming Memory Controller* (*SMC*) [17] focuses on real-time steaming traffic, which allows the memory controller to deal with relatively large requests, up to the size of an entire page (1 KB for their device). This is equivalent to choosing BC such that the access granularity is 1 page, with BI 1. Their target device is a DDR1 memory, which we did not consider in this book, but their results relating a larger access granularity to lower power usage are in line with the trends we observed in Chap. 5. Arbitration across streams is done through a credit-based system, which guarantees a certain number of requests within a fixed period is processed for each client. This suggest that the controller is predictable, although a detailed analysis is not provided. It is grouped here with [20] since its large scheduling granularity effectively also gives it a static command schedule, even though it is not precomputed.

8.1.2.2 Semi-Static (Pattern-Like) Command Schedulers

Predator

The Predator controller [18] dynamically schedules precomputed sequences of SDRAM commands (patterns) according to a fixed set of scheduling rules, creating a predictable pattern-based memory controller, implementing a close-page policy. Through a design-time analysis, a *latency-rate* bound [28] on the performance provided to each application is determined. Akesson and Goossens [19] combines the

Table 8.1 Related memory controllers (in chronological order of publication)

Name	1. Target	2. Page policy	3. Device/ DIMM	4. Schedule granularity	5. Burst order	6. Refresh mechanism	7. Predictable/ composable	8. Mapping granularity
[17] SMC	DDR (1)	Close	Device	Static, 1 page	Consecutive columns in a row	Not published	Predictable (analysis is superficial)	1 row
[18, 19] Predator	DDR2/DDR3	Close	Device	Semi-static, 1-n bursts to all banks	Interleave all banks, n-bursts per bank	Internal timer	Predictable, composable with delay block	1 row in all banks
Bayliss et al. [20]	DDR2-533	–	Not published	Static, 1 application	Fully static schedule	Not published	Predictable and composable	Not applicable
[21, 22] AMC, RTCMC	DDR2/DDR3	Close	Device	Semi-static, 1 burst to all banks	Interleave all banks	Internal timer	Predictable	1 row in all banks
[11] PRET	DDR2-400	Close	DIMM	Semi-static, 1 burst to half the banks	Interleave independent resources, interleave ranks	Manual	Predictable within, and composable across independent resources	1 row in an independent resource (2 banks)
Wu et al. [23]	DDR2/DDR3	Open	DIMM	Dynamic	No enforcement	Internal timer	Predictable	1 bank
[12] MCMC	DDR3-1333H	Close	DIMM	Semi-static, 1 burst to half of all banks	Interleave virtual devices, interleave ranks	Manual (like [11])	Composable across virtual devices	1 row in a virtual device (2 banks)
[10] DCmc	DDR2-667	Open	DIMM	Dynamic	No enforcement	Not fully specified. Refers to [14]	Predictable	1 row in a bank
[13] ROC	DDR2/DDR3	Open	DIMM	Dynamic	Interleave ranks	Refers to [23]	Predictable	1 bank

(continued)

Table 8.1 (continued)

Name	1. Target	2. Page policy	3. Device/ DIMM	4. Schedule granularity	5. Burst order	6. Refresh mechanism	7. Predictable/ composable	8. Mapping granularity
[24, 25] RTMemController	DDR3	Close	Device	Dynamic (single request gets consecutive bursts)	Interleave over BI banks with BC bursts per bank. BI and BC may vary per request	Internal timer	Predictable	1 row in BI banks
[26] PMC	DDR3-1333	Hybrid	Device	Semi-static (single request gets consecutive bursts)	Interleave over all banks, 1 burst per bank	Refers to [8]	Predictable	1 row in all banks
[27] (cmd-priority)	DDR2/LPDDR2	Open	DIMM	Dynamic	No enforcement	Internal timer	Predictable	2Ki rows for critical tasks, 1 row in a bank for non-critical tasks
This work (Raptor)	DDR2/3/4 LPDDR1/2/3	Hybrid	Device	Semi-static, BC bursts to BI banks	Programmable BI, BC	Internal timer	Predictable, composable with delay block or composable patterns	1 row in BI banks

controller with a front-end containing delay blocks [15], turning it into composable controller. The combined template is documented in [29], and forms the jumping-off point of the controller presented in this book, as mentioned earlier in Sect. 2.2.

AMC and RTCMC

The *Analyzable Memory Controller* (*AMC*) [21] or, by its new name, *Real-time Capable Memory Controller* (*RTCMC*) controller [22] dynamically schedules commands at run-time, but is effectively semi-static in its worst-case analysis. It interleaves requests over all (4) banks in the devices it considers, issuing one burst per bank (i.e., like a (4, 1) configuration). The work distinguishes HRT and *Non Hard Real-time Tasks* (*NHRT*) (tasks map to clients in our terminology, and NHRT maps to SRT or best-effort). Arbitration across tasks is done through a round-robin arbiter, which first considers all HRTs before the NHRTs. A request from a HRTs can preempt an executing NHRT request by taking over its remaining bursts. The NHRT request is continued once the HRT's request is completed. Both the arbiter type (round-robin) and this preemption mechanism make the controller non-composable, since the presence or absence of requests from competing clients influences the timings of other clients.

The worst-case behavior of the controller is bounded by evaluating what command schedules it uses in the worst case. This analysis is very similar to the pattern-based analysis that we applied in this work, since it essentially involves scheduling access patterns and switching patterns, and then using their (worst-case) concatenated lengths. The controller in [21] has a special mode where request of a task running in isolation can be delayed until their (analytical) worst-case starting time. This mode is used when estimating the WCET of a task by simulation, under the assumption that the processing platform is performance monotonic. However, as illustrated in Sects. 3.4.3.2 and 7.5.2, real applications and systems are not (by default) performance monotonic. It can only be guaranteed by adhering to many restrictions in terms of programming model and system architecture such as done in CompSOC [30], for example.

PRET and MCMC

The *PREcision Timed* (*PRET*) memory controller [11] partitions the SDRAM into pairs of banks which they call *independent resources*, each consisting of two banks in the same rank. The controller executes a static periodic command schedule, in which banks are accessed with one burst at a time (i.e., with a BC of 1), and all independent resources are visited once. Consecutive bursts are guaranteed to be interleaved across different independent resources and ranks, and a close-page policy is used. This paper effectively introduces the concept of *bank privatization* (partitioning), where the independent resources offer composable performance as long as they are not shared.

Another novel feature in [11] is the application of a manual refresh scheme. The controller periodically uses its regular command schedule to activate and precharge a row in each bank, which refreshes those rows, as described in Sect. 2.4.2.1.

PRET uses a relatively slow DDR2 device that has a relatively small read-write switching timing constraint. When combined with rank interleaving, it allows read and write burst to be alternated every 2 bursts without prohibitively large penalties. This does not scale well to faster memories as the read-to-write and write-to-read constraints grow and start to dominate the schedule length. The RRD (ACT-to-ACT) constraint for these memories is also small enough, such that even with $BC = 1$ the efficiency is still reasonable, but this again changes when faster memories are considered. An implementation of the PRET controller on a Xilinx Virtex 5 FPGA is shown in [31].

The *Mixed Critical Memory Controller* (*MCMC*) [12] is similar to PRET in almost every aspect, although the independent resources have been renamed to *virtual devices*. PRET allowed its independent resources to be optionally shared through round-robin arbitration, while MCMC allows two clients, one *critical* and the other *noncritical*, to share one of its virtual devices through a fixed-priority arbiter. The critical client is prioritized over the other noncritical client as it shares a virtual device with. This allows the critical client to have a relatively low WCRT, while slack can be used by the noncritical client. The noncritical client receives non-predictable performance, since it can be blocked by the critical client indefinitely, causing starvation. The critical client receives composable performance.

PMC

The *Programmable Memory Controller* (*PMC*) [26] uses a variation on the conservative open-page patterns we introduced in Chap. 6. The controller generates statically scheduled command sequences, which are very similar to the conservative open-page patterns. In the terminology of [26], patterns are called *bundles*. Bundles effectively implement the access patterns[2] corresponding to the four modes we defined in Chap. 6. Each bundle always interleaves bursts over 8 banks (an (8, 1) equivalent). Our patterns implement just 1 fixed granularity under the assumption all clients produce the same size requests, or are atomized down to this fixed size. Large request from different clients may thus be chopped up and interleaved by the arbiter, destroying their inherent locality, and hence the worst-case analysis has to assume no locality is present. PMC allows large requests to stay mostly intact even after arbitration (up to a certain threshold), preserving the locality. It issues a concatenation of bundles to execute the required commands for such a variable-sized request.

