Embedded Systems

Benny Akesson
Kees Goossens

Memory Controllers for Real-Time Embedded Systems

Predictable and Composable Real-Time Systems

Springer

Embedded Systems

Benny Akesson · Kees Goossens

Memory Controllers for Real-Time Embedded Systems

Predictable and Composable Real-Time Systems

Springer

Benny Akesson
Faculty of Electrical Engineering
Eindhoven University of Technology
Potentiaal/PT 9.11, Den Dolech 2
5600 MB Eindhoven
Netherlands
k.b.akesson@tue.nl

Kees Goossens
Faculty of Electrical Engineering
Eindhoven University of Technology
Potentiaal/PT 9.34, Den Dolech 2
5600 MB Eindhoven
Netherlands
k.g.w.goossens@tue.nl

ISBN 978-1-4419-8206-3 e-ISBN 978-1-4419-8207-0
DOI 10.1007/978-1-4419-8207-0
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011934803

Preface

The authors of this book first met in the context of a master project in Philips Research in Eindhoven, the Netherlands, back in the summer of 2004. One was a student in Computer Science and Engineering from Lund University of Technology in Sweden, and the other a principal research scientist, leading the team of researchers designing a predictable network-on-chip called Æthereal. The work on the network-on-chip had been going on for several years, and the team had realized that predictable system-level guarantees also required a predictable solution for the memory system, which was the topic of the master project.

Inspiration for the memory controller designed in this project was found in two different groups within Philips. One group made statically scheduled SDRAM controllers for high-performance video pipelines with firm real-time requirements, which inspired the idea of precomputed memory patterns. The other group made dynamically scheduled memory controllers for digital TV and set-top boxes, requiring dynamic scheduling to reduce latencies of latency-sensitive memory clients. The memory controller proposed in this book is a hybrid design that combines the approaches of these groups by dynamically scheduling statically computed memory patterns. From these two groups, the authors would particularly like to thank Frits Steenhof and Ad Siereveld for the inspiration and creative discussions.

At the end of the 9 month master project, it was clear that only the surface of a large research topic had been scratched and that much interesting work remained to be done. An opportunity to continue the work in the Electronic Systems group at Eindhoven University of Technology presented itself through a Philips-funded PhD position in the PreMaDoNa project. The bulk of the research presented in this book was carried out in the scope of this project during the following 4 years and resulted in a thesis, several publications, and both a simulation model and a hardware implementation of the memory controller integrated in a design flow. This would not have been possible without the contributions from several master students. Thank you Markus Ringhofer, Eelke Strooisma, Getachew Teshome, Williston Hayes, and Winston Siauw for your hard work and for all the good times. The authors also

express their gratitude to Prof. Lambert Spaanenburg and Prof. Jef van Meerbergen for being the key enablers of the master project and the PhD project, respectively.

During the PhD project, parts of Philips Research turned into NXP Semiconductors, and the research on the Æthereal network-on-chip finished. The fruits of this research were combined with processor tiles featuring Silicon Hive VLIW cores and MicroBlaze cores, resulting in the CoMPSoC platform and design flow. This effort demonstrated that it was possible to design a predictable and composable platform capable of concurrently executing a mix of real-time and non-real-time applications. Since then, this platform has been extensively used both as a research vehicle and for embedded system education at Eindhoven University of Technology, Delft University of Technology, and NXP Semiconductors.

For every door you close in research, another one opens. Both authors of this book are currently employed at Eindhoven University of Technology extending the work on the CoMPSoC platform and the predictable and composable memory controller. The authors express their gratitude to all the members of the CoMPSoC team, and in particular to those contributing to the memory controller research. Thank you Karthik Chandrasekar, Firew Siyoum, Anand Khot, Sven Goossens, Tim Kouters, and Manil Dev Gomony, for continuing to push the boundaries of real-time memory controllers.

Hard and passionate work takes its toll and sacrifices evenings and weekends when required. The authors are grateful for the support from family and friends, in particular for the friendship and help from Andreas Hansson. Finally, Benny acknowledges the support of his wife María Eugenia Martelli, whose love and understanding made this journey easier and more enjoyable.

Eindhoven, the Netherlands Benny Akesson and Kees Goossens

Intended Audience

This book is generally intended for readers interested in Systems-on-Chips with real-time applications. It targets senior architects and engineers, as well as academics (both teachers and students), looking for a perspective on the design and use of memory controllers in these systems. It is especially well-suited for readers looking to use SDRAM memories in systems with hard or firm real-time requirements. The book provides enough background on memories and memory controllers to be self-contained and provides extensive background references for the interested reader. There is a strong focus on real-time concepts, such as predictability and composability, and only a brief discussion about memory controller architectures for high-performance computing.

The reader learns step-by-step how to go from an unpredictable SDRAM memory offering highly variable bandwidth and latency to a predictable and composable shared memory providing guaranteed bandwidth and latency to isolated applications. This journey covers concepts for making memories and arbiters behave in a predictable and composable manner, as well as architecture descriptions of hardware blocks that implement the concepts.

The book will appeal to readers with different levels of prior knowledge and different learning goals:

Novice/student & teachers: learns/teaches about system verification trends and challenges, the general architecture of SDRAM and memory controllers, and the problem of using these in Systems-on-Chips with real-time applications.

Practitioner: learns about concrete concepts, architectures, and implementations for predictable and composable memory controllers.

Expert: learns about complete design philosophy and concepts for predictable and composable memory controllers that are directly applicable to different system approaches, such as time-triggered architectures, precision-timed architectures, and different platforms, such as MERASA and CoMPSoC.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

People in modern society are surrounded by computers. This is very impressive, considering that the electronic computer was a rare and simple calculator the size of a house little over half a century ago. Since then, we have seen an amazing development that turned these machines into computational marvels that contribute to most aspects of our daily lives. Computers became faster and cheaper, and found their way into our homes. They also became smaller and more energy efficient, resulting in portable laptop computers that accompany us when traveling. However, the majority of computers in our daily lives are not the general personal computers we use at work, school, or in the office. Instead, these are the embedded systems that are built for a particular purpose, such as our mobile phones, MP3-players, televisions, DVD-players, and navigation systems. Examples of embedded systems outside the consumer electronics domain involve the many computers inside washing machines, cars, and airplanes. The impressive development of embedded systems is not without drawbacks. As systems become increasingly powerful and integrate more and more functionality, they also become more difficult to produce. More advanced devices consist of more hardware and software components that must be designed, integrated and verified. To stay ahead of the competition, companies have to design these complex systems in a very short time [54]. A particular challenge with embedded systems is that their applications often have timing requirements and must produce the right result at the right time to prevent quality degradation or even system malfunction. These timing requirements provide the high-level problem addressed in this book.

We begin this book in Sect. 1.1 by discussing trends in embedded system design, followed by an introduction to the intended application domains and considered platforms. We then explain the problem of mapping applications on platforms and verifying that all timing requirements are satisfied. This results in the problem statement of this book, presented in Sect. 1.2, which focuses on these issues in a main system component: the memory controller. Section 1.3 explains how predictability, abstraction, composability, and automation reduce the mapping and verification effort of embedded systems, and introduces them as requirements on our

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_1, © Springer Science+Business Media, LLC 2011

solution. Section 1.4 briefly then introduces the CoMPSoC platform, which provides the primary system context for the proposed memory controller. The contributions of this work are summarized in Sect. 1.5, before we present an outline of the rest of the book in Sect. 1.6. Lastly, the contents of this chapter are summarized in Sect. 1.7.

1.1 Trends in Embedded System Design

This section discusses some general aspects of embedded system design to create understanding for the different steps and the complexities involved in designing the embedded systems that surround us in our daily lives, such as smart phones and navigation systems. Challenges are highlighted, as well as past and current trends to help us extrapolate future problems in the field. The contents of this section revolve around the example embedded system design flow shown in Fig. 1.1. The first part of the discussion considers applications, which are one of the starting points of the design process and the input to the partitioning step in the design flow.

1.1.1 Applications

The functionality provided by an embedded system is determined by its applications. An application is an independent program that performs a well-defined function for the user, such as playing audio or video content. Trends show that the amount of application software in embedded systems is rapidly increasing [54]. This evolution towards systems with more and more functionality is visible in both the consumer electronics and the automotive domains. Already a decade ago, it was shown that the amount of software in high-end consumer electronic products, such as televisions, video recorders and stereo sets, increased exponentially with

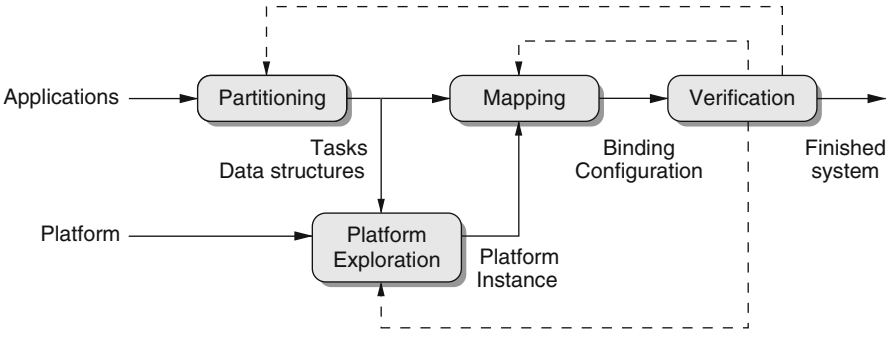

Fig. 1.1 Example design flow comprised of a partitioning, platform exploration, mapping, and a verification step

an annual growth rate of about 40% [27]. Currently, convergence in application domains causes the number of applications in consumer electronic and mobile devices to increase. A prime example of this development is that the functionality of previously separate devices, such as MP3 players, movie players, cell phones, digital cameras, game consoles, and personal-digital assistants, are all coming together in a single hand-held device, called a smart phone. The large number of applications in these devices covers a vast space from multimedia decoding to Internet and gaming [54]. As a result of this trend, the computational load of smart phones grows exponentially and doubles every 5 years [125]. A similar trend of increased functionality is also visible in the automotive domain, although for different reasons. Traditional automotive systems have been implemented as federated architectures. This means that applications, such as engine control system, braking system, and multimedia system, are mapped on nearly autonomous distributed application subsystems, consisting of electronic control units (ECU), networks, sensors and actuators. A state of the art car is a complex distributed system with up to 70 ECUs [97]. For cost, dependability and weight reasons, there is a transition towards integrated architectures, where multiple applications share a common hardware base [30]. Future automotive systems are hence also expected to be highly integrated systems, executing many applications.

Apart from being functionally correct, applications may also have different types of *real-time requirements*. Some applications have *latency requirements*, which means that the result of certain computation must be finished within a specified time, called a deadline. This type of requirement is common in control applications that need to react quickly to incoming events. Other applications are pipelined and have *throughput requirements* instead of latency requirements. In this case, it is less important how long it takes to perform the pipelined computation, as long as a result is being produced often enough to sustain the required throughput. Examples in this category are real-time streaming applications, such a video decoder that must be able to present a new video frame on a television screen with a frequency of 100 Hz. This means that a new image must be displayed on the screen every 10 ms. The time to decode a frame may, however, be greater than 10 ms if the decoding process is pipelined.

Real-time requirements exist in a number of different classes. In this work, we distinguish three such classes [17], being *hard real-time requirements*, *firm real-time requirements*, and *soft real-time requirements*. Applications with hard real-time requirements are often *safety critical* and are primarily found in the health-care, automotive and aerospace domains. The real-time requirements of hard real-time applications, such as the brake system in a car, must *always be satisfied* to ensure safety of the passengers. To guarantee that hard real-time requirements are satisfied even in the presence of hardware failure, some architectures even include redundant hardware. Some applications, such as a Software-Defined Radio [86], have firm real-time requirements. Missing a firm deadline is *highly undesirable* and may result in failure to comply with a given standard, and may even violate the functional correctness of the System-on-Chip (SoC) [37, 117]. Firm real-time requirements, unlike their hard counterpart, are not safety critical, and costly measures, such as

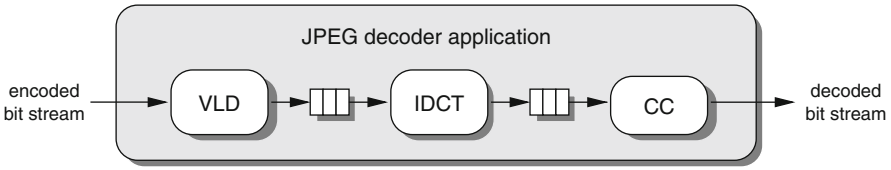

Fig. 1.2 A JPEG decoder application consisting of three tasks

hardware redundancy, are not taken to exclude the possibility of missing a deadline. This type of requirement is hence more prevalent in domains where applications are not safety-critical, such as consumer electronics. The temporal behavior of soft real-time applications, such as media decoders, are not critical to preserve the functional correctness of the SoC. Missing a soft deadline results in *quality degradation* of the application output, such as causing visual artifacts in decoded video or clicks in audio playback. Although this is perceived as annoying by the user, it may be acceptable as long as it does not occur too frequently [1]. There are also applications without real-time requirements, such as a JPEG decoder or a graphical user interface. These applications do not have any timing requirements, but must still execute fast enough to be perceived as responsive by the user.

The partitioning step in Fig. 1.1, partitions applications into smaller *tasks* that communicate through shared data structures. The JPEG decoder in Fig.1.2 is an example of a partitioned application. It is partitioned into three communicating tasks, being variable-length decoding (VLD), inverse-discrete cosine transform (IDCT), and color conversion (CC). The reason to partition an application is to enable parallel execution by binding the tasks to different Processing Elements (PEs) and the shared data structures to memories. This allows computations to be done faster, increasing application performance if the overhead of communication and synchronization is limited [56]. This has been demonstrated for the example JPEG decoder in [43].

Multiple applications may execute at the same time and we refer to a set of concurrently running applications as a *use-case*. The number of use-cases in a system varies greatly, but is growing rapidly and is already in the hundreds for high-end televisions. This impressive growth is intuitively understood by considering that the number of possible use-cases in a system *increases exponentially* with the number of applications. Applications can be dynamically started and stopped at any time, triggering a *use-case transition*. This is shown in Fig. 1.3, where five use-cases are created as three applications start and stop their executions.