The effect this has is comparable to the MULTI- TDM- 2 experiment in Sect. 6.4.2.3, where we provide more than one consecutive TDM slot to a client, assuming clients generate requests that are larger than an atom. The difference is that, in order to guarantee a request is served as consecutive patterns, it should be fully buffered before the arbiter may schedule it, where previously we only had to buffer an atom worth of data. PMC (presumably) guarantees this behavior, and can hence use the reduced request WCRT this creates in its analysis. The trade-off is that each interfering client

[2]Data bus switches are taken into account separately in their worst-case analysis, but it not clear from [26] if there are bundle counterparts of switching patterns.

potentially occupies the back-end for a longer time in this scheme, proportional to the maximum request size instead of a fixed (typically smaller) atom size. The experiments in [26] show improved bandwidth and reduced latency when requests are very large (512 B and up), while the gains for typical requests (64 B) are modest compared to a solution with close-page patterns.

The name PMC refers to the programmability of its TDM arbiter. Programming is done at boot time, and reconfiguration is not considered. It is combined with a slot-allocation algorithm that guarantees periodicity within the TDM table it generates, allowing them to be stored efficiently, while still implementing a relatively large frame size.

8.1.2.3 Dynamic Command Schedulers

Statically or semi-statically scheduled controllers reduce the number of variables on which the (run-time) command schedule depends. This acts as an abstraction layer for the worst-case analysis, simplifying it. Dynamically scheduled controllers do not do this. Instead, the sequences of commands they can execute emerge directly from the interaction of the SDRAM timing constraints with the incoming request streams. Their worst-case analyses rely purely on knowledge of the behavior that the controller is capable of exhibiting based on its architecture.

Wu and ROC

Wu et al. [23] propose an analysis model for a dynamically scheduled memory controller. The focus is on analysis and not on architecture, but the paper does explain the hardware structure of the memory controller it assumes. Its main feature is that it is an open-page controller using bank privatization. When multiple clients share a bank, their requests are interleaved in an (at design time) unpredictable manner, due to the interaction between the client arbiter and the arrival times of requests, as discussed earlier in Chap. 6. When an open-page policy is used in such a system, each new request finds its bank(s) in a potentially different state than the client left it in, since another client might have used it in the mean time. Any locality information obtained at the client side is hence not usable in a worst-case analysis. However, private banks have consistent state from the client's point of view, which allows [23] to incorporate hit/miss information in the worst-case analysis, when it is available. Benefiting from the use of an open-page policy in terms of lower worst-case bounds is only possible when bank privatization is used, as far as we are aware. RTCMC [22] analyses a similar privatization-based open-page approach, but concludes that giving a complete bank to each application (or thread) leads to scalability issues (the number of banks is limited), and therefore drops the idea in favor of a close-page (4, 1) configuration.

The *Rank-switching Open-row Controller* (*ROC*) [13] builds upon the work in [23], adding an enforced rank-interleaving mechanism. Similar to [11, 12], this (partially) hides the data bus direction-switching overhead of one rank with accesses to another rank.

The underlying assumption in [13, 23] is that hit/miss information can be derived for a client's traffic stream. This requires a bank-aware mapping of data for clients [32], and static analysis or measurements of the request address sequences after address mapping. This analysis may not always be possible, especially if the application is dependent on inputs that are unknown at design time. Another assumption is that distributing the memory capacity (space) across cores at the granularity of banks is feasible, i.e., that over-allocation is limited. Sharing data across cores is also nontrivial in this privatized scheme: when two clients communicate through a shared bank, the worst-case analysis problem for a general open-page policy emerges again. The authors suggest designating a separate set of banks for shared data might be an option.

Rank interleaving is used in [11–13] to limit read-write switching overhead. A hypothetical extension of our controller could apply rank interleaving, for example by ensuring the first burst in a pattern maps to a different rank than the last burst. This would make it easier to satisfy read-write switching constraints across patterns. However, we estimate the benefits of such a scheme would be limited. The reason is that switching patterns do not contribute to the worst-case pattern sequence when a close-page policy is used in 70 % of the configurations that were evaluated in Sect. 5.2, and is hence not relevant for worst-case performance in those cases. In the remaining 30 % of the configurations, the average overhead (in terms of worst-case bandwidth) is 5 %, with a maximum of 16 % for the slowest LPDDR memory. Taking into account that switching ranks has a penalty of about 2 cycles, which replaces a switching pattern with a length of the same order of magnitude, we estimate the efficiency improvement obtained from applying rank interleaving is only 2–3 % on average, within the 30 % of the configurations that would benefit at all. The main reason [11–13] derive benefit from rank-interleaving is their focus on DIMMs. Using more than one burst per request in such controllers is not possible, since a single burst typically fulfills the data needs of a (reasonable sized, i.e. 64 byte) request. Consecutive bursts hence potentially originate from different clients for them, and swap the bus direction every other burst in the worst case, which is very expensive.

RTMemController

The memory controller proposed by [24, 25] uses a close-page policy and uses dynamic command scheduling. The flexibility this offers is used to efficiently serve requests of variable sizes. A (BI, BC) combination for a range of request sizes is stored in a LUT, which is used at run-time to steer the bursts into banks. The selection of BI and BC is based on our work in [33]. Li et al. [24] focuses only on the back-end, while [25] also describes the interaction with a TDM arbiter in the front-end.

Similarly to Predator, RTCMC, PMC, and our work, the RTMemController keeps the bursts that belong to one request (or atom) together as one scheduling unit. However, it works actively toward completing multiple requests at the same time using bank parallelism to pipeline ACT and PRE commands across requests.

The controller is characterized in great detail, and the appropriate worst-case situations are derived in a traceable manner. This forms the basis of the (analytical) WCRT analysis. Li et al. [25] compares the performance of RTMemController with a

(BI, BC)-aware version of Predator, which is comparable to our work. When serving requests of fixed size, the execution times of Predator and RTMemController are found to be identical, while RTMemController deals better with variable request sizes.

DCmc and CMD-Priority

The *Dual-Criticality Memory Controller* (*DCmc*) [10] aims to offer high performance to some clients, while giving real-time guarantees to others. It uses an open-page policy, but when deriving worst-case bounds, it assumes all requests are misses. Banks privatization separates the high-performance clients from the real-time clients. Multiple clients of the same class can share a bank. For high-performance banks, the inter-client arbitration is based on FR-FCFS, while the real-time banks are shared through round-robin arbitration. Commands are generated per bank and forwarded to the memory in a round-robin-like fashion, although commands from real-time banks get priority over the high-performance banks. The WCRT of requests from real-time clients is analytically determined.

The most recent work we discuss here is the *CMD-Priority* controller from [27], which has many similarities with DCmc. Requests are first split into per-bank queues. As the name suggests, this controller incorporates a priority-based arbitration mechanism. Request that are *critical*, can always take a priority slot at the head of the bank queues. Noncritical requests are reordered using a FR-FCFS arbiter, just like in DCmc. Commands in [27] are generated for the requests that are at the head of the queue for each bank. Critical request can overtake noncritical requests midway through their service, for example after their ACT command already has been scheduled but before the associated RD or WR command (the then useless ACT command of the noncritical request has to be repeated later). Banks forward their command to the SDRAM when they are selected by a round-robin-like arbiter, which prioritizes commands from critical requests above noncritical request, just like DCmc does. Both [10, 27] potentially starve their respective high-performance and non-critical clients due to this mechanism. The WCRT analysis for critical requests involves a full search of the possible interfering command combinations and their associated latency as a function of the number of critical clients. Scalability may be an issue, but at least up until 8 critical clients the run-time is reasonable (several hours), as shown in their paper.