1.1.2 Platform-Based Design

Technological advances in the semiconductor industry continuously increase the achievable density of very large-scale integrated (VLSI) circuits [27]. This

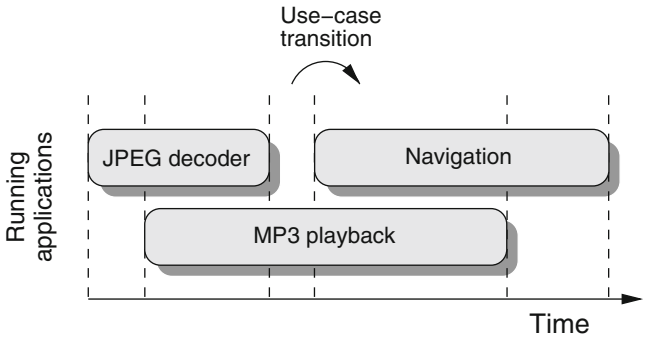

Fig. 1.3 Starting and stopping applications triggers use-case transitions

development has followed a trend known as Moore's law [84, 85] for more than four decades. Moore's law predicts that the number of transistors that can be integrated on a chip will double every 24 months. This prediction remains valid today and is considered a self-fulfilling prophecy, as the semiconductor industry strives towards its continuation.

Previously, a system was distributed over multiple chips connected on a printed circuit board. However, the increasing transistor density has enabled more and more components to be integrated on a single chip. This has resulted in a transition towards SoC solutions, where an entire system is implemented on a single chip. This development has not only reduced the size of the resulting systems, but also power dissipation and ultimately cost [108]. The increasing transistor density has many advantages and paved way for many of the complex embedded systems we enjoy today. However, the benefits of Moore's law do not come without their share of associated challenges. One of the most prominent challenges concerns design productivity [21]. According to Moore's law, the number of transistors on a chip doubles every 24 months, corresponding to an annual increase of 40%. In contrast, the hardware productivity of VLSI designers only increases annually with 20% [108]. This results in an exponentially increasing *design productivity gap*, as illustrated in Fig. 1.4. A consequence if this trend is that designers are unable to make efficient use of the additional transistors provided by developments in process technology without just replicating regular structures, such as memories. Resolving this gap has been identified as one of the grand design challenges for the near future in the International Technology Roadmap for Semiconductors (ITRS) [59].

The design productivity problem has led to adoption of *reuse methodologies*, where pre-designed and pre-verified components are reused between products [108]. However, productivity gains from reusable Intellectual Property (IP) components alone are not enough to close the productivity gap and reduce cost, due to the large associated integration effort. Additionally, a *platform-based design* approach has been proposed that promotes reuse at a higher level of abstraction [30, 108]. A platform comprises a set of hardware and software components, specific to

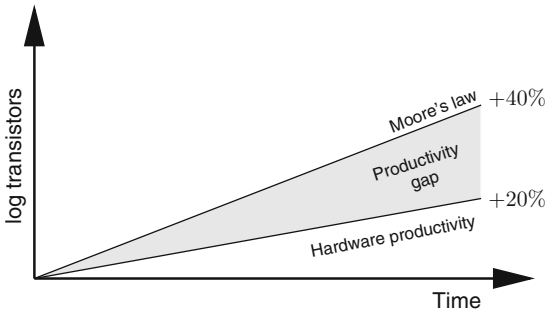

Fig. 1.4 The design productivity gap

a particular application domain. The platform software is not application code, but rather middleware (software for hardware), operating system, and compilers, required to program the platform. This may hence involve operating system kernel, hardware drivers, communication and synchronization libraries, and resource managers. The purpose of the platform is to serve as a starting point for products in the intended domain and differentiation is achieved by integrating additional components, either in hardware or software [135]. Which components to add are determined during the platform exploration step in Fig. 1.1. The purpose of this step is to find a suitable *platform instance* for the tasks of the considered applications that satisfies all design requirements. A drawback of reusing platforms across an application domain is that the resulting designs are slower and more expensive in terms of area and power than customized solutions. The reason is that the platform is more general than what is required for a particular design and may be slightly over-designed to leave room for future products [67]. On the other hand, platform-based design increases design productivity and reduces time-to-market, resulting in increased revenue.

In the past years, platforms for embedded systems have been progressing towards multi-processor systems-on-chip (MPSoC) architectures. This transition is motivated by diminishing returns from instruction-level parallelism, and that it is no longer possible to increase performance of a processor by increasing the clock frequency, due to power and thermal constraints [2, 35, 53, 63]. As an alternative to increasing performance of a single processing element, parallel execution uses multiple processing elements that run at a lower clock frequency, reducing power consumption [35, 137]. This has prompted industry to move towards exploiting task-level parallelism by executing tasks on multiple processors [35, 53, 109, 119]. This trend is well-known and has also been observed in many homes, since most personal computers, both stationary and portable, are now shipped with up to four processors on a single die [35, 53]. Similarly, the number of processors in SoCs in both consumer electronics [68] and mobile phones [125] are increasing with every generation. However, the required processing power in portable consumer SoCs is expected to increase with three orders of magnitude over the next 10 years, while power consumption must remain largely unaffected to maintain battery life time [59]. To deliver on this expectation, we require highly parallel heterogeneous

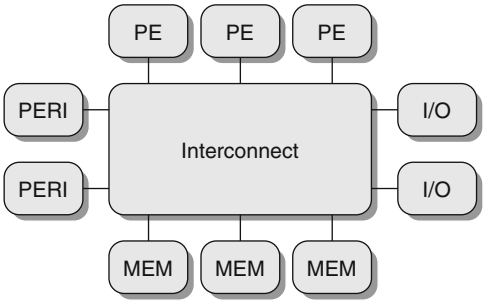

Fig. 1.5 The platform template considered in this book

platforms with a single or a few general purpose processors and many processing elements, to strike a good balance between performance, cost, power consumption and flexibility [35, 42, 54, 59, 63, 119, 125, 137]. Processing elements in this context correspond to application-specific processors or hardware accelerators that realize computationally intensive functions in hardware at low cost in terms of area and power. The general purpose processors and the peripherals used in these architectures are expected to maintain constant complexity over time. However, ITRS indicates that the number of processing elements on a chip will increase by an order of magnitude over the next 10 years [59], pushing parallel computing to its limits. The combination of more processing elements and increasing heterogeneity results in an overall trend towards *increasing system complexity* that is expected to persist in the coming decades.

1.1.3 Platform Architecture

In Sect. 1.1.1, we mentioned that the number of applications in embedded systems is increasing. We then explained in Sect. 1.1.2 how increased customer demand for more applications and pressure to reduce cost and time-to-market caused embedded systems to move from being single-processor designs to being based on reusable heterogeneous multi-processor platforms. In this section, we discuss what the architectures of these platforms may look like. The discussion revolves around a general architecture template, shown in Fig. 1.5. The considered architecture template applies to industrial heterogeneous multi-processor platforms, such as NXP's Nexperia [32, 42, 68], STI's Cell Broadband Engine [63], BroadCom MediaDSP [109], STMicroelectronics' Platform 2012 [119], and Texas Instruments OMAP [42].

Based on the design trends explained in Sect. 1.1.2, we consider a platform architecture that consists of many Processing Elements (PEs). The processing elements in a platform typically consist of one or a few general-purpose RISC processors, such as ARM [13] or MIPS [81] cores. These processors orchestrate the execution on the platform by starting and stopping applications, and configuring

components during use-case transitions. It is also possible that some of these are high-performance processors that are used to speed up execution of code that is either legacy or inherently sequential [56]. The bulk of the computation in the platform is carried out by a large number of application-specific instruction-set processors, such as Digital Signal Processors (DSPs), vector processors, or very-long instruction-word processors, targeting a particular application domain. However, they may also be hardware accelerators, efficiently implementing a single computationally intensive function, such as a Fast-Fourier Transform or inverse-discrete cosine transform.

Apart from processing elements, the platform also contains memories. There are often many different types of memories, representing different cost and performance trade-offs. On-chip Static RAMs (SRAMs) are often used to store instructions or data local to the CPUs and PEs, either in form of caches or scratchpads. Being on-chip, SRAMs have the benefit of being faster to access than off-chip memories, but they are often limited to less than a megabyte (MB) to reduce cost. In addition to local memories, there are centralized memories (MEM) that are typically shared by multiple processing elements. SRAMs may be used to implement these centralized memories, especially if local memories cannot be accessed by remote CPUs or PEs. However, many platforms have a central interface to an off-chip Synchronous Dynamic RAM (SDRAM). An advantage of SDRAMs is that a memory cell is implemented with a single transistor and a capacitor, as opposed to the six transistors required by an SRAM. SDRAMs are furthermore manufactured in large volumes in an optimized process technology. Together, these factors allow them to provide a large storage capacity, up to several gigabytes (GB), at relatively low cost per bit. This makes SDRAMs an important component in any cost-sensitive SoC with applications using large data sets, such as video decoders. Both SRAMs and SDRAMs are volatile memories, which means that they lose the stored data whenever they are switched off. For this reason, it is common to also have non-volatile memory to store instructions and data required to boot the system. These days, this is most commonly done using flash memories. Finally, the platform contains peripherals (PERI), such as mice, keyboards, speakers and displays, and I/O devices providing connectivity to other systems. Common types of connectivity involve USB, UART, HDMI, PCI, I^2E, or Ethernet.

Communicating components are connected using an interconnection fabric that can be direct wires, switches, or buses. Decreasing feature size has created a need for multi-hop interconnects, since it is not always possible to cross a chip in a single clock cycle. Complex SoCs hence require bridged buses or networks-on-chips [29], which are multi-hop interconnects that allow multiple transactions to be served in parallel.

The different hardware components, i.e. processing elements, memories, peripherals, I/O devices, and interconnect, may run at different clock frequencies. This is required either to achieve different power and performance trade-offs using dynamic voltage and frequency scaling, or because the maximum clock frequency of a component is limited. To cope with different clock frequencies, communicating

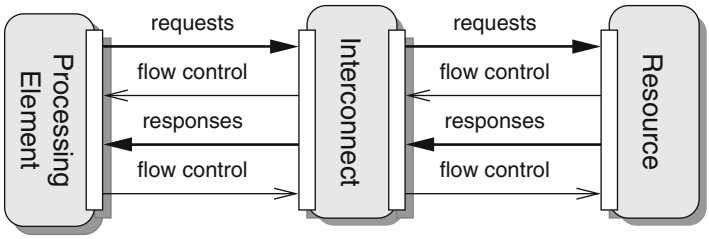

Fig. 1.6 Processing element and resource communicating via a standard protocol

components are bridged using a clock domain crossing, typically implemented using asynchronous first-in-first-out (FIFO) buffers. The considered system is hence globally-asynchronous locally-synchronous (GALS) [88].

IP components in the architecture communicate by sending read and write transactions on ports. The transactions consist of requests and responses, as shown in Fig. 1.6. The components communicate using a protocol, such as the Device Transaction Level (DTL) protocol [101] used by Philips and NXP, or Advanced eXtensible Interface (AXI) protocol [14] promoted by ARM. These protocols often feature a flow-control mechanism, as illustrated by the flow-control signals in Fig. 1.6. This mechanism is typically implemented by a two-phase valid / accept handshake between the sender and receiver. The benefit of flow control is that it allows a receiving component to stall the sender if it is not ready to accept a request or a response, which is useful to prevent a buffer overflow, or to implement clock domain crossings. Throughout the figures in this book, standard DTL/AXI ports are colored white, while grey ports indicate other types of interfaces.

Resources, such as memories and peripherals, are often shared between multiple processing elements, since area, power and pin constraints prevent them from being duplicated. If a resource is shared, arriving requests are stored in a Request Buffer, located in front of the resource. Access to the resource is provided by a bus, controlled by a resource arbiter. The resource processes the request and stores a response in the Response Buffer of the corresponding processing element when it is finished. This is illustrated in Fig. 1.7. Contemporary platforms contain a large variety of resource arbiters with different properties. One common example is Time-Division Multiplexing (TDM), which shares the resource in time among the processing elements according to a fixed periodic schedule. An advantage of this arbiter is that the service provided to a processing element is known at design time and is completely independent of others. Another example is Round-Robin arbitration [89], which cycles between processing elements trying to access the resource. This arbiter tries to be fair by treating all processing elements equally. In contrast, a static-priority arbiter provides differentiated service by always scheduling the processing element with the highest priority. This enables low latency to be provided to applications with tight deadlines, while applications with loose deadlines, or no deadlines, access the resource with a longer latency.

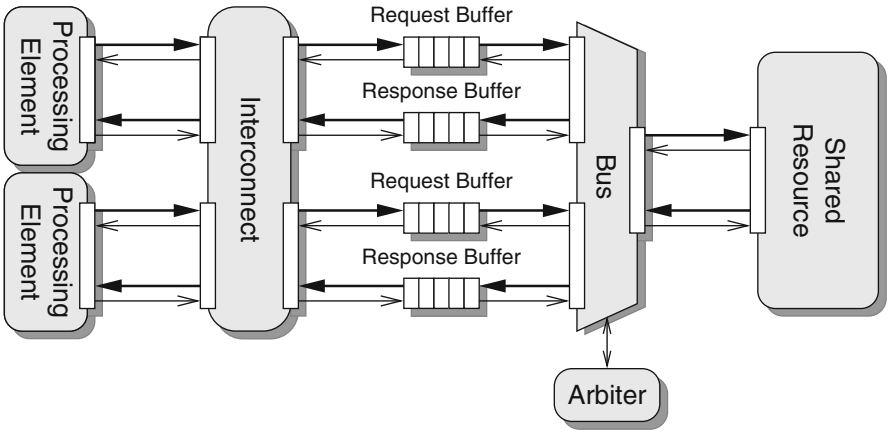

Fig. 1.7 Multiple processing elements sharing a resource

1.1.4 Mapping

Mapping is the process of binding applications to the platform instance, such that all functional and non-functional requirements are satisfied. The mapping process hence takes place after applications have been partitioned and a suitable platform instance has been found, as shown in Fig. 1.1. The mapping process consists of two parts. The first part deals with binding tasks and data structures to IP components in the platform instance, and the second with computing IP parameters and configurations. We proceed by discussing these steps and their associated challenges in more detail.

In the binding step, all tasks are assigned to processing elements, and shared data structures to either local or centralized memories. This process is illustrated in Fig. 1.8, as the JPEG decoder application is mapped on an instance of the considered platform. The three tasks are mapped to different processing elements and the buffers for inter-task communication are mapped in centralized SRAMs. The encoded bit stream is read from an SDRAM and the decoded output is written to a display controller. The binding is a non-trivial problem, since processing elements have different performance and power consumption and memories have different capacities and access latencies. This results in a large design space that grows with the increasing system complexity, as more and more components are added to SoC platforms [59]. However, there are no industrial-strength tools that automatically derive suitable bindings, leading to that the embedded system industry often performs this step manually. Fortunately, the scope of the problem is somewhat mitigated by the increased specialization of processing elements in heterogeneous platforms. A particular implementation of a task may hence be limited to a subset of the processing elements, or even to a single core [68, 125]. Imagine, for example,

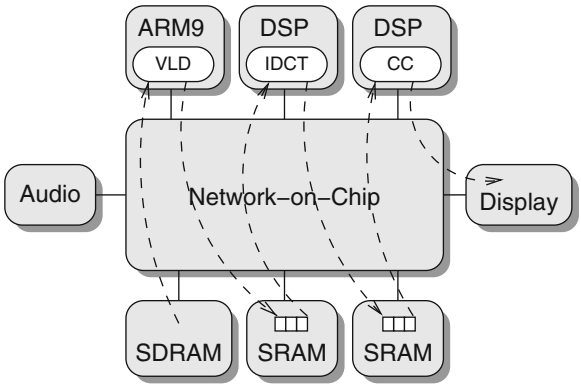

Fig. 1.8 Tasks are mapped to processing elements, data structures to memories, and communication channels to the interconnect as a part of the mapping process

if an IDCT task has to be mapped to a platform and an implementation is available as highly optimized C-code for a particular type of DSP. In this case, the binding is limited only to DSPs of this type unless alternative implementations are developed. Once a satisfactory binding is found, the bandwidth and latency requirements for all resources, such as interconnect and memories, can be derived. In this book, we use the term *requestor* to represent a component that performs resource access on behalf of an application. This corresponds to a port on a processing element connected to the resource through a communication channel. A partitioned application is hence associated with multiple requestors with requirements that may be very diverse in terms of bandwidth, latency, and real-time classification.

The second part of the mapping process is computing parameters and configuration settings for all IP components, such as memory controllers, interconnect and arbiters. IP parameters, such as buffer sizes, are used to instantiate components at design time. Configuration settings, on the other hand, may be different per use-case and are programmed at run time. Finding these parameters and configuration settings is challenging, since all bandwidth and latency requirements of the requestors must be satisfied for all use-cases. In practice, parameters and configuration settings are often determined by trial-and-error using simulation-based techniques [54, 126]. Transaction-level models (TLM) that capture the temporal behavior of the system may be used to speed up simulations [42], making the search for appropriate parameters more feasible, possibly at the expense of accuracy. Simulation-based techniques are predominant over analytical approaches, since the impact of changing the configuration settings on the bandwidth and latency of a requestor is often not well understood. This problem is particularly difficult when there are multiple arbiters, often with different characteristics, interacting in the platform [125]. The configuration step is expected to get increasingly difficult as more and more heterogeneous components, executing increasingly diverse concurrent applications, are added to the platforms.

1.1.5 Verification

The purpose of the verification process is to assert that a system meets its specification and hence that all application requirements are satisfied. The verification process starts when a mapping has been determined in the mapping stage, as shown in Fig. 1.1. The mapping is considered successful if all application requirements are satisfied. Otherwise, if verification fails, it is time to consider a different task partitioning, a different mapping, or a different platform instance, as indicated by the dashed back-arrows in the figure.

Verification is typically done by system-level simulation of the applications executing on the platform instance. The simulation speed of a complete system is very slow. For this reason, verification is sometimes performed using transaction-level models of the components, enabling the accuracy of the verification to be traded for increased simulation speed. Simulation-based verification of real-time requirements is complicated by resource sharing, which causes *scheduling interference* between requestors, as they have to wait for each other before accessing the resource. Interference makes the temporal behavior of concurrently executing applications inter-dependent, resulting in three problems. The first problem is that it is not sufficient to verify that the requirements of each application are satisfied when executing individually. Instead, all concurrently executing applications have to be verified together for all use-cases, causing the verification complexity of the system to *increase exponentially* with the number of applications [44]. However, system-level simulation of all use-cases is far too slow to be feasible in practice. As a result, industry often resorts to reducing the coverage and verifying only a subset of use-cases that have the tightest requirements [42, 117]. The second problem is that verification of a use-case cannot begin until all applications it comprises are available. Timely completion of the verification process hence depends on the availability of the applications, which may be developed by different teams both inside and outside the company. The last problem with application dependencies is that use-case verification becomes a *circular process* that must be repeated if an application is added, removed, or modified [69]. Together these three problems contribute to making the integration and verification process a dominant part of SoC development, both in terms of time and money.

An alternative to simulation-based verification is to analytically verify that requirements are satisfied using a formal performance analysis framework, such as network calculus [28] or data-flow analysis [113]. These frameworks can be used to derive hard performance guarantees on latency or throughput of an application, provided that worst-case execution times of its tasks are known. Firm performance guarantees, on the other hand, can be analytically derived based on execution time estimates. However, in this case it is important to know the quality of the estimates and the assumptions under which they are valid. Formal methods are not necessarily faster than simulation-based techniques, considering that the run-time of mapping and verification algorithms can be very long. Formal methods do, however, guarantee coverage of all possible initial states, input sequences, and interactions

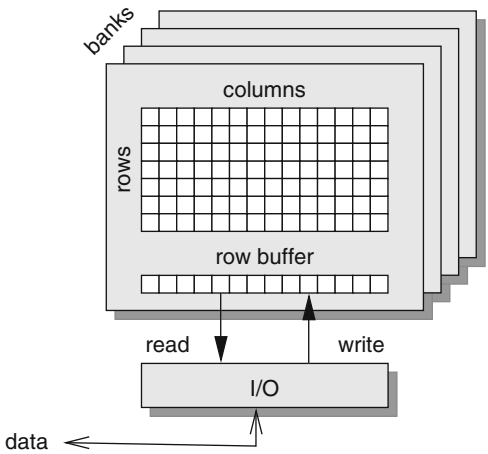

Fig. 1.9 The SDRAM architecture consists of banks, rows, and columns

with other requestors in shared resources, assuming conservative execution times for all tasks. This contrasts to the poor coverage achieved by simulation. The time required to develop formal performance models is not negligible, but these models can be reused together with the software or hardware block they model. Verification of real-time requirements using simulation-based techniques, on the other hand, cannot typically be reused. The problem with formal verification is that it requires performance models of the software, the hardware, and the mapping [17, 71]. Suitable application models, such as data-flow graphs, exist, but are not yet widely adopted by industry. Most industrial hardware has furthermore not been designed with formal analysis in mind. There have been recent advances in the research community, where some IP components have been proposed together with corresponding performance models [48]. However, a satisfactory solution has not yet been developed for SDRAM memories. This prevents formal analysis techniques from being applied to many platforms, since SDRAMs are essential to satisfy large storage requirements at a reasonable cost. The reason SDRAM memories are difficult to combine with formal analysis is due to a combination of complex temporal behavior that is inherent to their architecture and contradictory requestor requirements. The next section elaborates on these problems.

1.1.6 SDRAM and Real-Time Requirements

SDRAM memories are challenging to use in systems with real-time requirements because of their internal architecture. An SDRAM memory comprises a number of banks, each containing a memory array with a matrix-like structure, consisting of rows and columns [60]. A simple illustration of this architecture is shown in Fig. 1.9. Each bank has a row buffer that can hold one open row at a time, and read and write operations are only allowed to the open row. Before opening a new row

in a bank, the contents of the currently open row are copied back into the memory array. The elements in the memory arrays are implemented with a single capacitor and a resistor, where a charged capacitor represents a one and an empty capacitor a zero. The capacitor loses its charge over time due to leakage and must be refreshed regularly to retain the stored data.

The SDRAM architecture causes the offered bandwidth and the time to serve a memory request to depend on three things. First, there is a dependency on the row targeted by the request and the rows that are currently open in the banks. The reason is that a request targeting an open row can be served immediately, while a request targeting a closed row must wait until the current open row has been closed and the required row has been opened. The overhead from opening and closing rows results in additional latency, as well as idle cycles on the data bus. The latter implies a reduction of the offered bandwidth. The second dependency is on the direction (read/write) of the current and previous request. The reason for this dependency is that the data bus is bi-directional and requires a number of clock cycles to change direction from read to write or write to read, again adding latency and wasting bandwidth. The last dependency is on the temporal alignment with respect to refresh operations, since a refresh operation requires tens of clock cycles during which no data can be transferred on the data bus. Together, these three dependencies create large variations in the time required serve a read or a write request. The first two dependencies are especially problematic, since they involve previous requests that may have been issued by other requestors sharing the resource. This creates *resource interference* between requestors, where the time required by the resource to serve a scheduled request from one requestor depends on other requestors. These effects make it very difficult to bound the bandwidth offered by the memory and the latency of memory requests at design time, which is required to support firm and hard real-time requirements.

We proceed by elaborating on the requirements of SDRAM requestors, and explain what makes them contradictory and difficult to satisfy. The bandwidth requirements of requestors may be diverse and range from a few KB/s for audio decoding to a few GB/s for high-definition video processing. SDRAM requestors are furthermore categorized as either *latency sensitive* or *latency tolerant* [126]. Latency-sensitive requestors require low-latency memory accesses to reduce the number of stall cycles on the processing elements. This is typical for blocking processing elements that do not support multiple outstanding transactions and that store data in a remote memory, such as an SDRAM. When no more transactions can be issued, the processing element blocks until a response has been returned, potentially resulting in long stalls [63, 126]. This problem is often mitigated by using a cache to store commonly used data locally, significantly reducing the average memory access latency for applications with good locality. However, many processing elements still spend a significant number of clock cycles waiting for data, due to long latencies in the interconnect and memory controller. This problem got increasingly severe throughout the single-processor era, since processor speed increased faster than memory speed. In fact, both processor and memory speeds increased exponentially, but with different exponents, causing the difference

between the two to also increase exponentially [138]. This observation has resulted in the theory that the performance of many applications will eventually be dominated by the memory latency, a situation that is known as hitting the *memory wall* [138]. The effects of the memory wall can be observed in transaction-based workloads and high-performance scientific computing [76], where processors can stall up to 95% of the time. The recent step to multi-processor platforms has reduced the clock frequencies of processors [2], which should mitigate the effects of the memory wall. However, the cumulative memory bandwidth requirement of all processing elements is still increasing, adding a new dimension to the problem.

Some applications, such as media processing, can often be implemented in a pipelined fashion. The requestors of these applications are more latency-tolerant, but require guaranteed bandwidth to sustain their throughput requirements. In this case, higher bandwidth enables higher resolutions and support for more functionality, such as additional tasks that improve the quality of the output. However, external memory bandwidth is a *scarce resource* in many platforms. The reason is that an SDRAM controller is an expensive component both in terms of area and power consumption. Adding more memory controllers, or making the SDRAM interface wider, requires more pins. More pins further increases both the area and power consumption, and may also require a more expensive packaging. Using multiple memory controllers is hence often not an option, making it important to use the existing SDRAM bandwidth as efficiently as possible.

The requirements of latency-sensitive and latency-tolerant requestors are challenging to satisfy, since low latency and high offered bandwidth are inherently contradictory properties for SDRAMs. The memory is efficiently utilized by limiting the number of switches between reads and writes and using large requests to make better use of an open row. Providing low latency to sensitive requestors, on the other hand, is achieved by letting them switch directions immediately and preempt less important requestors, potentially closing the open rows they are using. Both of these actions reduce latency for sensitive requestors at the expense of a reduction of the bandwidth offered by the SDRAM.

1.2 Problem Statement

The high-level problem addressed in this book is to design a memory controller that satisfies the real-time requirements of applications in embedded systems, thereby reducing the mapping and verification effort. More specifically, the proposed memory controller should *address the diversity of contemporary platforms* by supporting different types of memories (SRAM and SDRAM in particular) and arbiters. The memory controller must *use the memory bandwidth efficiently*, since it is a scarce resource that must be carefully utilized. To reduce the mapping effort, the memory controller should be supported by tooling that *automatically determines instantiation parameters and configuration settings* for all components in the architecture, such that all application requirements are satisfied. The memory controller should *improve verification coverage* by enabling formal verification of

real-time requirements. It should furthermore *reduce the verification complexity* by enabling independent verification of applications using either formal methods or simulation-based techniques.

1.3 Requirements

Based on the problem statement in the previous section, we impose four requirements on the memory controller design: predictability, abstraction, composability and automation. We proceed by explaining the concepts behind these requirements, and motivate their relevance with respect to the problem statement. An overview of how our solution implements these requirements is provided in Chap. 2, and is hence not discussed here.

1.3.1 Predictability

The first requirement on the memory controller is predictability. In this book, we consider a component predictable if and only if *a useful bound is known on temporal behavior that covers all possible initial states and state transitions*. A component in this definition may refer either to a piece of hardware or software, which affects the particular temporal behavior that should be bounded. For example, determining the time required by a memory controller to serve a memory request requires both the allocated bandwidth and the latency of the controller to be bounded. On the other hand, computing the throughput of a video application may require bounds on the worst-case execution times of all its tasks. Predictability has a hierarchical aspect to it, since the temporal behavior of a component is determined by the timings of the sub-components it comprises. This implies that a predictable system must be built from predictable components. We proceed by discussing the relevance and implications of our definition of predictability more closely, starting with a brief discussion about predictability versus determinism.

A component is *deterministic* if it can be implemented by a state machine that provides a unique output, given a particular input and state. The behavior of a deterministic component is hence perfectly well-defined given a particular input sequence and initial state, making it predictable in some sense of the word. A non-deterministic component, on the other hand, can transition to multiple states with possibly different outputs, given a particular state and input. An example of non-deterministic component is an asynchronous clock domain crossing, the latency of which varies depending on the alignment of the different clock signals and the time to settle the signals to a stable state [127]. A non-deterministic component may intuitively feel unpredictable. However, our definition of predictability requires a bound on temporal behavior, as opposed to knowing the exact temporal behavior. This implies that our notion of predictability is not exclusive to deterministic components.

To use a bound in a general analysis, we require it to cover all possible state transitions and initial states. This is a key problem when analyzing the behavior of a component. For a deterministic component, the possible transitions depend on the input sequence. Non-deterministic components additionally require all possible transitions from a visited state to be considered, further complicating analysis. Determining the state transitions that trigger the worst-case behavior may be extremely difficult, especially if the temporal behavior of the component is data dependent and the set of possible inputs is large. Consider, for instance, the problem of determining the worst-case decoding time of a video frame in an H.264 decoder. Due to the difficulties in deriving these general bounds, we do not consider components predictable until this analysis has been done. Knowing that a bound exists is hence not a sufficient condition for a component to be considered predictable in this book.