Kim et al. [27] compare their work with a version of our pattern-based back-end paired with both a nonwork-conserving and work-conserving TDM arbiter, loosely based on [34]. Since composability is not considered, the work-conserving version is the most appropriate comparison. The patterns that are used have a (8, 1) configuration.[3] WCRT bounds of the two approaches are derived. Kim et al. [27] does slightly better in cases with 1-3 critical clients, while our approach produces slightly smaller bounds for 4-8 critical clients (each client receives 1 slot in the TDM arbiter, and there is a single slot for all non-critical clients). In a trace-based experiment, the average

[3]Based on Chap. 5, a (4, 2) or (2, 4) configuration would be preferable for the DDR2-800 memory that is used.

response time for high-performance clients is smaller for [27]'s controller, which is mainly attributed to use of an open-page policy and the associated FR-FCFS arbitration. We estimate that the conservative open-page version of our controller would be more competitive, although it might still struggle to keep up. The limiting factor is the finite time-window in which locality can be exploited, and the single address windows in which hits can be found. In contrast, [27] has 8 individual windows (one for each bank) and a (practically[4]) unlimited time-window that has at its disposal.

8.1.2.4 Distinguishing Aspects of Our Work

None of the related memory controller we discussed (except [17]) take power into account, despite it being an important design constraint [35]. Also, the (BI, BC) trade-off is usually not explored ([24, 25] of our colleagues is the exception), and the analysis in each of the papers is limited to one or two memory generations, while we show a much broader range of memories and a suitable abstraction to deal with their relative differences. The real-time implications of the introduction of bank groups in DDR4 have also not been evaluated in related work. The conservative open-page policy predates the other works in Table 8.1 that apply open-page policies in a mixed-real-time context.

Furthermore, we have demonstrated our controller in a full system implementation through our FPGA prototype, in contrast to the works in Table 8.1 (except for PRET [11]), which at most show VHDL simulation results. Our controller is integrated into the CompSOC platform [30] and the associated design flow [36], where it offers predictable and composable performance to its clients.

8.2 SDRAM Performance Overviews

A few memory-generation overview papers, comparable to Chap. 5 exist. Micron Technology Inc [37] discusses some of the differences in memory timings across DDR2/3/4, but does not show the effect on worst-case performance. The authors in [38] show the bandwidth/energy efficiency trade-offs for different SDRAM when applied in data centers, but it considers a smaller set of memory generations compared to Chap. 5, and does not focus on worst-case performance. Power is estimated based on Micron's power model, which is less accurate than the DRAMPower model we use [39]. However, they show DIMM level power usage and thus include I/O power in their comparison. We show power at the device level. Cuppu et al. [40] compares several (asynchronous) DRAM architectures and considers SDRAM as one special case within this family, but does not zoom in further. In [41], the focus lies on selecting a suitable memory for a design rather than giving a general performance overview, and it only considers LPDDRx and pre-datasheet WIDE I/O memories.

[4]Banks have to close occasionally for refresh.

8.3 Reconfiguration

Related work on the reconfigurability of memory controllers is scarce. Even though it is quite common for COTS [42] and custom [43] memory controllers to have configurable registers that control SDRAM timings, they are generally only meant to be programmed at boot time, and offer limited control over the behavior of the controller.

The *PARDIS* programmable memory controller [44] is reconfigurable in several respects. Two small processors with custom instruction set architectures take the role of memory controller, their firmware determining the command scheduling policy, address mapping, refresh scheduling and power management. However, no bounds on performance are given, so it is not clear how to apply it in a real-time system. This holds for most best-effort memory controllers.

Focusing on high-level problem of reconfiguration of resources with real-time performance guarantees, two strategies can be distinguished. The first strategy requires knowledge of the frequency of reconfiguration events to analytically bound their interference [45, 46]. The second strategy constrains the reconfiguration process such that the guaranteed performance during reconfiguration is not worse than during regular operation [47].

In [45], task-level WCRT analysis for multimode applications that share resources in multi-core systems is discussed. Mode changes are defined as changes in the set of active tasks or applications, which we refer to as use-cases switches in this book. The resource arbitration mechanism that is used involves software-based critical sections combined with priorities. Interference due to reconfiguration is bounded by limiting the number of simultaneous mode changes to one.

Garcia-Valls et al. [46] presents a reconfiguration method for soft real-time applications. Reconfiguration is interpreted as a change of the active set of tasks, or a change in their configuration in the resources they are using. The hardware offers no performance during reconfiguration, but the reconfiguration time is bounded at design time. This suggests that predictable performance bounds can be derived based on this technique, but it can not be composable, since reconfiguration influences all running applications.

The work presented in [47] describes reconfiguration algorithms for TDM-based servers while guaranteeing schedulability of the client applications. The algorithms assume that server time can be continuously allocated, and by carefully choosing the location of the unallocated server time and the length of transition periods, predictable performance bounds are given. The algorithms are not applicable to composable resources that rely on constant slot times, because the starting time of all slots varies as a result of reconfiguration.

The reconfigurable TDM-based network-on-chip proposed in [48] provides composable performance to selected clients during reconfiguration, basically by not changing their time-slots and the frame size, similar to our approach in Chap. 7. However, the notion of predictable (but not composable) performance during reconfiguration does not exist in [48]. Our TDM arbiter reconfiguration strategy provides an additional predictable performance level with worst-case bounds even during reconfiguration.

In contrast to related work, this book presented a reconfigurable SDRAM controller suitable for mixed time-criticality systems. By programming new patterns into the pattern memory, a suitable (BI, BC) combination can be selected based on the active use-case. We analyzed which configuration parameters can be adapted under the assumption that some clients continuously use the memory controller. Both predictable and composable performance can be offered during reconfiguration when a TDM arbiter is used. The SDRAM resource is modeled as a latency-rate server, and we formally prove that behavior during reconfiguration is not worse than during regular operation.

References

1. Rixner S, Dally WJ, Kapasi UJ, Mattson P, Owens JD (2000) Memory access scheduling. In: International symposium on computer architecture (ISCA), pp 128–138
2. Dodd J (2006) Adaptive page management. US Patent 7,076,617
3. Mutlu O, Moscibroda T (2008) Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems. SIGARCH Comput Architect News 36(3)
4. Stuecheli J, Kaseridis D, Daly D, Hunter HC, John LK (2010) The virtual write queue: coordinating DRAM and last-level cache policies. SIGARCH Comput Architect News 38(3):72–82
5. Ipek E, Mutlu O, Martinez J, Caruana R (2008) Self-optimizing memory controllers: a reinforcement learning approach. In: International symposium on computer architecture (ISCA), pp 39–50
6. 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
7. Yun H, Yao G, Pellizzoni R, Caccamo M, Sha L (2013) Memguard: mnemory bandwidth reservation system for efficient performance isolation in multi-core platforms. In: Real-time and embedded technology and application symposium (RTAS), pp 55–64
8. 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
9. Wilhelm R, Grund D, Reineke J, Schlickling M, Pister M, Ferdinand C (2009) Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. IEEE Trans Comput Aided Des Integr Circuits Syst 28(7)
10. Jalle J, Quinones E, Abella J, Fossati L, Zulianello M, Cazorla F (2014) A dual-criticality memory controller (DCmc): proposal and evaluation of a space case study. In: Real-time systems symposium, pp 207–217
11. 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
12. 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 systems and applications (RTCSA)
13. 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
14. Bhat B, Mueller F (2011) Making DRAM refresh predictable. Real-Time Syst 47(5):430–453

15. Akesson B, Hansson A, Goossens K (2009) Composable resource sharing based on latency-rate servers. In: Digital system design (DSD)
16. 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
17. Burchardt A, Hekstra-Nowacka E, Chauhan A (2005) A real-time streaming memory controller. In: Design, automation and test in Europe conference and exhibition (DATE), pp 20–25
18. Akesson B, Goossens K, Ringhofer M (2007) Predator: a predictable SDRAM memory controller. In: Proceedings of CODES+ISSS
19. 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
20. Bayliss S, Constantinides G (2009) Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search. In: International conference on field-programmable technology, pp 304–307
21. Paolieri M, Quiñones E, Cazorla F, Valero M (2009) An analyzable memory controller for hard real-time CMPs. Embed Syst Lett IEEE 1(4)
22. 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
23. 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
24. Li Y, Akesson B, Goossens K (2014) Dynamic command scheduling for real-time memory controllers. In: Euromicro conference on real-time systems (ECRTS), pp 3–14
25. Li Y, Akesson B, Goossens K (2015) Architecture and analysis of a dynamically-scheduled real-time memory controller. Real-Time Syst 1–55
26. 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
27. Kim H, Broman D, Lee EA, Zimmer M, Shrivastava A, Oh J (2015) A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In: Real-time and embedded technology and application symposium (RTAS)
28. Stiliadis D, Varma A (1998) Latency-rate servers: a general model for analysis of traffic scheduling algorithms. IEEE/ACM Trans Netw 6(5)
29. Akesson B, Goossens K (2011b) Memory controllers for real-time embedded systems, Embedded systems series. Springer
30. Goossens K, Azevedo A, Chandrasekar K, Gomony MD, Goossens S, Koedam M, Li Y, Mirzoyan D, Molnos A, Nejad AB, Nelson A, Sinha S (2013a) Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. SIGBED Rev 10(3):23–34
31. Liu I, Reineke J, Broman D, Zimmer M, Lee E (2014) A PRET microarchitecture implementation with repeatable timing and competitive performance. In: 2012 IEEE 30th international conference on computer design (ICCD), pp 87–93
32. Yun H, Mancuso R, Wu Z-P, Pellizzoni R (2014) Palloc: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In: Real-time and embedded technology and application symposium (RTAS), pp 155–166
33. Goossens S, Kouters T, Akesson B, Goossens K (2014) Memory-map selection for firm real-time SDRAM controllers. In: Design, automation and test in Europe conference and exhibition (DATE), pp 828–831
34. 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
35. ITRS (2011) International technology roadmap for semiconductors (ITRS) - system drivers. http://www.itrs.net/reports.html

36. 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
37. Micron Technology Inc (2014) DDR4 networking design guide introduction. TN-40-03
38. Malladi K, Nothaft F, Periyathambi K, Lee B, Kozyrakis C, Horowitz M (2012) Towards energy-proportional datacenter memory with mobile DRAM. In: International symposium on computer architecture(ISCA), pp 37–48
39. Chandrasekar K (2014) High-level power estimation and optimization of DRAMs. Ph.D. thesis, Delft University of Technology
40. Cuppu V, Jacob B, Davis B, Mudge T (1999) A performance comparison of contemporary DRAM architectures. SIGARCH Comput Architect News 27(2):222–233
41. Gomony M, Weis C, Akesson B, Wehn N, Goossens K (2012) DRAM selection and configuration for real-time mobile systems. In: Design, automation and test in Europe conference and exhibition (DATE), pp 51–56
42. PowerQUICC and QorIQ DDR3 SDRAM controller register setting considerations AN4039 Rev. 4 (2014) Freescale semiconductor
43. Whitty S, Ernst R (2008) A bandwidth optimized SDRAM controller for the morpheus reconfigurable architecture. In: Proceedings of the parallel and distributed processing symposium (IPDPS)
44. Bojnordi M, Ipek E (2012) Pardis: a programmable memory controller for the DDRx interfacing standards. In: International symposium on computer architecture (ISCA), pp 13–24
45. Negrean M, Klawitter S, Ernst R (2013) Timing analysis of multi-mode applications on AUTOSAR conform multi-core systems. In: Design, automation and test in Europe conference and exhibition (DATE)
46. Garcia-Valls M, Basanta-Val P, Estevez-Ayres I (2011) Real-time reconfiguration in multimedia embedded systems. IEEE Trans Consum Electron 57(3):1280–1287
47. Stoimenov N, Thiele L, Santinelli L, Buttazzo G (2010) Resource adaptations with servers for hard real-time systems. In: Proceedings of the tenth ACM international conference on embedded software, pp 269–278
48. Hansson A, Coenen M, Goossens K (2007) Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In: Design, automation and test in Europe conference and exhibition (DATE)

Chapter 9
Conclusions and Future Work

In this chapter, we look back at what we have done in this book. Section 9.1 briefly discussed the motivation behind the work, and lists the main contributions from Chaps. 2–7. Additionally, we provide a slightly broader perspective on the presented content and its relation to the research field in which we operate. This leads to suggestions for future research directions in Sect. 9.2.

9.1 Conclusions

The improvements in CMOS technology enabled the creation of large, and typically power-constrained SoCs that host many different applications. Typically, these applications load and store data on an SDRAM. Since the SDRAM interface is a scarce resource that cannot be replicated without significant expenses, more and more applications share the same SDRAM controller. Resource sharing leads to undesired interaction between otherwise unrelated applications in the form of interference, which changes their performance in unpredictable ways. This is problematic for real-time applications, that require their timing constraints to always be satisfied, regardless of the co-running applications. At the same time, there are applications that use the (same) SDRAM, and benefit from improved average-case (typical) performance. Therefore, a memory controller should deliver sufficient real-time performance, while improving the average case as much as possible.

The mix of active applications using the memory controller changes over time, as they are started and stopped. To effectively deal with these varying requirements, the controller's configuration needs to be adaptable to the different use-cases it is subjected to. At a relatively larger timescale, we observe another type of variation, as SDRAM technology rapidly evolves with the development of new standards. A memory controller should be flexible enough to keep up with these changes, such that the best SDRAM for a specific products can be selected. This book showed what a mixed-time-criticality controller that satisfies these requirements could look like.

Chapter 2 presented an SDRAM controller architecture template, capable of delivering predictable and composable performance. It can hence guarantee that the worst-case or actual-case behavior of real-time applications is unaffected by interference. In some sense, this controller is a major update of the pattern-based memory controller from [1], with a strong focus on enabling reconfigurability and the design of the controller's back-end, which was previously unexplored.

The development of *Raptor*, the instance of the controller which is currently a part of the CompSOC platform [2] and design flow [3], provided valuable insights into the inner workings of SDRAM controllers. Some of these reflected back onto its worst-case performance analysis. For example, the realization that the parallel read/write port on the back-end, in combination with non-equal read and write latencies may lead to cases where service accumulates non-intuitively fast, as shown in Fig. 2.15, is obvious when considering a real implementation, but easily missed in an analysis model. As another example, consider that the PHY introduces a variable latency component as a consequence of the calibration process that ensures byte-alignment at boot-time, and we should take into account in the WCRT. As a memory controller that offers very fine-grained software-controlled configurability of the commands it schedules, Raptor has been used to research SDRAM power consumption [4], timing process variation [5], and retention time [6].

Chapter 3 discussed memory patterns, which are prescheduled sequences of SDRAM commands. These patterns are programmed into in the back-end, and scheduled at run-time by the controller. Patterns serve as an abstraction layer, both at the (worst-case) analysis level, and within the hardware architecture. Both become less complex, since they do not have to consider the low-level command-to-command constraints, but instead deal with (far fewer) pattern-to-pattern constraints.

Chapter 3 introduced a pattern generation heuristic that is straight-forwardly applicable to a large range of contemporary SDRAM generations. To achieve this generality, we proposed a new abstraction layer that can sit before any command scheduling algorithm. It converts SDRAM timings and timing constraints, which are generation specific, into command-to-command constraints, which are not. The scheduling algorithm only has to refer back to these high-level constraints, and can hence remain generation-agnostic. DDR4 is exceptional, in the sense that an architectural change with respect to the other considered SDRAM generations, in the form of bank groups, impacts the command scheduling process. A small modification of our heuristic allows us to exploit this new feature to generate more efficient patterns.

The pattern generation heuristic is parameterized with *Bank Interleaving* (*BI*) and *Burst Count* (*BC*), which describe the mapping of consecutive bursts to the different banks in the memory. Especially the notion that BI is not necessarily either "one" or "all banks of the SDRAM" is innovative with respect to related work. Varying both BI and BC reveals a spectrum of possible command scheduler configurations, and we found that both parameters are useful in describing the behavior of both our own controller and (perhaps unexpectedly) those of others, as demonstrated in Chap. 8. A comparison with an ILP pattern generator showed that our heuristic creates near-optimal results.

In Chap. 4, we introduced DRAMPower, a high-level power model that uses JEDEC-specified current metrics as it input to estimate the power usage of an SDRAM. Like the pattern generation heuristic, the model is generic and can be applied to a wide range of different SDRAM generations. We validated DRAM-Power through measurements on a real SDRAM device, and found it to be more accurate than the commonly used model from Micron [7].

Chapter 5 puts the pattern generation heuristic to work on 12 different memories from 6 SDRAM generations. Combined with DRAMPower, it shows the configuration trade-offs in terms of worst-case bandwidth, power (energy efficiency), and (indirectly) the WCRT as a function of BI and BC for each of these memories. Figure 5.1, which plots worst-case power versus worst-case bandwidth, shows that the shapes of the plotted constellations across different generations are very similar. This indicates that an understanding of the influence of BI and BC on the relative performance of an SDRAM device is useful and generally applicable to identify efficient command scheduler configurations.