Our definition of predictability also states that the derived bounds must be useful. The reason is to prevent behaviors that are bounded with useless bounds from being considered predictable. For example, we do not consider a memory controller to be predictable if the latency of a memory access is bounded by a year, since it cannot satisfy any realistic requirements. The exact meaning of usefulness and the required tightness of the bound is of course highly dependent on the behavior that is being bounded and the context in which is going to be used. This part of the definition hence has to be considered on a case-by-case basis. Considering usefulness in the definition of predictability implies that unlike the definition in [41], we mix the ability to predict a property with the quality of the prediction. Although keeping these concerns separate provides a cleaner definition of the concept, we find our definition both sufficient and easy to work with.

We proceed by exercising our definition by an example, where we consider bounding the offered bandwidth from a typical Double-Data-Rate (DDR) SDRAM controller. If we cannot exploit any knowledge of the initial SDRAM state or the incoming request stream, which is typically the case, we have to assume that every memory access targets a closed row. The currently open row hence has to be closed and the requested row opened before the access can proceed. This results in added latency and many unused cycles on the data bus of the memory, as explained in Sect. 1.1.6. It is not possible under this assumption to guarantee that the offered bandwidth will be greater than some 10–40% of the maximum bandwidth, depending on the speed of the memory [7]. Although this is a known bound on relevant behavior that covers all state transitions and initial states, it is not considered useful for many SoC designs, since SDRAM bandwidth is a scarce resource that must be efficiently utilized.

The memory controller proposed in this book is required to provide useful bounds on offered bandwidth and latency to be able to satisfy the communication requirements of the requestors. This requirement addresses the problem statement in this book by enabling formal verification of application requirements in predictable systems. Note that this requires performance models of the applications, as well as all other hardware components they are using.

Formal verification of a predictable system has the benefit of covering all possible input sequences and initial states, as opposed to the limited subset that can be verified by simulation. This makes this verification approach essential in systems with hard and firm real-time requirements. Formal verification is furthermore less sensitive to changes in use-case specifications than simulation-based techniques, since an application only requires re-verification if the temporal bounds of any of its tasks increases. This provides some additional flexibility in development of IP components and reduces the verification effort. However, this benefit assumes that the application model is performance monotonic, which means that a local reduction in latency cannot result in an overall latency increase. Another benefit of formal performance analysis is that there is a clear relation between platform parameters and the resulting temporal behavior. This may allow buffer sizes and configuration settings that satisfy the application requirements to be automatically synthesized, removing the need for mapping approaches based on trial-and-error.

1.3.2 Abstraction

Contemporary SoCs platforms consist of an increasing number of shared resources of different types, such as peripherals, interconnect, and several different types of memories. Access to these resources may furthermore be controlled by many different types of arbiters, which may all affect the temporal properties of an application. The complexity resulting from the diversity of shared resources can be reduced by abstraction. Abstraction is a mapping from one description of an object to another, where the second description is simpler in some sense [77]. An example is the digital abstraction that reduces continuous-time analogue signals with continuous amplitude into a discrete sequence of ones and zeroes. A *shared resource abstraction* can be used to capture the (temporal) behavior of the diversity of shared memories and their arbiters, and hide the details of their implementations [106]. A good abstraction should be simple to reduce complexity, yet capture relevant behavior as closely as possible. An abstraction with many parameters can be difficult to use, while hiding too much detail may result in suboptimal models and poor utilization of the resulting system. Abstraction hence presents a delicate trade-off between simplicity and accuracy.

The memory controller proposed in this book requires a shared resource abstraction that captures temporal behavior in a way that makes the details of the types of memory and arbiter transparent to the user. The chosen abstraction must be simple and general enough to apply to a wide range of memories and arbiters, while providing useful accuracy. Abstraction should also be used in the hardware architecture to allow memories and arbiters to be exchanged with minimum effort. The value of the abstraction requirement is that it allows the user to deal with different memories and arbiters in a homogeneous way, thus reducing complexity [106]. It furthermore enables reuse of models and tooling for different combinations of memories and arbiters, which increases design productivity and greatly simplifies automation.

1.3.3 Composability

A system is considered *composable* if applications cannot affect each other's behavior in the value and time domains [9]. This implies that applications are completely independent and cannot change each other's data, nor affect each other's temporal behavior by even a single clock cycle. Composability is an issue with shared resources, as they often enable requestors to affect each other's temporal behavior by either scheduling interference or resource interference, discussed earlier. Scheduling interference occurs in the arbiter, where the presence or absence of a request from one requestor may cause another requestor to be scheduled earlier or later. Resource interference happens in the resource itself when a requestor alters the resource state in a way that affects the time it takes to serve a request from another requestor. An example of resource interference is switching the direction of the data bus in an SDRAM memory.

The proposed memory controller is required to provide composable service to applications. For simplicity, we will refer to a memory controller that satisfies this requirement as a composable memory controller. Composability addresses the problem statement in this book by reducing the verification effort with simulation-based techniques in the following four ways [48]: (1) Applications can be verified by simulation in isolation, resulting in a linear and non-circular verification process. (2) Simulating only a single application and its required resources reduces simulation time compared to complete system simulations. This allows more use-cases to be verified, increasing the verification coverage. (3) The verification process can be incremental and start as soon as the first application is available. (4) Functional verification is simplified, since bugs caused by, for instance, race conditions in the integrated application, are independent of other applications. Another benefit of composability is that independent applications create well-defined liabilities, which is important if applications are developed by different parties [97]. IP protection is furthermore improved, since the verification process no longer requires the IP components of independent software vendors to be shared. Note that composability eliminates all interference from other applications, but that the platform may be non-deterministic, or even unpredictable [47]. For example, the platform may contain asynchronous clock domain crossings with non-deterministic latency [127], which may result in that a particular simulation trace from the execution of an application is hard to reproduce.

Composability is not a concept without drawbacks, since it involves eliminating both positive and negative interference between applications. This implies that *slack*, which is unreserved resource capacity or resource capacity reserved by one application that is currently not used, cannot be used by another application. Although this does not impact the worst-case latency of an application, it affects the average case, for instance by making non-real-time applications appear less responsive. However, composability does not imply that all slack is wasted. It is possible to safely distribute slack between requestors belonging to the same application [9].

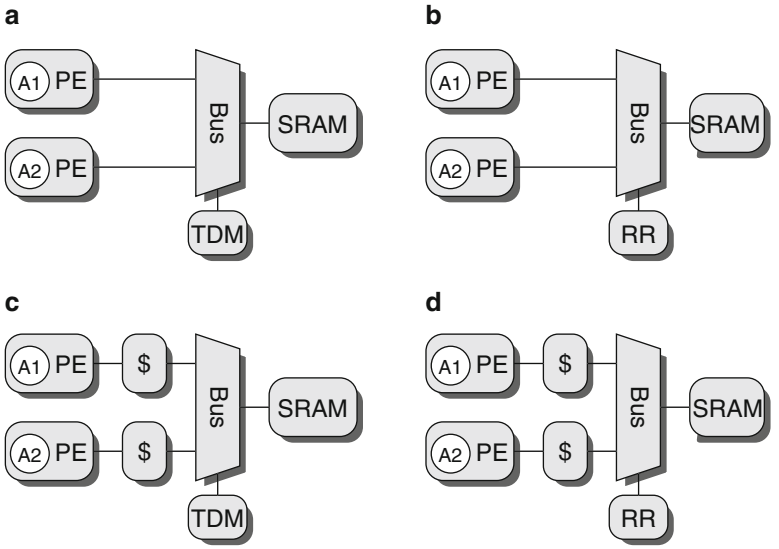

Fig. 1.10 Four systems demonstrating all combinations of the predictability and composability properties. (**a**) Predictable and composable system. (**b**) Predictable system. (**c**) Composable system. (**d**) Neither predictable nor composable system

It is important to realize that predictability and composability are two different properties and that one does not imply the other. Predictability means that a useful bound is known on temporal behavior and is hence a property of a *single application* mapped on a set of resources. Composability, on the other hand, implies complete functional and temporal isolation between applications and is a property of *multiple applications* sharing resources, where each application may be predictable or not. We illustrate the difference by discussing four example systems, shown in Fig. 1.10, that cover all combinations of composability and predictability. The first system, depicted in Fig. 1.10a, consists of two PEs, each executing a single application (A1 and A2, respectively). We assume that both applications are predictable and hence that worst-case execution times are known for all tasks when running on predictable hardware. Data is stored in a shared remote Zero-bus-turnaround (ZBT) SRAM that is reached via a bus. This type of SRAM has a latency of one clock cycle per read or written word that is independent of other requestors. The SRAM is shared using TDM arbitration, which is a predictable and composable arbitration scheme, since the latency of a requestor is both bounded and independent of other requestors. This makes this system as a whole both predictable and composable. For our second system in Fig. 1.10b, we replace the TDM arbiter with a Round-Robin arbiter (RR). This system is not composable, since response times of requests vary depending on the presence or absence of requests from other requestors. However, it is still predictable, since this interference is easily bounded. We create our last two systems by adding private L1 caches ($) with random replacement

policies to the processors in both previous systems. A private cache is composable, since it is not shared between applications. However, the random replacement policy makes the systems unpredictable, since a useful bound cannot be derived on the time to serve a sequence of requests. The third system, in Fig. 1.10c, is hence composable, but not predictable. The last system, shown in Fig. 1.10d, is neither predictable, nor composable.

1.3.4 Automation

Automation refers to having parts of the design process done by tools. Automation has grown to become an essential part of embedded system design, since it reduces the design time, directly impacting time to market [135]. As explained in Sect. 1.1.4, the mapping process contains a configuration step that is typically performed manually. An SDRAM controller has many instantiation parameters and configuration settings, such as buffer sizes and the burst size of the SDRAM. Many arbiters furthermore need to be configured. The particular configuration settings vary depending on the arbiter type, but may involve bandwidth allocations and priority assignments.

The proposed memory service is required to have an automated approach to finding IP parameters and configuration settings. This involves automatic computation of configuration settings for the memory controller and its associated arbiter at design time. This is required to reduce design time by removing a manual step from the mapping process that relies on trial-and-error and extensive system-level simulation.

1.4 System Context

Predictable and composable systems are built from predictable and composable components, such as processing elements, interconnect, memories, and operating system. The predictable and composable memory controller presented in this work is hence only a piece of the puzzle, albeit a very important one. The proposed memory controller is intended as a useful addition to any predictable and/or composable system. Several such systems have been proposed, such as Time-Triggered systems [69], Loosely Time-Triggered systems [19], METERG systems [72], Virtual Private Machines [94], PRET Machines [33], the MERASA platform [124], and our own CoMPSoC [48] platform.

This book uses the CoMPSoC platform as the primary system context and is the basis for all experiments. The CoMPSoC platform is a predictable and composable multi-processor platform fitting with the general platform template previously shown in Fig. 1.5. The platform comprises predictable and composable processor tiles [82], a simple SRAM controller with TDM arbitration [48], the

Æthereal Network-on-Chip (NoC) [38, 47], and the CompOSe real-time operating system [45,82]. An overview of the general techniques used to achieve predictability and composability in these components is presented in [9].

This work presents a memory controller architecture for the platform that supports both SRAM and DDR2/DDR3 SDRAM and a variety of arbiters in a unified manner. An architecture instance of the CoMPSoC platform using the proposed memory controller to interface to both SRAM and SDRAM is shown in Fig. 1.11. Next, we provide a brief overview of this architecture.

At a high level, the CoMPSoC architecture can be divided into processor tiles, interconnect, memory tiles, and peripheral tiles. A processor tile is equipped with a MicroBlaze processor for computation. The MicroBlaze is a soft-core 32-bit RISC processor developed by Xilinx [139] for use with their FPGAs. The processor runs the CompOSe real-time operating system [45], which enables predictable and composable execution of applications distributed across multiple processors and memories. Power management in a tile is enabled by a voltage and frequency control module (VFCM). The VFCM is used by the operating system to support predictable and composable dynamic voltage and frequency scaling (DVFS) [39] by doing energy and power budgeting per application [91].

A processor tile furthermore contains an unshared instruction memory (IMEM) and a data memory (DMEM) that store private code and data, respectively. For each application mapped on the processor tile, there is at least one remote DMA engine (RDMA), incoming communication memory (CMEMi), and outgoing communication memory (CMEMo). These are used for all off-tile communication, including incoming memories (CMEMi) in other processor tiles, distributed shared memories in memory tiles, and peripheral tiles. The purpose of this infrastructure is to ensure that communication is composable and decoupled from computation [39]. An application communicates by first writing data in its outgoing communication memory (CMEMo) and then instructing the RDMA to transfer the data to an off-tile memory. Responses, if any, arrive in the application's incoming communication memory (CMEMi), where they can be accessed by the processor. Inter-task communication for distributed applications is implemented on top of this using FIFO buffers, managed according to the C-HEAP protocol [95]. These FIFOs can be mapped in any off-tile memory, such as a memory tile or the incoming communication memory (CMEMi) of the producing or consuming tile.

A request entering the interconnect from a processor tile first passes through a shell that serializes the parallel bus protocol used by the processor tile, in our case DTL [101], to a sequence of words. These words are then passed through a clock domain crossing (CDC) to transition from the clock domain of the initiator tile to that of the network. The data is then sent through the network, comprising Network Interfaces (NI) and routers (R), through a logical connection. The NI packetizes the data and determines the route through the network. The routers merely forward the data to its destination NI where it is depacketized, before transitioning to the clock domain of the target tile in another clock domain crossing. The shell then deserializes the request and presents it to the port of the target processor tile,

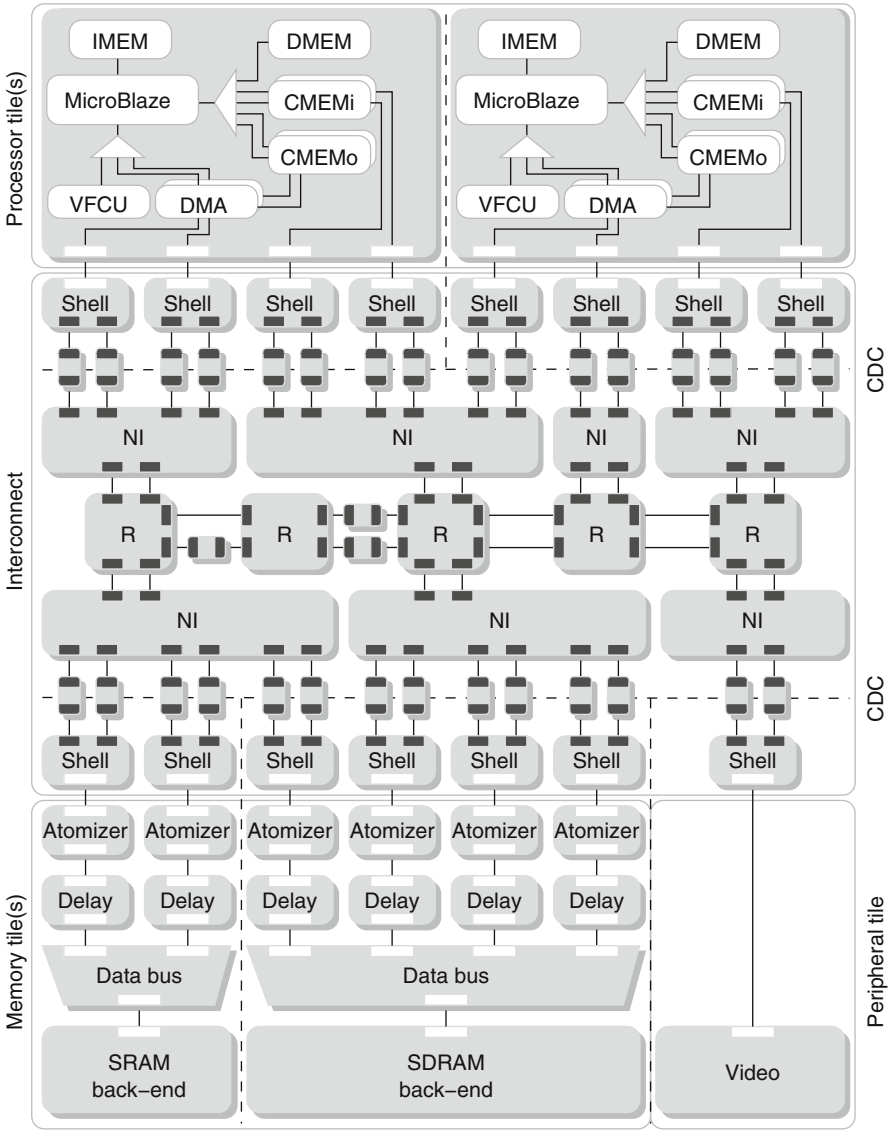

Fig. 1.11 Simplified architecture of a CoMPSoC instance with two processor tiles and both centralized SRAM and SDRAM memories

memory tile, or peripheral tile. A response, if generated by the target, follows the same logical connection back through the network until it reaches the initiator.

The architecture of the memory tiles is not explained in this section, as it comprises the memory controller discussed throughout the rest of the book. We also do not discuss peripheral tiles, since they are memory-mapped resources that

are treated just like memories. Note that the peripheral tile in Fig. 1.11 is not shared by multiple requestors and that shared instances have the same Atomizers, Delay Blocks, and Data bus as the memory tile.

1.5 Contributions

This section lists the main contributions of this book. The contributions are discussed in terms of the illustration of the proposed predictable and composable memory controller shown in Fig. 1.12. All hardware is implemented both as SystemC simulation models and in synthesizable VHDL.

- A *predictable SDRAM back-end* [7] is presented that provide hard/firm real-time guarantees on bandwidth and latency with any DDR2/DDR3 SDRAM memory, while increasing the level of flexibility over previous approaches. (Chap. 4)
- We *evaluate three predictable arbiters*: TDM, Frame-Based Static-Priority (FBSP), and our own Credit-Controlled Static-Priority (CCSP) arbiter. The arbiters are compared both analytically and experimentally in terms of maximum latencies and wasted resource capacity to highlight their strengths and weaknesses when scheduling accesses to shared SoC resources. (Chap. 5)
- A *general predictable resource front-end* is proposed that provides access to shared predictable memories, such as our SDRAM back-end or an SRAM controller. The front-end contains an arbiter in the class of Latency-Rate ($\mathcal{LR}$) servers, which is a class with many well-known predictable arbiters, including the ones presented in Chap. 5. The front-end guarantees a requestor a minimum bandwidth and a maximum latency with *any combination* of supported arbiters and predictable memories. They hence act like a $\mathcal{LR}$ server, which is the *shared resource abstraction* used in this work. This abstraction enables formal

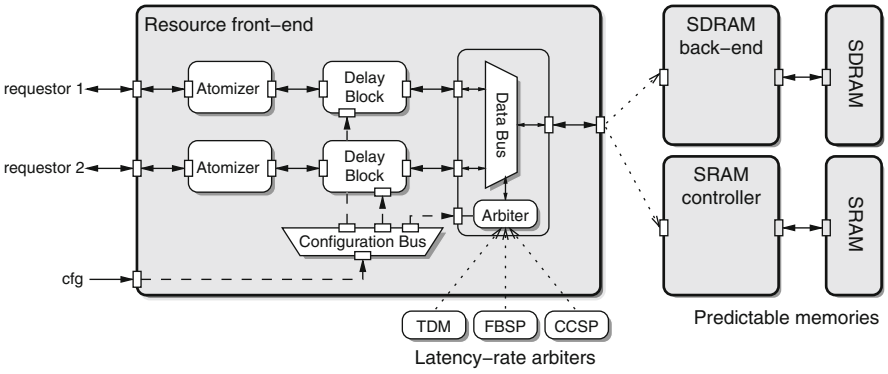

Fig. 1.12 The proposed predictable and composable memory controller

verification of real-time requirements in a transparent manner for multiple types of memories and arbiters using several commonly used performance analysis frameworks. (Chap. 6)

- We introduce a novel approach to *composable resource sharing* that makes predictable shared resources composable [8]. The idea is to add a Delay Block to the front-end that delays all signals sent to a requestor to emulate worst-case interference from others. This approach enables composability with a wider range of applications and shared resources than previous work. It furthermore allows requestors that do not require composable service to *use slack bandwidth* to improve performance. (Chap. 6)
- We propose an *automated configuration flow* that computes instantiation parameters and configuration settings to satisfy requestor requirements. The flow uses abstraction to make the memory and arbiter configuration independent of each other. This enables all supported arbiters to be configured for all supported memories in a streamlined fashion without a special case for every combination. (Chap. 7)

1.6 Outline

This book is organized as follows. Chapter 2 provides an overview of our proposed memory controller in terms of the four requirements: predictability, abstraction, composability, and automation. Chapter 3 contains an introduction to SDRAM memories and explains why they are difficult to use in real-time systems. It also discusses the general building blocks of an SDRAM controller and highlights interesting design options. An SDRAM back-end is presented in Chap. 4 that makes a DDR2/DDR3 SDRAM behave in a predictable manner, and bounds are derived on bandwidth and latency. Chapter 5 addresses how to share the back-end, or other resources, among multiple requestors by analytically and experimentally comparing three predictable arbiters in terms of maximum latency and resource allocation efficiency. A resource front-end is presented in Chap. 6. The front-end provides predictable service with any combination of arbiter in the class of $\mathcal{LR}$ servers and predictable resource, such as our SDRAM back-end or an SRAM controller. We furthermore show how to make the shared predictable resource composable by delaying all signals sent from the front-end to a requestor to emulate maximum interference from others. Chapter 7 presents our configuration flow and demonstrates with a running example how instantiation parameters and configuration settings are derived for both the front-end and the back-end. The proposed solution is positioned with respect to related work on resource arbitration, memory controllers, and composability in Chap. 8. Lastly, conclusions and future work are presented in Chap. 9.

1.7 Summary

Embedded system design gets increasingly complex. Each new product generation integrates more applications and contains more hardware and software. The product life time is furthermore reducing, requiring new generations to be designed, verified, and released faster than ever before.

Applications in embedded systems often have *real-time requirements*, meaning that they must perform a particular computation before a deadline. A real-time requirement is classified as either hard, firm, or soft, depending on its criticality. Hard real-time requirements must always be satisfied to guarantee safety or functional correctness. Similarly, firm real-time requirements must be satisfied to prevent significant quality degradation, while missing a soft requirement may just be perceived as annoying to the user. An application is partitioned into tasks, which can be mapped to different processing elements in the platform. Contemporary platforms often contain multiple heterogeneous processing elements, to provide good balance between performance, cost, power consumption and flexibility. They also have a distributed memory hierarchy with different types of shared memories, such as Static RAM (SRAM) and Synchronous Dynamic RAM (SDRAM), to achieve large storage capacity with low latency at a reasonable cost per bit. However, due to pin constraints, SDRAM bandwidth is a scarce resource that must be efficiently utilized.

Mapping applications on the platform such that all real-time requirements of the applications are satisfied is very challenging. The number of possible bindings of tasks to processing elements, and data structures to memories is very large, and appropriate instantiation parameters and arbiter settings must be derived. Verifying that a particular mapping satisfies all application requirements is very time consuming, since it is often done by simulation with poor use-case coverage. Formal verification offers significantly better coverage, but is typically not an alternative, since most industrial hardware and software is not designed with formal analysis in mind. SDRAM memories are examples of commonly used components that make verification difficult. The bandwidths and latencies provided by these memories depend highly on the requests sent by the applications. The timing behaviors of concurrently executing applications hence become inter-dependent, making it impossible to verify them in isolation. Instead, concurrently executing applications must be verified together, resulting in that the *verification complexity grows exponentially* with the number of applications.

The problem in this book is to design a memory controller that satisfies hard, firm, and soft real-time requirements, thereby reducing the mapping and verification effort of embedded systems. We impose four requirements on the solution to achieve this goal. The memory controller should be *predictable* in the sense that there must be useful bounds on the latency of a memory request and on the provided bandwidth. This enables the controller to be used with formal verification techniques, improving use-case coverage. The solution should make use of *abstraction* to support different types of memories, such as SRAM and SDRAM, and different arbiters transparently.

This reduces design time by enabling reuse of tools and models. We require the memory controller to be *composable*, which means that two applications sharing the memory cannot modify each other's data or affect each other's temporal behavior by even a single clock cycle. This property allows applications to be verified in isolation, reducing the verification complexity. Lastly, we require *automation* of the memory controller configuration to reduce the mapping effort. The controller should hence be supported by tooling that automatically derives configuration settings and instantiation parameters, such that application requirements are satisfied.

We briefly introduced the CoMPSoC platform, which is a predictable and composable multi-processor platform that provides the primary system context for the proposed memory controller. The platform features predictable and composable processor tiles with dynamic voltage and frequency scaling and real-time operating system, network-on-chip, and a simple SRAM controller. This work extends the platform with a predictable and composable memory controller that supports both SRAM and DDR2/DDR3 SDRAM and a variety of arbiters in a unified manner.

Chapter 2
Proposed Solution

The previous chapter identified problems related to mapping applications with real-time requirements to a heterogeneous multi-processor platform with SDRAM memory and verifying that all requirements are satisfied. We then committed to designing a memory controller with requirements on predictability, abstraction, composability, and automation to address this issue. This chapter presents an overview of the proposed solution, and explains how it delivers on each of the four requirements. We begin in Sect. 2.1 by discussing predictability. We then move on to abstraction in Sect. 2.2, followed by composability and automation in Sects. 2.3 and 2.4, respectively. Lastly, the chapter is concluded with a summary in Sect. 2.5.

2.1 Predictability

Section 1.3.1 stated that the memory controller must provide useful bounds on the offered bandwidth and latency of memory transactions. This section explains how the proposed memory controller delivers on this requirement. First, we present an overview of our approach to predictability, which is based on combining predictable memories with predictable arbitration. Then, we explain how to make an SDRAM memory behave in a predictable manner, before concluding with a discussion on predictable arbitration.

2.1.1 Overview of Approach

Our approach to predictable memory controllers is based on combining memories and arbiters with predictable behaviors. More specifically, from the memory, we require bounds on the offered bandwidth and the time to serve a scheduled request, since these characterize the worst-case behavior of an unshared memory. We refer

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_2, © Springer Science+Business Media, LLC 2011

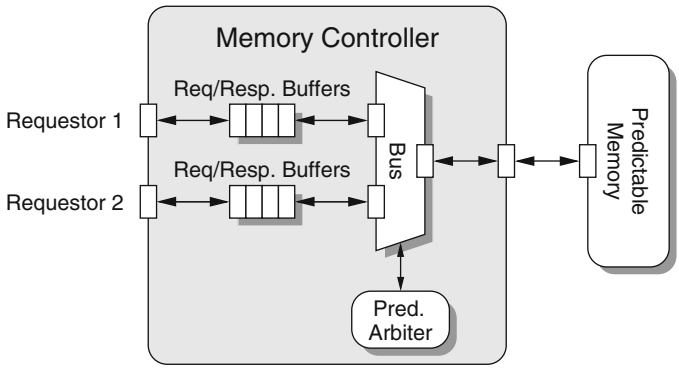

Fig. 2.1 Overview of predictable memory controller

to a memory satisfying this requirement as a *predictable memory*. We also require a *predictable arbiter*, where the number of interfering requests that can be scheduled before a particular request is bounded. Combining a predictable memory and a predictable arbiter allows the maximum time to schedule a particular request to be computed by multiplying the number of interfering requests with the maximum time to serve a scheduled request. This takes the effects of sharing the memory into account. Our approach is hence based on combining *independent analyses* of the memory and the arbitration. The strength of this approach is that it lets us design a general memory controller, providing predictable service for *any combination of predictable memory and predictable arbiter*. This helps us satisfy our abstraction requirement, as further discussed in Sect. 2.2. An illustration of a basic memory controller is provided in Fig. 2.1. We use this architecture as a starting point and extend it with additional elements throughout this chapter until we reach the final design, previously shown in Fig. 1.12.

2.1.2 Predictable SDRAM Back-End

As previously mentioned, our approach to predictable memory controllers requires a useful bound on: (1) the bandwidth offered by the memory, and (2) the time to serve a request. Satisfying these requirements is straight-forward for stateless Zero-bus-turnaround (ZBT) SRAM memories, where the available bandwidth simply corresponds to the product between the width of the memory interface and the clock frequency, and a word is served with a fixed latency of one clock cycle. However, as mentioned in Sect. 1.1.6, this is more difficult for SDRAMs, where both the offered bandwidth and the time to serve a request depend on the interleaving of requests from all requestors sharing the memory, which is not known at design time.