We observed that newer memory generations typically use less power and deliver higher worst-case bandwidth. These effects can be attributed to the growth of their data bus width and the increase in operating frequency. The latency components that feed into the WCRT analysis are affected to a much smaller degree. This underlines the memory-wall problem in the context of worst-case performance: handling a series of page misses simply has not gotten much faster over the years.

In Chap. 6, we introduced the conservative open-page policy. From a worst-case perspective, it is a close-page policy, but with a twist: the controller can change its mind about precharging when it is certainly beneficial to keep the row open. Where possible, we substitute auto-precharges by explicit precharge commands to postpone the precharge decision, increasing the size of the time-window in which locality can be exploited. The presented policy ensures that worst-case guarantees are unaffected.

The conservativeness of this approach comes at a price: the latency (stall-time) reduction the policy achieves is modest, and relies on non-blocking processors and cooperation from the client-level arbiter in multi-application scenarios to be effective. When a slight WCRT penalty is tolerable, a speculative policy might be significantly more effective at improving average-case performance, as explored more recently by [8–10], for example.

Finally, in Chap. 7, we investigated how the memory controller's behavior can be adapted when its set of active clients changes, by reconfiguring the various reconfigurable components its architecture exposes. At boot-time, when no clients are active yet, its flexibility definitely pays off in the sense that both the patterns and all arbiter settings can be freely customized for the use-case in which the controller will be used. This means it can be customized for different memories, and for different power/performance operating points (in the form of (BI, BC) configurations) and application sets.

Maintaining predictability and composability for running clients while reconfiguring to accommodate others is significantly harder, and hence only possible in a more restricted fashion. The direct link between the patterns and the memory-map prohibits (useful) changes in the patterns, since they would scramble the data clients stored before reconfiguration. This means we are limited to reconfiguration of the arbiter. We demonstrated a TDM arbiter and associated safe reconfiguration protocol, allowing us to move slots of an active application without violating its *Latency-rate* ($\mathcal{LR}$) guarantees. The memory controller provides predictable performance before, during, and after reconfiguration to the target of the reconfiguration as a result, while composable clients are completely unaffected by the process. Repeating this effort for different (more stateful) arbiters, like CCSP for example, does not seem to be fundamentally impossible, but could be significantly harder.

9.2 Future Work

This section highlights three concrete opportunities for extensions of the pattern-based controller concept, based on the lessons learned in this book.

Dealing with Large Interface Widths

The request size limits the range of usable (BI, BC) configurations, and with that, the attainable memory efficiency, as shown in Chap. 5. If the SDRAM has a large interface width, in cases where a DIMM is used for example, one request may only be big enough to fill a single burst, limiting BI and BC to 1, which is not desirable. Subdividing the interface width of a DIMM into smaller ranks has been considered in the context of power saving [11], and would help to alleviate the issue. Unfortunately, this requires changes in the SDRAM architecture, and is hence not much more than fiction at the moment, so we consider a different solution at the controller level.

A viable strategy could be to share a single pattern amongst multiple atoms at the granularity of banks, for example by scheduling (4, 1) patterns, but filling each burst with a different request (from potentially a different client). This leads to a scheduling scheme that looks somewhat similar to PRET [12] or MCMC [13], the difference being that commands for different banks can still be interleaved within the pattern, as opposed to PRET and MCMC which use a strict TDM arbitration of the command bus across banks. One of the challenges in this scheme is dealing read-write switches, which preferably should happen per pattern as opposed to per burst to reduce their overhead [9].

Conservative Open-Page Policy with Improved Slack Exploitation

The conservative open-page policy might be too conservative for the reasons we highlighted earlier. It is possible to leave banks in an open state at the end of a pattern, if we accept an increase in WCRT. Page hits would result in NANP patterns, while misses would require a new *Precharge and Activate* (*PA*) pattern. The PA pattern is potentially longer than the AP patterns we currently use, since the precharges are no longer executed in a pipelined fashion across pattern boundaries. Other than the effects on the worst-case performance, operation should be relatively similar to the current approach.

The conservative open-page policy tried to improve average-case performance while retaining a clean separation between the front-end and back-end. Letting go of this restriction can result in better average-case performance by borrowing ideas from FR-FCFS arbiters. Pattern lengths are known at design time, so we know exactly how many cycles are saved when a NANP pattern is issued, compared to a PA pattern. This (proven) slack can be used to schedule additional atoms that are also page hits from best-effort clients, as long as the arbiter is made aware of the amount of time that is available.

Verification of the Raptor Instance to the WCRT Model, CompSOC Integration

The $\mathcal{LR}$-based WCRT model that was presented in Sect. 2.5 covers the presented controller architecture template. However, the verification of the model with respect to the Raptor instance of the controller only covers the worst-case bandwidth (Sect. 5.4), but not the WCRT. A logical next step is to characterize the Raptor instance, to extract the various parameters that the model requires, and then experimentally check that the delivered performance is bounded by it. The model can then be integrated in the larger CompSOC platform model [14] to derive (more) accurate application-level WCRT guarantees.

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. 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
4. Chandrasekar K (2014) High-level power estimation and optimization of DRAMs. Ph.D. thesis, Delft University of Technology
5. Chandrasekar K, Goossens S, Weis C, Koedam M, Akesson B, Wehn N, Goossens K (2014) Exploiting expendable process-margins in DRAMs for run-time performance optimization. In: Design, automation and test in Europe conference and exhibition (DATE), pp 1–6
6. Weis C, Jung M, Ehses P, Santos C, Vivet P, Goossens S, Koedam M, Wehn N (2015) Retention time measurements and modelling of bit error rates of wide I/O DRAM in MPSOCs. In: Design, automation and test in Europe conference and exhibition (DATE), pp 495–500
7. Micron (2007) Calculating memory system power for DDR3. Technical report, Micron Technology Inc. TN-41-01
8. Jalle J, Quinones E, Abella J, Fossati L, Zulianello M, Cazorla F (2014) A dual-criticality memory controller (DCmc): Proposal and evaluation of a space case study. In: Real-time systems symposium, pp 207–217
9. 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

10. Kim H, Broman D, Lee EA, Zimmer M, Shrivastava A, Oh J (2015) A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In: Real-time and embedded technology and application symposium (RTAS)
11. Fang K, Zheng H, Lin J, Zhang Z, Zhu Z (2014) Mini-rank: a power-efficient DDRx DRAM memory architecture. IEEE Trans Comput 63(6):1500–1512
12. 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
13. 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 systems and applications (RTCSA)
14. 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

Appendix A
ILP Problem Formulation

Chapter 3 describes a parameterized ILP formulation that can generate read and write patterns. This appendix formalizes the ILP formulation.

A.1 High-Level Goal

The ILP formulation should generate a command schedule that satisfies the following high-level constraints:

1. the schedule is a valid SDRAM command schedule for the memory device under consideration, i.e., it follows the basic SDRAM state machine (first ACT, then RD/WR, then PRE), and the timing constraints within this schedule are not violated, and
2. the schedule can be repeated after itself without violating timing constraints.

We make a distinction between ILP constraints, i.e., the linear equations that are part of the ILP description, and timing constraints, which are defined by JEDEC and specify the minimum distance between pairs of SDRAM commands [1–6], as introduced earlier in Sect. 2.1.2.

A.2 Variables

Which commands are included in a memory pattern is defined by

1. the type of the pattern, i.e., either *read* or *write*. Refresh and switching patterns can be derived later based on the read and write pattern, as described in Sect. 3.2.3.
2. the number of banks interleaved, BI.
3. the number of bursts per bank, BC.

S. Goossens et al., *Memory Controllers for Mixed-Time-Criticality Systems*,
Embedded Systems, DOI 10.1007/978-3-319-32094-6

Each pattern contains BI activates, BI precharges, and BI $\cdot$ BC read or write commands, for read and write patterns, respectively. The ILP problem schedules two incarnations of the pattern, of which the first is complete, and the second is not. The second incarnation consists only of ACT commands and is used to express constraints regarding activate and precharge commands that span across patterns. We use this partial pattern to simplify the formulation. Note that by doing this, we ignore potential constraints related to read and write commands in the first incarnation that influence the second incarnation. This only works if the following two assumptions hold:

- The ACT-to-RD/WR constraint is always greater than or equal to the RD-to-RD and WR-to-WR constraints, which has been true for all memories introduced up to now.
- Switching patterns resolve all remaining constraints across read-to-write and write-to-read patterns boundaries. This assumption also holds since we construct switching patterns for this purpose.