The behavior of an SDRAM memory is determined by the sequence of SDRAM commands that are communicated from the memory controller to the memory device. These commands tell the memory to activate (open) a particular row in

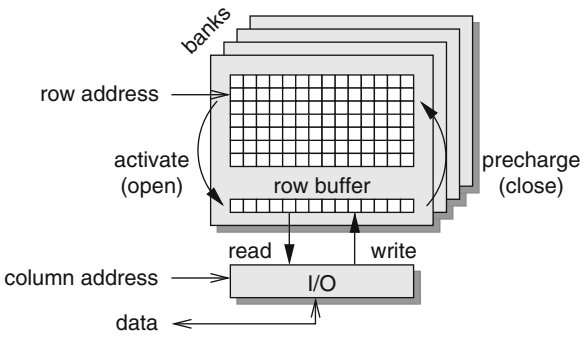

Fig. 2.2 The behaviors of some important SDRAM commands

the memory array, to read from or write to an open row, or to precharge (close) an open row and store its contents back into the memory array. There is also a refresh command that charges the capacitors of the memory elements to ensure that the contents of the memory array are retained. The behaviors of some of these commands are illustrated in Fig. 2.2. Scheduling SDRAM commands is not a trivial task, since there are a considerable number of timing constraints that must be satisfied before a command can be issued. These timing constraints are minimum delays between issuing particular SDRAM commands, such as two activates, or an activate and a read or a write.

Existing SDRAM controllers can be divided into two categories, depending on how they schedule the SDRAM commands. Statically scheduled controllers execute precomputed command schedules that are guaranteed at design time to satisfy all timing constraints of the memory. Executing precomputed schedules makes these controllers predictable and easy to analyze. However, they are also unable to adapt to the dynamic behavior of applications in contemporary System-on-Chips (SoCs), such as bandwidth requirements or read/write ratios that vary over time. The second category of controllers uses dynamic scheduling of commands, which requires the timing constraints to be enforced at run time. These controllers have sophisticated command schedulers that attempt to maximize the average offered bandwidth and to reduce the average latency at the expense of making the resource extremely difficult to analyze. As a result, the offered bandwidth can only be estimated by simulation, making bandwidth allocation a difficult task that must be re-evaluated every time a requestor is added, removed, or is modified.

We propose a hybrid approach to SDRAM command scheduling that combines elements of statically and dynamically scheduled SDRAM controllers in an attempt to get the best of both worlds. Our approach is based on *predictable memory patterns*, which are precomputed sequences (sub-schedules) of SDRAM commands that are known to satisfy the timing constraints of the memory. These patterns are dynamically combined at run-time, depending on the incoming request streams. The memory patterns exist in five flavors: (1) read pattern, (2) write pattern, (3) read/write switching pattern, (4) write/read switching pattern, and (5) refresh pattern. The patterns are created such that multiple read or write patterns can be

scheduled in sequence. However, a read pattern cannot be scheduled immediately after a write pattern. In this case, the read pattern must be preceded by a write/read switching pattern. This works analogously in the other direction. The refresh pattern can be scheduled immediately after either a read pattern or a write pattern. Both read and write patterns can be scheduled immediately after a refresh without any preceding switching patterns.

The read and write patterns consist of a fixed number of SDRAM bursts, all targeting the same row in a bank. The bursts are issued to the different banks in sequence, since the data bus is shared between all banks to reduce the number of pins on the SDRAM interface. The fixed number of bursts is hence first sent to the first bank, then to the second, and so forth in an interleaving fashion until all banks have been accessed. This way of accessing the SDRAM results in that banks have a short period with frequent accesses, followed by a longer period without any accesses. The patterns exploit bank-level parallelism by issuing activate and precharge commands to the banks during the long intervals in which they do not transfer any data. The read and write patterns are hence very efficient in terms of bandwidth, since it is possible to hide a significant part of the latency incurred by activating and precharging rows. This limits the overhead cycles incurred by always precharging a bank immediately after it has been accessed, which is known as a close-page policy. We implement this policy, as it effectively removes the dependency on rows opened by earlier requests by returning the memory to a neutral state after every access. Removing this dependency between requests is a *key element* in our approach, since it *reduces the variation in the offered bandwidth and latency*, enabling tighter bounds on bandwidth and latency to be derived.

Although interleaving memory patterns allow us to bound the offered bandwidth, they come with two drawbacks. The first drawback is that continuously activating and precharging the banks increases power consumption compared to if a single bank is used at a time [31, 78, 79]. The second drawback is that the memory is accessed with large granularity and hence requires large requests to be efficient. An efficient access requires at least one SDRAM burst to every bank. A typical burst length for SDRAM is 8 words and the number of banks is either four or eight. The minimum efficient request size for a 32-bit memory interface is hence between 128-256 B. Working with large requests in a non-preemptive manner also means that urgent requests can be blocked longer, resulting in longer latencies.

Figure 2.3 shows example read and write patterns for a 16-bit DDR2-400 memory device. The SDRAM commands in the figure are encoded according to activate (ACT), read (RD), write (WR), and no-operation (NOP). All read and write commands are issued with an automatic precharge option, causing the bank to be precharged automatically at the earliest possible convenience. This removes the need to explicitly issue precharge commands and furthermore ensures that an arbitrary row can be opened in the bank in the shortest possible time. The numbers in the figure correspond to the number of the bank associated with a command or data element. Note that two data elements are transferred every cycle due to the Double-Data-Rate (DDR) of the memory. The patterns in the figure are very efficient in terms of bandwidth, as they transfer data during every cycle if they are repeated

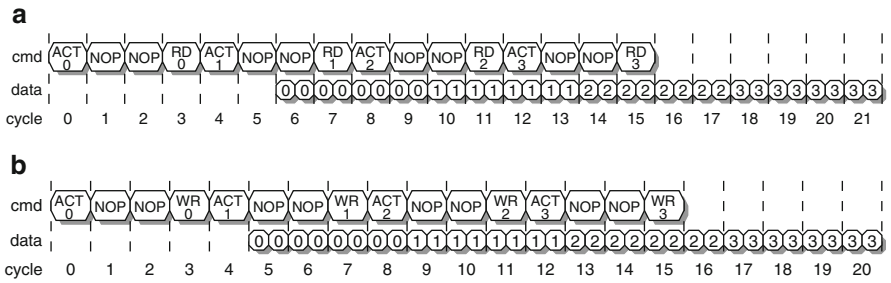

Fig. 2.3 Read pattern and write patterns with burst length 8 for a DDR2-400. (**a**) Read pattern. (**b**) Write pattern

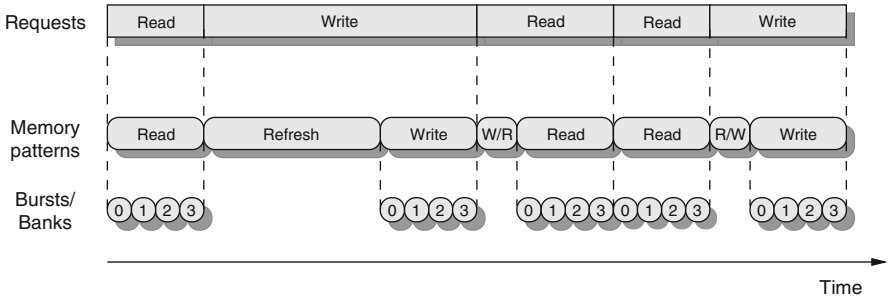

Fig. 2.4 Mapping from requests to patterns to SDRAM bursts

multiple times. The figure also shows that scheduling a write pattern immediately after a read pattern (first command of the write pattern in cycle 16 of the read pattern) causes a conflict on the data bus, which is one of the reasons switching patterns are needed.

Requests are dynamically mapped to patterns in a non-preemptive manner by the command generator in the memory controller. A scheduled read request maps to a read pattern, possibly preceded by a write/read switching pattern. Similarly, a write request is mapped to a write pattern and a potential preceding read/write switching pattern. Refresh patterns are scheduled automatically by the memory controller on a regular basis between requests. The mapping from requests to patterns and from patterns to SDRAM bursts is shown for an SDRAM with four banks in Fig. 2.4. The figure illustrates that the time to serve a request of four bursts varies depending on whether or not a switching pattern is required and if a refresh is scheduled before the request.

The benefit of memory patterns is that they raise SDRAM command scheduling to a higher level. Instead of dynamically issuing individual SDRAM commands,

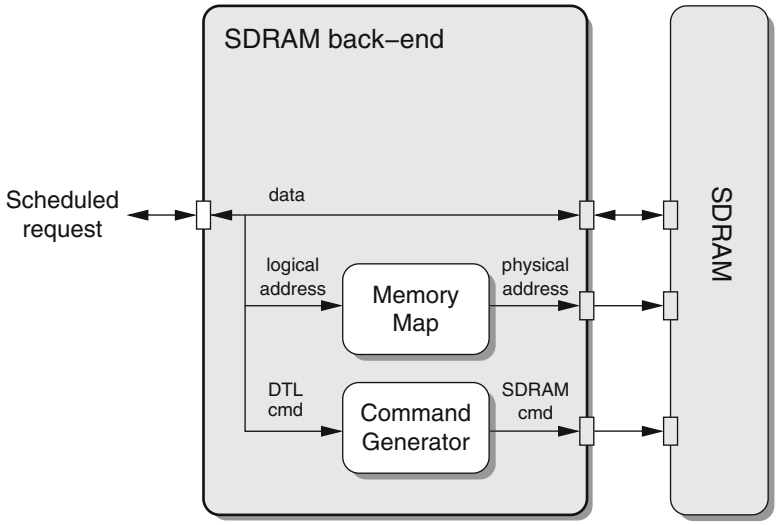

Fig. 2.5 Overview of the predictable SDRAM back-end

like a dynamically scheduled SDRAM controller, our proposed controller issues memory patterns that are sequences of commands. This implies a reduction of state and constraints that have to be considered, making our approach easier to analyze than completely dynamic solutions. Memory patterns allow a lower bound on the offered bandwidth and the time to serve a request to be determined, since we know the length of each pattern, how much data they transfer, and how they are dynamically combined in the worst case. The use of memory patterns hence gives our approach the predictability of statically scheduled memory controllers. In addition, our approach also has some properties of dynamically scheduled controllers, such as the ability to dynamically choose between read and write requests, and the use of run-time arbitration. The latter is the topic of the following section.

Our approach is implemented as an SDRAM back-end, as shown in Fig. 2.5. The back-end accepts a scheduled request through a Device Transaction Level (DTL) [101] port, and translates the logical address into a physical address (bank, row, and column) using an interleaved memory map. A command generator then issues the appropriate memory patterns and sends the SDRAM commands to the memory device. The implementation of the back-end is very light weight and has a small area foot print.

2.1.3 Predictable Arbitration

After the previous section, we assume that we have a predictable memory, such as an SRAM or our proposed SDRAM back-end based on predictable memory patterns,

where useful bounds on both the offered bandwidth and the time to serve a request are known. In this section, we consider the effects of sharing the predictable memory between multiple requestors. As mentioned in Sect. 2.1, we require a predictable arbiter, where the number of interfering requests before a particular request is scheduled is bounded. There exists are large number of both predictable and unpredictable arbiters in literature. To provide some concrete examples, we return to the three arbiters introduced in Sect. 1.1.3. Time-Division Multiplexing (TDM) schedules requestors according to a static schedule that is computed at design time. The latency of this arbiter is hence easily bounded by inspecting this schedule and knowing the request sizes of the requestors. TDM is hence a predictable arbiter. Round-Robin arbitration cycles between requestors that are trying to access the resource, skipping any requestors that are currently idle. This is another example of a predictable arbiter, since the latency is determined by the number of requestors and their request sizes. A static-priority arbiter, on the other hand, is an example of an arbiter that is unpredictable. The reason is that a high-priority requestor that continuously accesses the memory can prevent access from a low-priority requestor indefinitely, resulting in unbounded latency.

From the above example, we learn that a predictable arbiter must protect a requestor against other uncooperative or malfunctioning requestors that continuously access the memory. This is accomplished using either explicit or implicit *rate regulation*. The purpose of rate regulation is to protect requestors from the request rate of others by enforcing an upper bound on the provided service, for instance using an allocated budget. This is done explicitly by a TDM arbiter that assigns a number of slots in its schedule to each requestor. In contrast, a Round-Robin arbiter is not programmable and does not have explicit rate regulation. However, the service provided to a requestor is implicitly regulated, since the arbiter fairly cycles through all requestors that want to access the resource. A static-priority arbiter does not have either an explicit or implicit rate regulation mechanism, which is the reason it is unpredictable.

To be completely robust, we also need to be independent of the sizes of scheduled requests to prevent a malfunctioning requestor from continuously using the resource by issuing very large requests. We solve this problem by using *preemption*. This is implemented by adding an additional hardware block, called an *Atomizer* [48], to the memory controller architecture. The Atomizer splits requests into *atomic service units*, referred to as atoms, which are served by the memory in a known bounded time. Large requests are hence chopped up in smaller pieces, ensuring that other requestors can access the resources within a bounded time. The size of the atoms are fixed and determined at design time. The size of an atom is chosen to be the minimum request size that can be efficiently served by the resource. For an SRAM, the natural service unit is a single word, but it is much larger for an SDRAM with predictable memory patterns. In this case, the service unit might be between 16 and 256 words, depending on the memory device and the patterns. Using fixed-sized requests in the memory controller furthermore simplifies other blocks in the architecture, resulting in a faster implementation. Another benefit of adding the Atomizer as a separate hardware block on front of the arbiter is that it

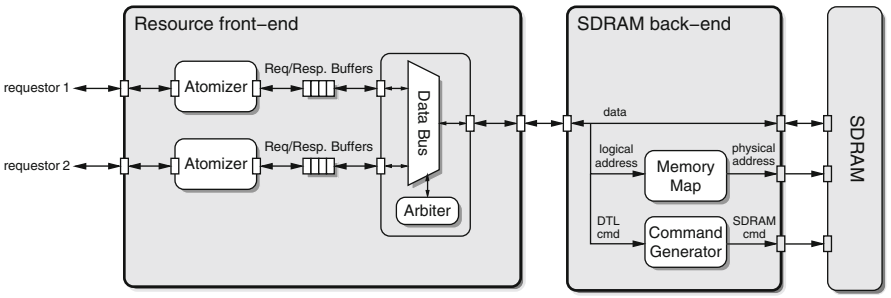

Fig. 2.6 A predictable SDRAM controller supporting two requestors

effectively makes all predictable arbiters preemptive on the granularity of atoms. This qualifies any existing predictable arbiter for use with our approach, which adds to the flexibility, while promoting reuse.

The benefit of using a predictable arbiter that combines rate regulation and preemption is that it makes it possible to bound the latency of a requestor without relying on the cooperation of others. Instead, bounds are based either on the allocated budgets (explicit rate regulation) or on analysis results of the scheduling mechanism (implicit rate regulation), both of which are known at design time.

A predictable SDRAM controller with two requestors is shown in Fig. 2.6. In addition to the SDRAM back-end and memory from Fig. 2.5, we see a predictable resource front-end with multiple DTL inputs and a single DTL output. The front-end contains an Atomizer per requestor that chops incoming requests into atoms. After the Atomizer are the Request and Response Buffers. Arriving atoms are stored in the Request Buffer until they are scheduled by the predictable arbiter. A scheduled atom is routed through the Data Bus to the output port of the front-end, arriving in the SDRAM back-end. The proposed front-end is general and fits with *any predictable resource* with a DTL interface. For instance, if we want to access an SRAM, we simply remove the SDRAM back-end and connect the output port of the front-end directly to an off-the-shelf SRAM controller with a DTL interface. The same technique also applies for simple peripherals, such as display controllers. The implementation of the front-end is hence general both with respect to the target resource and to the type of arbiter, as long as they are predictable.

2.2 Abstraction

The memory controller is required to use a common abstraction that captures the temporal behavior of many different memory and arbiter types to mitigate the increasing system complexity. We have chosen Latency-Rate ($\mathcal{LR}$) servers [118] as the shared resource abstraction in this work. In essence, a $\mathcal{LR}$ server guarantees a requestor a minimum allocated bandwidth, b', after a maximum service latency,

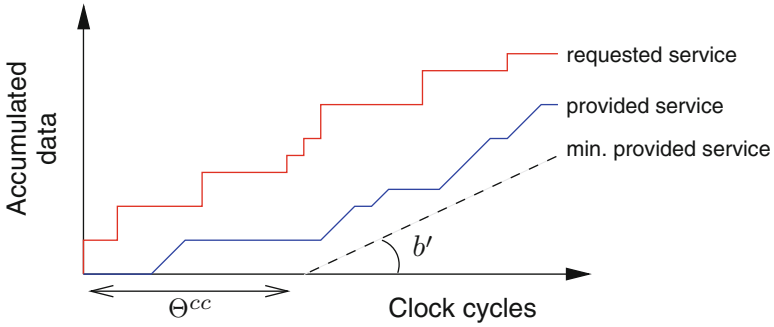

Fig. 2.7 The $\mathcal{LR}$ server abstraction

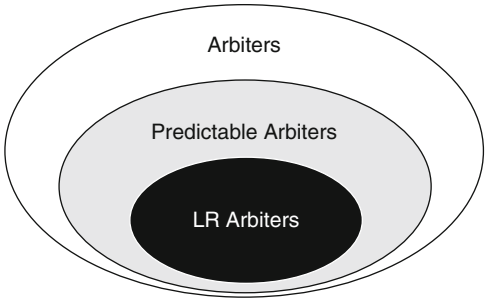

Fig. 2.8 $\mathcal{LR}$ arbiters are a subset of predictable arbiters

Θ^{cc}, as shown in Fig. 2.7. A $\mathcal{LR}$ server hence provides a lower bound on the amount of data that can be transferred during an interval, making it an abstraction of predictable service.

The $\mathcal{LR}$ server model applies to a wide range of shared resources, which is required by our chosen abstraction. In theory, all predictable arbiters belong to the class of $\mathcal{LR}$ servers, since they guarantee that a request is scheduled within a maximum latency, making them starvation free. However, no arbiter truly belongs to the class until the service latency has been derived, which is difficult for some arbiters. The arbiters that belong to the class of $\mathcal{LR}$ servers are hence a subset of the set of predictable arbiters, as illustrated in Fig. 2.8. In this work, we refer to arbiters in the class of $\mathcal{LR}$ servers as $\mathcal{LR}$ arbiters. It is shown in [118] that many well-known arbiters, such as Weighted Round-Robin [66], Deficit Round-Robin [111], and several varieties of Fair Queuing [140] are $\mathcal{LR}$ arbiters. Another example of a commonly used $\mathcal{LR}$ arbiter is TDM [90]. The applicability of the $\mathcal{LR}$ model with respect to resources is very good, since it can be used with any predictable resource. Example uses of the model in literature involve modeling communication channels in buses [128] and networks-on-chip [49].

The $\mathcal{LR}$ server model uses two parameters, service latency and allocated bandwidth, to model the service provided by a shared resource. The model is hence more sophisticated than a model with a single parameter that only considers the

maximum time to serve a request and uses this for every resource access. The added value of the $\mathcal{LR}$ model is that it considers the service history of a requestor. This allows it to exploit the fact that many requests from a requestor may be waiting for service at a particular time, and that all of them cannot experience worst-case interference from other requestors. This allows *tighter bounds* to be derived on the time required to serve a sequence of requests, as shown in [49]. It is possible to conceive using more than the two parameters used by the $\mathcal{LR}$ server model to further improve the accuracy of the model. There are, however, three main reasons not to go in this direction in this work. (1) The $\mathcal{LR}$ model has been shown to apply to many well-known arbiters. This body of work would not necessarily be reusable by a more refined model. (2) It may be more difficult to prove that a particular arbiter belongs to a class with more parameters. (3) Having more parameters makes it more difficult to specify requestor requirements. This is important since requirements often have to be specified manually. Getting the requestor specification may hence involve significant manual labor that has to be repeated whenever changes are made to an application.

A benefit of the $\mathcal{LR}$ server abstraction is that it supports formal performance analysis using approaches based on network calculus [28] or data-flow analysis [113]. This enables formal verification of real-time requirements in a transparent manner for *any combination* of arbiter in the class of $\mathcal{LR}$ servers and predictable resource using any of these frameworks. A limitation of predictability is that some applications have behavior that is too complex to model accurately using formal models, and have to be verified by simulation. To reduce the verification effort of these applications, our memory controller also provides composable service, as discussed next.

2.3 Composability

The memory controller is required to provide composable service to applications to enable them to be developed and verified independently, as explained in Sect. 1.3.3. Composability requires that applications are independent in both the value and time domains. The proposed memory controller only explicitly addresses composability in the time domain. Applications must hence be unable to change each other's temporal behavior, positively or negatively, with even a single clock cycle. We assume that applications are composable in the value domain by some other mechanism, and cannot affect each other's behavior. An example of such a mechanism is to map applications to different, potentially protected, memory regions. Composability affects the design of all hardware and software where applications can interfere with each other temporally, such as stateful resources and most run-time schedulers. We already mentioned SDRAM as an example of a stateful resource, where requestors can interfere with each other's temporal behavior by activating and precharging rows and changing direction of the data bus. Another example is caches, where requestors can evict each other's cache lines, resulting in increased memory latency.

There are currently three approaches to composable system design. The first involves not sharing any resources, which is trivially composable, but prohibitively expensive for systems not running safety-critical applications. The second is to statically schedule all interaction between components in the system [69]. This approach requires a global notion of time and is limited to applications that can be statically scheduled. The third is to share resources dynamically at run-time using TDM [17, 48]. However, this approach is very inefficient for resources with highly variable latency, such as SDRAM, especially in presence of latency-sensitive requestors [9].

In this work, we present a fourth approach to composable resource sharing that is based on the $\mathcal{LR}$ server abstraction, previously presented in Sect. 2.2. The major advantage of this approach is that is extends the use of composability beyond resources and arbiters that are inherently composable. Our approach is hence not limited only to stateless SRAM controllers, but can capture the behavior of any predictable resource, such as our proposed SDRAM back-end based on predictable memory patterns. It furthermore supports any arbiter in the class of $\mathcal{LR}$ servers, enabling service differentiation that increases the possibility of satisfying a given set of requestor requirements. A key benefit is that the approach does not have *any restrictions* on the applications. This ensures that all applications that cannot be formally verified can be verified independently by simulation with a linear verification complexity.

The main problem with non-composable resources and arbitration is that they cause the time to serve a read or a write request to depend on other requestors. This might cause an application that has been verified in isolation to miss deadlines after being integrated with other applications due to contention for shared resources. The key idea behind our approach is to make the system composable by delaying all signals sent to the requestor to *emulate maximum interference* from other requestors. A requestor hence always receives the same worst-case service no matter what other requestors are doing, decoupling their temporal behaviors. Intuitively, it may seem sufficient to verify that the applications meet their real-time requirements under worst-case conditions and then disable emulation of worst-case interference after verification to benefit from improved performance. However, this intuition assumes that applications executing on the system are *performance monotonic* [72] and that having additional resources cannot result in worse performance. This only holds for applications that do not exhibit timing dependent behavior executing in systems that are free from timing anomalies [40], which may occur in shared caches, dynamically scheduled processors [74], and some multi-processor systems [40]. We propose to always emulate maximum interference to avoid restricting the range of supported systems and applications.

Our approach to composable resource sharing makes the temporal behaviors of requestors independent of each other, thus implementing composability at the level of requestors. This is a sufficient condition to be composable at the level of applications, which is our actual requirement. However, composability at the level of requestors is a stricter requirement, since requestors belonging to the same application are allowed to interfere with each other in a composable system.

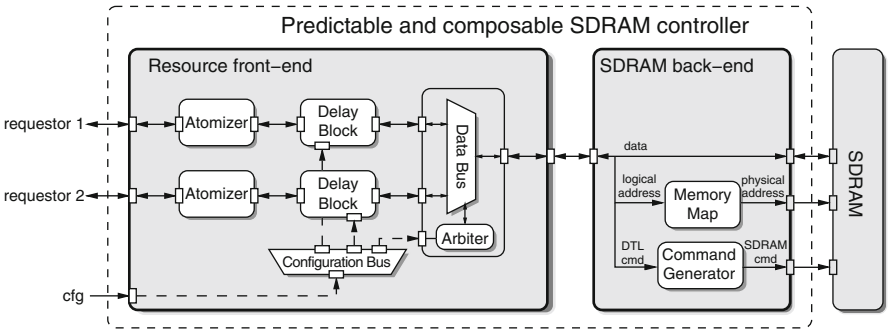

Fig. 2.9 An instance of a predictable and composable SDRAM controller, supporting two requestors

A drawback of our approach is hence that it is not possible to benefit from unused resource capacity reserved by requestors belonging to the same application (slack). However, a feature of our approach is that it can be dynamically enabled or disabled per requestor at run-time by turning the emulation of worst-case interference on or off. Composable service can hence be provided to only a subset of the applications, while providing predictable service to the rest. We refer to this type of system as a *partially composable system*. This type of system enables slack to be used by requestors that do not require composable service, such as non-real-time requestors, or those belonging to applications that are verified using formal approaches. The slack may be used by these requestors to improve performance or reduce power [83]. Partial composability is also interesting if the provider of a system wants to isolate the applications shipped with the system from those developed by third parties. In this case, applications shipped with the platform would have composable service, while it is up to third party to decide between using slack and composability. This creates a separation of concerns between different suppliers, making responsibilities more clear.

We implement this concept by adding an additional component, called a Delay Block, to the architecture in Fig. 2.6. This component embeds the Request and Response Buffers and contains the additional functionality to implement composable service according to our approach. The refined architecture, providing both predictable and composable service, is shown in Fig. 2.9. The purpose of the Delay Block is to emulate worst-case interference from other requestors, thus providing a composable interface towards the Atomizer. This makes the interface of the entire front-end composable, since the Atomizer is not shared. The Delay Block is composable if all signals sent from the Delay Block to the Atomizer exhibit composable behavior, which implies that both the response data and the flow-control signals must emulate maximum interference. This is achieved by computing the latest possible time this information can be sent, using the lower bound on service provided by the $\mathcal{LR}$ server abstraction.

To provide composable service, a Delay Block needs information about the maximum interference that can be experienced by its requestor. This information is typically different for all requestors and changes between use-cases. A *Configuration Bus* is hence added to the architecture, as shown in Fig. 2.9, that allows the worst-case interference to be programmed.

2.4 Automation

The memory controller is required to have an automated approach to finding instantiation parameters and configuration settings for its components to reduce design time. To satisfy this requirement, we have developed a configuration flow, shown in Fig. 2.10. This flow derives the instantiation parameters for all hardware blocks in the memory controller, as well as programmable configuration settings. The purpose of the configuration flow is to derive instantiation and configuration parameters that satisfy the requirements of all requestors for all use-cases. There may be many possible configurations that satisfy the requirements for a given use-case, in which case we prefer the configuration that produces the largest amount of slack bandwidth. The rationale behind this decision is that a configuration with more slack bandwidth is likely to provide better average performance for requestors that do not require composable service. The inputs to this flow are the requestor requirements, being the required minimum bandwidth and maximum service latency, and the timing specification of the memory device. We proceed by discussing the different steps in this flow.

The first step of the flow is to generate a set of memory patterns, assuming that the memory is an SDRAM. Otherwise, a specification is provided that represents the timing behavior of the particular memory. The second step in the flow is normalization of requestor requirements, which implies transforming the bandwidth and service latency requirements to make them independent of the memory device. To accomplish this, the original requirements and the generated memory patterns are

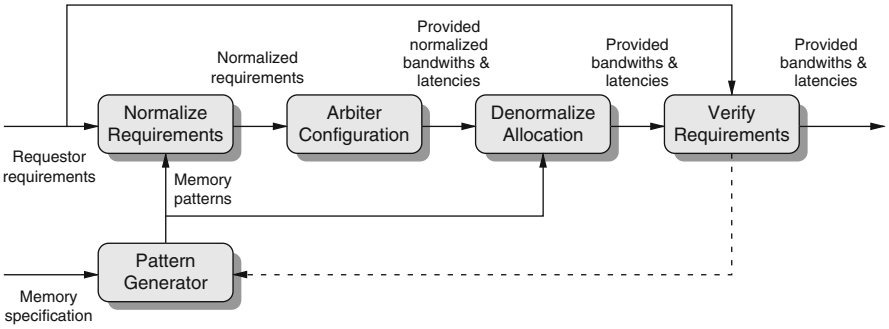

Fig. 2.10 Simplified overview of the automated configuration flow

required as input. The advantage of this step is that arbiter configuration becomes independent of the memory device, allowing the same configuration tool to be used for all memories. The normalized service latency requirement is expressed as the number of interfering atoms that can maximally be tolerated, which can be computed given the lengths of the memory patterns. Normalization of the required bandwidth implies expressing the requirement as a fraction of the total bandwidth offered by the memory. The third step is the arbiter configuration, which attempts to find arbiter settings that satisfy the normalized requirements. The implementation of this step is arbiter dependent. For a TDM arbiter, it involves finding a suitable TDM schedule, while a Round-Robin arbiter does not require any configuration at all. The output of the arbiter configuration is the normalized allocated (provided) bandwidths and service latencies, resulting from the chosen configuration parameters. The fourth step in the configuration flow is denormalization of the allocated bandwidths and service latencies. In addition to the output from the arbiter configuration, the memory patterns are required to convert the normalized allocation back into regular bandwidths and service latencies. The denormalized service allocation, being the provided bandwidths and latencies, is output from this step. The fifth step accepts the denormalized service allocation as input and verifies that the original requestor requirements are satisfied. If all requirements are met, the configuration is stored as a candidate configuration for the use-case. At this point, the flow may iterate to evaluate another set of memory patterns. After all interesting pattern sets have been evaluated, the configuration providing the most slack bandwidth is chosen.