For the purpose of this description, we regard commands as 3-tuples (c_t, c_b, c_n), consisting of a type $c_t \in \{\text{ACT}, \text{RD}, \text{WR}, \text{PRE}\}$, a bank $c_b \in \{0...\text{BI}-1\}$, and an incarnation id $c_n \in \{0, 1\}$. The set of all commands in the pattern is called C. All commands have c_n set to 0, except for the extra ACT commands representing the start of the second incarnation of the pattern. For those commands $c_n = 1$.

$$\begin{aligned}
C_{\text{ACT}} &= \{(\text{ACT}, c_b, c_n) \mid c_b \in \{0...\text{BI}-1\}, c_n \in \{0, 1\}\}\\
C_{\text{PRE}} &= \{(\text{PRE}, c_b, 0) \mid c_b \in \{0...\text{BI}-1\}\}\\
C_{\text{RW}} &= \bigcup_{0 \le i < BC} \{(\text{RD/WR}, c_b, 0) \mid c_b \in \{0...\text{BI}-1\}\}\\
C &= C_{\text{ACT}} \cup C_{\text{PRE}} \cup C_{\text{RW}}
\end{aligned}$$

The algorithm that generates the ILP formulation determines a conservative lower bound (L_c) and upper bound (U_c) on the position of each command c in the pattern, as further explained in Sect. A.3. A set of ILP variables (V_c) is created for each command. It contains one boolean variable X_i^c for each integer position i in the interval $[L_c, U_c)$. If the variable X_i^c is *true*, then this means that the command c is scheduled in position or cycle i:

$$\forall c \in C, \qquad V_c = \{X_i^c \mid i \in \mathbb{Z}, L_c \le i < U_c\}$$

One way to visualize this set of variables is like a matrix, where each column corresponds to a command, and each row to a position in the pattern (see Fig. A.1). The goal of the ILP formulation is to mark exactly one variable in each column as *true*. After 'graying out' the options we know are invalid based on the lower and upper bounds, all remaining variables in a command column are contained in the V_c set of that command.

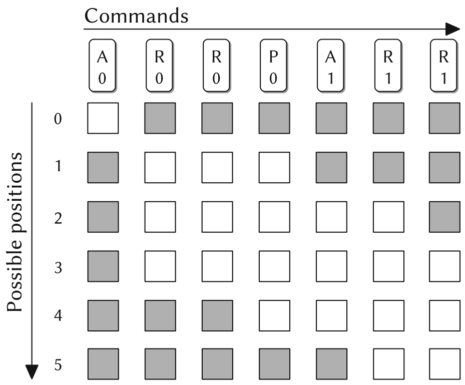


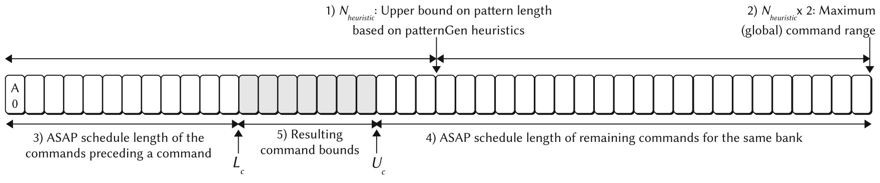
Fig. A.1 Visualization of the ILP variables in matrix form. *Gray boxes* represent variables that cannot be *true* in a valid solution of the problem

The ILP constraints and objective function are constructed as linear equations based on these variables. For ease of notation, we define a function $pos\,(V_c)$, which returns a sub-expression representing the position of a command c in the pattern. The $pos\,()$ function works based on the assumption that only *one* of the variables in each set V_c can be *true* in a valid solution. In Sect. A.4, we show how this is enforced with additional constraints:

$$pos\,(V_c) = \sum_{X_i^c \in V_c} i \cdot X_i^c \qquad i \in \mathbb{Z}, L_c \le i < U_c$$

A.3 Determining Lower and Upper Bounds

It is essential to bound the number of possible positions for each command to reasonable ranges to limit the problem size, and with that, the computation time of the ILP solver. Figure A.2 shows how the bounds of a command are determined. Based on Algorithm 2, an upper bound $N_{heuristic}$ on the optimal pattern length can be found. The ILP problem schedules two incarnations of the pattern (one complete, one partial consisting only of ACT commands), and hence the range of command positions that has to be considered is twice as large as $N_{heuristic}$.



1) $N_{heuristic}$: Upper bound on pattern length based on patternGen heuristics
2) $N_{heuristic}$ x 2: Maximum (global) command range
A 0
3) ASAP schedule length of the commands preceding a command
L_c
5) Resulting command bounds
U_c
4) ASAP schedule length of remaining commands for the same bank

Fig. A.2 Finding lower and upper bounds on the position of a command

We further limit the range on a per-command basis by considering which other commands *have* to precede it based on the pattern-generation rules and the SDRAM state machine, and which commands directed toward the same bank still need to be scheduled after it. Lower bounds on how many cycles it takes to schedule a series of commands are determined based on a very simplistic *As Soon As Possible* (*ASAP*) scheduler. Considering just two commands at a time, it sums the SDRAM timing constraints between each command pair, until it reaches the end of the series. The scheduler fails to spot constraints between commands separated by other commands, and is hence too optimistic, but it is sufficient to quickly find a lower bound on the duration of a sub-schedule. Note that these bounds do *not* have to be tight for correctness of the solution, although the tighter they are, the smaller the computation time of the solver will be.

Tighter lower and upper bounds could be potentially be found by improving this ASAP scheduler, and using $N_{heuristic}$ as a secondary reference point for commands that necessarily happen in the first incarnation of the pattern, further reducing the computation time of the solver. However, the bounds that are currently derived limit the problem size sufficiently such that the largest problems we consider (32 bursts to 8 different banks) are solved within an hour.

A.4 Constraints

This section reiterates the list of constraints from Sect. 3.2.4, and describes how each of them translates into an ILP constraint.

1. *An ACT to bank* 0 *is scheduled in cycle* 0.

$$X_0^c = 1 \mid c = (\text{ACT}, 0, 0)$$

2. *At most one command may be scheduled in each cycle.* Precharge commands are exempted from this rule, since they are executed using auto-precharge flags and do not require a slot in the schedule:

$$\forall i \in \mathbb{Z}, \quad \sum_{X_i^c \in V_c} X_i^c \leq 1 \mid c \in C, c \notin C_{\text{PRE}}$$

3. *Each command is scheduled exactly once*:

$$\forall c \in C, \forall i \in \mathbb{Z}, \quad \sum_{X_i^c \in V_c} X_i^c = 1$$

Based on these constraints, all sets V_c are *Special Ordered Sets* (*SOS*) of type 1 [7], i.e., a set of variables of which at most one can take the value *true*. These constraints are explicitly described as an SOS in the ILP formulation, since this helps guide the ILP solver in finding a solution more quickly. Each variable in such a set needs a relative weight that represents its costs. We assign weights to the elements according to the position (i) within the pattern the element corresponds to.

4. Two of the constraints in Sect. 3.2.4 are related to timing constraints:

   (a) *The relative delays between any pair of commands are at least as large as prescribed by the timing constraints of the SDRAM.*
   (b) *There are at most four ACT commands in each FAW window.*

   The ILP constraints required to implement 4a can be split into two categories: those for which the ordering of the commands involved is irrelevant for the timing constraint, and those for which the order matters. Starting with the first category, the possible command combinations are limited to (1) two activate, (2) two read, or (3) two write commands. Since commands for the same bank are required to happen in a fixed order according to the bank state machine (i.e., activate, then read/write bursts, followed by a precharge), we discard constraints that only apply when the associated commands target the same bank for now, and treat them later when we enforce the command ordering at Constraint 5.
   This only leaves the three previously mentioned pairs, directed at different banks and potentially different bank groups, i.e., six different timing constraints.
   Each timing constraint related to (pairs of) commands of the same type can be interpreted as a *window* in which only a specific number K of those commands may be scheduled. We use this interpretation to capture both the FAW constraint (where $K = 4$) and the narrow set of order-agnostic constraints ($K = 1$) within the same ILP constraint template. The window sizes are equal to the values in the constraint Tables 3.1 and 3.2.
   First, we deal with timing constraints for which the bank group to which the command is sent does not influence the constraint value. For all considered memory types except DDR4, no constraints care about the bank group. For DDR4, in the constraints with an _X postfix, _X is substituted by _L, since the *long* constraints have to be satisfied across all bank groups. The values of the timing constraints between two commands of type tp, $tp \in \{\text{ACT, RD, WR}\}$, are denoted as TC_{tp}. Each TC_{tp} is an integer number of clock cycles, $TC_{tp} \in \mathbb{Z}$.
   A constraint has to be added for all windows of size TC_{tp} in the valid command range, i.e., between 0 and $2 \cdot N_{heuristic}$:

$$\forall j \in \{0..2 \cdot N_{heuristic} - 1\}, \forall\, tp \in \{\text{ACT, RD, WR}\},$$
$$\sum_{X_i^c \in V_c} X_i^c \leq K \mid c \in C, c_t = tp, j \leq i < j + TC_{tp} \qquad \text{(A.1)}$$

   For DDR4, certain timing constraints only need to be satisfied when the associated commands target the *same* bank group. We refer to them as TC'_{tp}, and to the total number of bank groups as nbg. We again iterate over all possible windows, but additionally limit the commands we include in the ILP constraints by their bank group, which is given by their bank id c_b modulo nbg:

$$\forall\, j \in \{0..2 \cdot N_{heuristic} - 1\}, \forall\, bg \in \{0..nbg - 1\}, \forall\, tp \in \{\text{ACT}, \text{RD}, \text{WR}\},$$
$$\sum_{X_i^c \in V_c} X_i^c \le K \mid c \in C, c_t = tp, \quad \bmod (c_b, nbg) = bg, j \le i < j + TC'_{tp} \tag{A.2}$$

In the implementation, constraints that are trivially satisfied because the number of selected X_i^c variables in the equation is smaller than K are not added to the problem description. Figure A.3 shows an example of the variables that are selected for each window based on Eqs. (A.1) or (A.2).

5. *The commands for each bank are executed in the proper order, i.e., start with an activate, followed by BC read or write commands, followed by a precharge, followed by the activate in the second pattern instance.* For each pair of commands c_0 and $c_1 \in C$, which are constrained by a timing constraint $T \in \mathbb{Z}$, and for which the required order is known, such that $pos\left(V_{c_1}\right) > pos\left(V_{c_0}\right)$ in a valid solution of the ILP problem, we add the following ILP constraints:

$$pos\left(V_{c_1}\right) - pos\left(V_{c_0}\right) \ge T \tag{A.3}$$

6. The relative order of commands across pattern incarnations is constrained by the following rules:

   (a) *Commands for the second instance of the pattern cannot be scheduled before the extra activate to bank 0.* This constraint only refers to the activate commands in the second pattern incarnation. They are dealt with in Constraint 7.

   (b) *Commands for the first instance need to be scheduled before the extra activate to bank 0.*

   Since Constraint 5 enforces commands per bank to be ordered, it is sufficient to enforce that the last read or write command to each bank happens before the extra activate command to bank 0. Because Constraint 5 already asserts this property for bank 0, we further limit the set of commands by only including read or writes to banks > 0. We then simply use the template given by Eq. (A.3) on all commands fitting these criteria, substituting them for c_1, and using $c_0 = (\text{ACT}, 0, 1)$ and $T = 0$.

7. *The first and second incarnation of the pattern should be the same.* A set of constraints enforces that the distance between the extra activate command to a bank larger than 0, $pos\left(V_{(\text{ACT},c_b,1)}\right)$, and the start of the second pattern incarnation $pos\left(V_{(\text{ACT},0,1)}\right)$, is equal to the distance between the first activate command to that bank ($V_{(\text{ACT},c_b,0)}$) and cycle 0.

$$\forall\, c_b \in \{1...\text{BI} - 1\}, \quad pos\left(V_{(\text{ACT},c_b,1)}\right) - pos\left(V_{(\text{ACT},0,1)}\right) = pos\left(V_{(\text{ACT},c_b,0)}\right) - 0$$

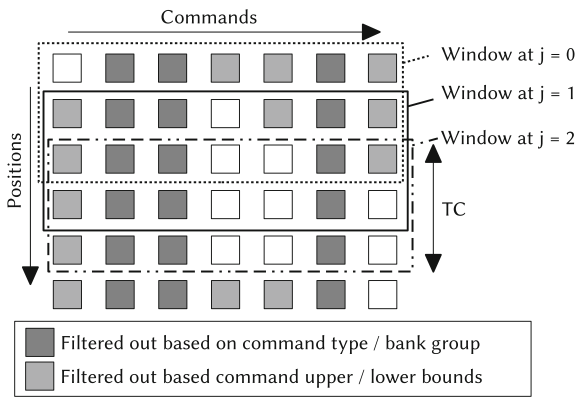

Commands
Window at j = 0
Window at j = 1
Window at j = 2
Positions
TC
Filtered out based on command type / bank group
Filtered out based command upper / lower bounds

Fig. A.3 Visualization of window-based constraints. Only the *white boxes* in each window are included in the sum in Eq. (A.2)

Note that this constraint is stronger than Constraint 6 since $pos\left(V_{(\mathrm{ACT},c_b,0)}\right)$ is guaranteed to be greater than 0, and hence it forces all (remaining) commands of the second instance of the pattern to happen after the second activate to bank 0.

A.5 Objective Function

The objective of the ILP formulation is to minimize the pattern length. This can be achieved by minimizing $pos\left(V_{(\mathrm{ACT},0,1)}\right)$. There may be more than one possible optimal pattern; all commands in the pattern could be postponed as long as the pattern length is not influenced by it. To eliminate some equivalent optimal solutions, we add an extra element to the objective function which attempts to minimize the unnecessary postponement of commands by adding the position of the last precharge in the pattern to the cost function. This makes it easier to visually compare the output to that of Algorithm 2. A helper variable $\mathrm{P\hat{R}E}$ is introduced to represent the position of the last precharge in the pattern. We force it to be greater than or equal to the position of the last precharge in the pattern:

$$\forall c \in C_{\mathrm{PRE}}, \quad \mathrm{P\hat{R}E} - pos\left(V_c\right) \geq 0$$

A sufficiently large scaling factor s is applied to make sure the pattern length remains the primary optimization goal. The optimization goal is then set to

$$\textbf{minimize}\; s \cdot pos\left(V_{(\mathrm{ACT},0,1)}\right) + \mathrm{P\hat{R}E}$$

References

1. JEDEC (2009) DDR2 SDRAM specification JESD79-2F
2. JEDEC (2009) Low power double data rate specification JESD209B
3. JEDEC (2010) DDR3 SDRAM specification JESD79-3E
4. JEDEC (2010) Low power double data rate 2 specification JESD209-2D
5. JEDEC (2012) DDR4 SDRAM specification JESD79-4
6. JEDEC (2013) Low power double data rate 3 specification JESD209-3B
7. Beale EML, Tomlin JA (1969) Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In: Proceedings of the 5th international conference on operations research

Appendix B
Memory Specifications

The following three tables describe the properties of the memories that are used in this book. All the devices we used are made by Micron. The final row in Table B.1 refers to the SO-DIMM of the ML605 development board. The associated timings in Table B.2 for this memory are derived based on a 400 MHz clock. The same timings are also used in experiments when the memory operates at the (lower) 300 MHz frequency in Sect. 3.4.3.2, even though some of them could technically be reduced then. Finally, Table B.3 shows the IDD currents used as input for DRAMPower.

S. Goossens et al., *Memory Controllers for Mixed-Time-Criticality Systems*, Embedded Systems, DOI 10.1007/978-3-319-32094-6

Table B.1 Memory device datasheets

Name	Part number	Datasheet
LPDDR-400	MT46H64M16LF–5	t68m_auto_lpddr.pdf–Rev. E 2 14 EN.pdf
LPDDR-266	MT46H64M16LF–75	t68m_auto_lpddr.pdf–Rev. E 2 14 EN.pdf
DDR2-800	MT47H64M16–25E	1GbDDR2.pdf–Rev. Z 03 14 EN.pdf
DDR2-1066	MT47H64M16–178E	1GbDDR2.pdf–Rev. Z 03 14 EN.pdf
DDR3-1066	MT41J64M16–178E	1Gb_DDR3_SDRAM.pdf–Rev. L 03 13 EN.pdf
DDR3L-1600	MT41K256M16–125	4Gb_DDR3L.pdf–Rev. I 9 13 EN.pdf
LPDDR2-667	MT42L64M32D1–3	2gb_mobile_lpddr2_s4_g69a–Rev. N 3 12 EN.pdf
LPDDR2-1066	MT42L64M32D1–18	2gb_mobile_lpddr2_s4_g69a–Rev. N 3 12 EN.pdf
LPDDR3-1333	EDF8132A1MC–15	178b_30nm_mobile_lpddr3–Rev. A 3 14 EN.pdf
LPDDR3-1600	EDF8132A1MC–125	178b_30nm_mobile_lpddr3–Rev. A 3 14 EN.pdf
DDR4-1866	MT40A512M8–107E	4gb_ddr4_dram–Rev. B 10/14 EN.pdf
DDR4-2400	MT40A512M8–083E	4gb_ddr4_dram–Rev. B 10/14 EN.pdf
ML605 SO-DIMM (DDR3-1066)	MT4JSF6464H–512MB (1G1)	JSF4C64_64x64HY.fm–Rev. B 3/08 EN.pdf

Table B.2 Memory device timings in clock cycles

Name	CCD	CL	CWL	DQSS	DQSCK	FAW	RAS	RC	RCD	REFI	RFC	RP	RL	RRD	RTP	WL	WR	WTR
LPDDR-266	–	3	–	1	–	1	6	9	3	1040	10	3	–	2	–	–	2	1
LPDDR-400	–	3	–	1	–	1	8	11	3	1560	15	3	–	2	–	–	3	2
DDR2-800	–	5	–	–	–	18	16	22	5	3120	51	5	–	4	3	4	6	3
DDR2-1066	–	7	–	–	–	24	22	29	7	4160	68	7	–	6	4	6	8	4
DDR3-1066	–	7	6	–	–	27	20	27	7	4160	59	7	–	6	4	–	8	4
DDR3L-1600	–	11	8	–	–	40	28	39	11	6240	208	11	–	6	6	–	12	6
LPDDR2-667	–	–	–	–	2	17	14	20	6	1300	44	6	5	4	3	2	5	3
LPDDR2-1066	–	–	–	–	3	27	23	32	10	2080	70	10	8	6	4	4	8	4
LPDDR3-1333	–	–	–	–	4	34	28	40	12	2600	87	12	10	7	5	6	10	5
LPDDR3-1600	–	–	–	–	5	40	34	48	15	3120	104	15	12	8	6	6	12	6
DDR4-1866	(4, 5)	13	10	–	–	22	32	45	13	7283	243	13	–	(4, 5)	7	–	14	(3, 7)
DDR4-2400	(4, 6)	16	12	–	–	26	39	55	16	9364	313	16	–	(4, 6)	9	–	18	(3, 9)
ML605	–	6	5	–	–	20	15	21	6	3120	44	6	–	4	4	–	6	4

For DDR4, the short and long timings are shown as a pair (*short*, *long*)

Table B.3 IDD [mA] / V_{DD} [V] parameters for DRAMPower

Name	LPDDR	LPDDR	DDR2	DDR2	DDR3	DDR3L	LPDDR2	LPDDR2	LPDDR3	LPDDR3	DDR4	DDR4
	266	400	800	1066	1066	1600	667	1066	1333	1600	1866	2400
I_{DD0}	70	95	80	90	75	66	20	20	8	8	58	64
I_{DD02}	0	0	0	0	0	0	53	71	63	63	4	4
I_{DD2N}	12	18	30	36	35	32	1.7	1.7	0.8	0.8	44	50
I_{DD2N2}	0	0	0	0	0	0	21	22	28	32	0	0
I_{DD2P0}	0.6	0.6	7	7	12	18	0.5	0.5	0.8	0.8	30	32
I_{DD2P02}	0	0	0	0	0	0	1.7	1.7	2	2	0	0
I_{DD2P1}	0.6	0.6	7	7	25	32	0.5	0.5	0.8	0.8	30	32
I_{DD2P12}	0	0	0	0	0	0	1.7	1.7	2	2	0	0
I_{DD3N}	16	20	35	42	45	47	1.2	1.2	2	2	61	67
I_{DD3N2}	0	0	0	0	0	0	29	30	36	40	0	0
I_{DD3P0}	3.6	3.6	10	10	30	38	1.2	1.2	1.4	1.4	44	44
I_{DD3P02}	0	0	0	0	0	0	4.12	4.12	11.2	11.2	0	0
I_{DD3P1}	3.6	3.6	20	23	30	38	1.2	1.2	1.4	1.4	44	44
I_{DD3P12}	0	0	0	0	0	0	4.12	4.12	11.2	11.2	0	0
I_{DD4R}	110	135	150	180	140	235	5	5	2	2	140	160
I_{DD4R2}	0	0	0	0	0	0	206	226	203	230	0	0
I_{DD4W}	110	135	160	185	155	171	10	10	2	2	156	196
I_{DD4W2}	0	0	0	0	0	0	203	213	213	243	0	0
I_{DD5}	100	100	150	160	160	235	15	15	28	28	190	192
I_{DD52}	0	0	0	0	0	0	136	136	153	153	0	0
I_{DD6}	0.45	0.45	7	7	8	20	1.2	1.2	0.460	0.460	20	20
I_{DD62}	0	0	0	0	0	0	2.6	2.6	1.780	1.780	0	0
V_{DD}	1.8	1.8	1.8	1.8	1.5	1.35	1.8	1.8	1.8	1.8	1.2	1.2
V_{DD2}	0	0	0	0	0	0	1.2	1.2	1.2	1.2	2.5	2.5

Appendix C
Code Listings

Algorithm 6 Pattern-generation helper functions

```
function ACTCYCLES(P)
    // Returns a set of cycles at which ACT commands happen in P
    return { cmd.cc | cmd.type == ACT ∀ cmd ∈ P }
function REMAININGFAWCYCLESAT(i, P)
    actCycles := ACTCYCLES(P)
    if |actCycles| >= 4 then
        // The current FAW started at the 4'th biggest ACT cycle.
        return max(0, FAW − (i − 4THBIGGEST(actCycles)))
    return 0
function FAWSATISFIEDACROSS(pattLen, P)
    // Returns true if the FAW constraint is satisfied across multiple
    // iterations of P. FAW is a constant in clock cycles, its value depends
    // on the memory device.
    if FAW == 0 or pattLen == 0 then
        return true
    if FAW < pattLen then
        // Check the FAW windows that span the end of the pattern and
        // the start of its next incarnation.
        lbRng := { pattLen - FAW, pattLen }
    else
        // Check 1 single FAW window, filled with a wrapping pattern.
        lbRng := { 0 }
    actCycles := ACTCYCLES(P)
    for all lb ∈ lbRng do
        if ACTSINWINDOW(pattLen, lb, lb + FAW, actCycles) > 4 then
            return false
    return true
```

The pattern generation algorithms from Chap. 3 have been implemented in a Python tool. The source code for this tool can be downloaded here: https://git.ics.ele.tue.nl/Public/pypatterngen.

Algorithm 7 Pattern generation helper functions, cont.

```
function ActsInWindow(wrapAt, lb, ub, actCycles)
    // Return the number of act commands in the window [lb .. up].
    // The pattern repeats itself every wrapAt cycles.
    nAct := 0
    for all i ∈ { lb...ub } do
        for all actCycle ∈ actCycles do
            if actCycle == (i % wrapAt) then
                nAct := nAct + 1
    return nAct
```

Algorithm 8 Conservative open-page functions

```
function getFirstPre(P)
    cc := $\infty$
    first := None
    for all cmd ∈ P do
        if (cmd.autoPrechargeFlag or cmd.type == PRE) and cmd.cc < cc then
            cc := cmd.cc
            first := cmd
    return first
```

the DRAMPower tool described in Chap. 4 is available at this url:
https://github.com/ravenrd/DRAMPower.

The data-sets and scripts we used to create Chap. 5 can be downloaded here:
https://git.ics.ele.tue.nl/Public/real-time-sdram-trade-offs.