The proposed dimensioning and configuration flow finds all parameters for instantiation and configuration of the memory controller. However, both the memory pattern generation algorithm and the arbiter configuration are heuristic, and are hence not guaranteed to find parameters that satisfy all requirements, even if they exist. However, the size of the design space is so large even for individual steps, such as the memory pattern generation, that optimal solutions are not considered feasible.

2.5 Summary

This chapter discussed how the proposed memory controller and its associated tooling deliver on the four requirements introduced in the previous chapter: *predictability*, *abstraction*, *composability*, and *automation*. First, we presented an approach to predictability, based on combining predictable resources with predictable arbitration. We showed how to make an SDRAM memory behave in a predictable manner using *memory patterns*, which are precomputed sequences of SDRAM commands. There are five types of memory patterns: read patterns, write patterns, read/write switching patterns, write/read switching patterns, and refresh patterns. These patterns are dynamically instantiated and combined at run-time by a proposed SDRAM back-end. To allow our SDRAM back-end to be shared among multiple requestors, a predictable arbiter, suitable for providing access to

shared memories, is needed. We explained that a predictable arbiter needs to use a combination of *rate regulation* and *preemption* to provide guaranteed service in a robust manner in presence of uncooperative or misbehaving requestors.

We presented *Latency-Rate ($\mathcal{LR}$) servers* as our shared resource abstraction. A $\mathcal{LR}$ server is an abstraction of predictability that uses two parameters to describe a lower linear bound on the amount of data that is transferred in an interval. The $\mathcal{LR}$ server model is very general and *applies to any predictable resource*, such as SRAM controllers or our proposed SDRAM back-end. It furthermore *supports many well-known arbiters*. An important benefit of the $\mathcal{LR}$ server model is that it is compatible with several commonly used formal performance analysis frameworks, such as network calculus and data-flow analysis. Any combination of supported resources and arbiters can hence be used transparently with any of these frameworks.

Some applications have behaviors that are too complex to model accurately using formal models, and have to be verified by simulation. Composability is required to reduce the verification complexity for these applications. However, existing approaches to composable system design are either restricted to applications that can be statically scheduled, or share inherently composable resources using time-division multiplexing, which is very inefficient for resources with highly variable latency, such as SDRAM, especially in presence of latency-sensitive requestors. We presented a new approach to composable resource sharing, based on the $\mathcal{LR}$ server abstraction. The key idea is to delay all signals sent from the resource to a requestor to *emulate maximum interference* from other applications. A benefit of our approach is that it can be dynamically enabled or disabled per requestor at run-time. This *enables slack bandwidth to be used to improve performance of requestors that do not require composable service*. However, the biggest advantage of this approach is that is extends the use of composability to work with *any application* sharing *any combination* of predictable resource and arbiter in the class of $\mathcal{LR}$ servers. This approach is implemented as a resource front-end that is located in front of a predictable resource, such as our SDRAM back-end.

The composable resource front-end and SDRAM back-end are supported by a configuration tool that *automatically computes memory patterns and arbiter settings*. The tool uses abstraction to separate the configuration of the memory and the arbiter. The tool hence only knows how to configure the supported SRAM or SDRAMs and each of the arbiters, but can compute configurations that satisfy bandwidth and latency requirements for any combination.

Chapter 3
SDRAM Memories and Controllers

The journey towards a predictable and composable SDRAM controller is started with some background information on both SDRAM memories themselves and their controllers. First, the architecture and temporal behavior of SDRAM memories are introduced in Sect. 3.1. A formal model is then presented in Sect. 3.2 that enables us to formally describe our techniques in later chapters. The concept of memory efficiency is introduced in Sect. 3.3, as we explain why it is difficult to bound the bandwidth and latency of an SDRAM at design time. A general memory controller architecture is presented in Sect. 3.4, and its main functional blocks are discussed. For each of these blocks, different design options are highlighted along with their impact on the provided bandwidth and latency. Lastly, we conclude the chapter with a summary in Sect. 3.5.

3.1 Introduction to SDRAM

Random Access Memory (RAM) is a fundamental component in computer systems and has been for the past decades. It is used as intermediate storage for the processing elements in the system. There are several types of RAM targeting different requirements on bandwidth, power consumption, and manufacturing cost. This work focuses on two common types of RAMs: Static RAM (SRAM) and Dynamic RAM (DRAM). SRAM was introduced in 1970 and is typically used as fast on-chip memory that can be accessed with low latency. For this reason, SRAM is often used for caches and scratchpads in the higher levels of the memory hierarchy to boost performance. The drawback of SRAM is cost, since at least six transistors are needed for every bit in the memory array. The DRAM was invented in 1968 by Robert Dennard at IBM [3]. DRAM is considerably cheaper than SRAM, as it needs only one transistor and a capacitor per bit. The capacitor is charged with a high or low voltage to indicate a one or zero, respectively. The term dynamic stems from the fact that the capacitor is leaking current and needs to be refreshed

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_3, © Springer Science+Business Media, LLC 2011

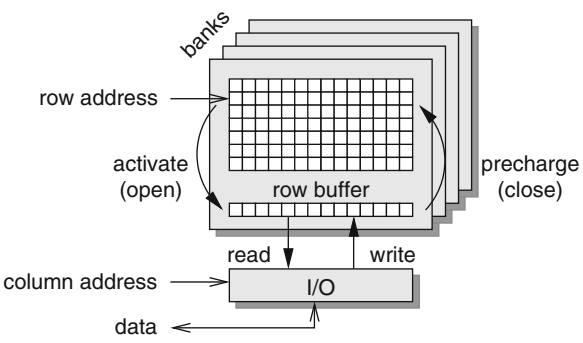

Fig. 3.1 The SDRAM architecture

several hundred times per second to prevent data loss. DRAM is manufactured in an optimized process technology, allowing it to reach high densities and speeds. However, it is typically an off-chip memory, which implies longer latencies and higher power consumption than its on-chip static counterpart. On-chip embedded DRAM exists, but has yet to gain widespread adoption. For these reasons, DRAM is often used as high-volume storage in the lower levels of the memory hierarchy.

In the past 10 years, there have been a number of improvements of the DRAM design. A clock signal has been added to the previously asynchronous DRAM interface to reduce synchronization overhead with the memory controller during burst transfers. This type of memory is called synchronous DRAM, or SDRAM for short. In 2001, a new generation of SDRAM was introduced, featuring significantly higher bandwidth. These memories transfer data on both the rising and the falling edge of the clock, hence the name Double-Data-Rate (DDR) SDRAM. The second and third generations of this memory, called DDR2 [61] and DDR3 [62], respectively, are very similar in design, but scales to higher clock frequencies and bandwidths.

3.1.1 SDRAM Architecture

The architecture of an SDRAM memory contains a number of banks. A bank stores a number of word-sized elements in a two-dimensional structure organized in rows and columns, as shown in Fig. 3.1. In essence, banks are independent memories, but they share a data bus, an address bus, and a command bus to reduce the number of off-chip pins. Each bank has a row buffer that stores one open row. Only the elements of this open row can be accessed by read and write accesses. To give an idea about the number of banks, rows, and columns in a contemporary SDRAM, we choose an example memory that we use throughout this book. This memory is a 512 megabit (Mb) DDR2-400 [61] device with a word width of 16 bits. This device has 4 banks, each with 8192 rows containing 1024 word-sized elements. Since each column holds an element of 16 bits, it follows that a row contains 2 kilobytes (KB) of data. This is referred to as the *page size* of the memory. Multiple devices can

be combined to create wider memory interfaces and increase storage capacity. The clock frequency of our example memory is 200 MHz and data elements can be transferred with a frequency of 400 MHz, due to the double data rate.

If we compare our example memory to other DDR2 or DDR3 memories, we notice that the DDR2-400 is the slowest memory of these generations. The reason for using this memory as our running example is that it results in shorter and less complicated memory schedules, increasing the clarity of our presentation. DDR2 memories start with the DDR2-400 memory and ends with the DDR2-800 running at 400 MHz. This is where the DDR3 generation begins with the DDR3-800 and the standard specifies up to DDR3-2133, which runs at 1,066 MHz. Except for the increase in clock frequencies, there are few differences between the DDR2 and DDR3 generations of SDRAM that are relevant to this work. We will point out these differences where applicable. The typical number of banks in a DDR memory is 4 or 8. All DDR3 memories have 8 banks, while it is depends on the density of the memory for DDR2. DDR2 devices with a density less than 1 Gb have 4 banks, while the larger ones have 8 banks. The memory devices are available with widths of 4, 8, and 16 bits, and specified capacities are 256 Mb to 8 Gb.

3.1.2 The SDRAM Protocol

An SDRAM is controlled by sending SDRAM commands to the memory interface according to the SDRAM protocol. The protocol contains six commands: *activate* (ACT), *read* (RD), *write* (WR), *precharge* (PRE), *refresh* (REF), and *no-operation* (NOP). We continue by discussing the function of each of these commands.

The activate command is issued with a row and a bank as argument, instructing the chosen bank to copy the requested row to its row buffer. Once the requested row is opened, column accesses, such as read and write bursts, can be issued to access the columns in the row buffer. These bursts have a burst length (BL) of 4 or 8 words. The burst length of a DDR2 memory is programmed when the memory is initialized, while a DDR3 allows it to be changed on the fly for every access. A burst length of 4 words is only supported by a burst chopping mechanism on DDR3 devices [62]. Such a chopped burst requires the same time as a burst of 8 words, but only transfers data during half the time. The read and write commands have the target bank, row, and column sent as arguments. The precharge command is the converse of the activate command, as it copies the contents of the row buffer back into its place in the memory array. This operation is required even if the data in the row buffer have not been modified, since activates are destructive to the contents of the opened row. Read and write commands can be issued with an *auto-precharge* flag, resulting in an automatic precharge at the earliest possible moment after the transfer is completed. This has the benefit of allowing a new arbitrary row to be opened as quickly as possible without causing contention on the shared command bus. The refresh command must be issued regularly to prevent the leaking capacitors from losing data. Multiple refresh commands are required to refresh the

Table 3.1 List of relevant timing parameters for a 64 MB x16 (512 Mb) DDR2-400 memory device

Parameter	Description	DDR2-400 (cycles)
tRC	Row cycle time. Minimum time between successive activate commands to the same bank	11
$tRCD$	Minimum time between activate and read/write commands on the same bank	3
tCL	CAS latency. Time after read command until first data is available on the bus	3
tWL	Write latency. Time after write command until first data is available on the bus	2
tRP	Minimum time between a precharge command and an activate command to the same bank	3
$tRFC$	Minimum time between a refresh command and a successive refresh or activate command	21
$tRAS$	Minimum time after an activate command to a bank until that bank is allowed to be precharged	8
$tRTP$	Minimum time between a read and precharge command	2
tWR	Write recovery time. Minimum time after the last data has been written until a precharge may be issued to the same bank	3
$tFAW$	Window in which maximally four banks may be activated	10
$tRRD$	Minimum time between activates to different banks	2
$tCCD$	CAS to CAS command delay. Minimum time between two read commands or two write commands	2
$tWTR$	Internal write to read command delay	2
$tREFI$	Average refresh interval	1560

entire memory array, as each individual command only refreshes a fraction of the capacitors. However, no argument is required by this command, since an internal counter supplies the appropriate address. All banks must be precharged before the refresh command is issued. The last command is the no-operation command, which is issued if no other command is required during a cycle. Figure 3.1 illustrates the behaviors of some of these commands.

3.1.3 Timing Constraints

There are many timing constraints and parameters that decide which SDRAM commands that can be issued during a particular cycle. The constraints are typically specified as minimum delays between successive commands. Table 3.1 lists all relevant constraints for our example memory. Detailed descriptions of all constraints for DDR2/DDR3 memories are provided in [61, 62]. The meanings of some of the constraints are illustrated in Fig. 3.2, which is a valid command sequence for our example memory. The figure shows that at least $tRRD$ cycles have to pass between

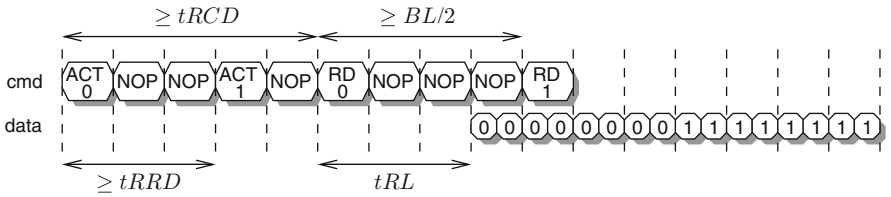

Fig. 3.2 Example of SDRAM timing constraints

consecutive activates to different banks. It also shows that at least $tRCD$ cycles have to pass from issuing an activate command before a read or write command is sent to the same bank. A read or a write command causes data to be sent over the data bus during $BL/2$ clock cycles with a DDR memory. This means that successive read or write commands must be scheduled at least $BL/2$ clock cycles apart to prevent a conflict on the data bus. This is seen in the figure, where the burst length is programmed to eight words. The first read and write data appears on the data bus a number of cycles after the corresponding command has been issued. This time is referred to as the read latency, tRL, and write latency, tWL, respectively. The figure shows that the read latency is 3 cycles for our example memory.

3.2 Formal Model

We proceed by introducing the formal model used in this book, which allows us to formally describe some of our techniques. For simplicity, we build up this model incrementally and add more content in later chapters. A complete list of the symbols used in this book along with brief descriptions and page references to the definitions are found in Appendix B.2. We start by introducing our choice of notation. Throughout this book, we use capital letters (A) to denote sets, hats to denote upper bounds ($\hat{a}$), and checks to denote lower bounds ($\check{a}$). We use subscripts to disambiguate between variables belonging to different requestors, although for clarity these subscripts are omitted when they are not required. We use $\mathbb{N}$ to denote the set of non-negative integers, and $\mathbb{N}^+$ to denote the set of positive integers. Time is discrete and counts from zero.

We start by defining the architecture of a memory in Definition 3.1. This definition is quite general and does not only describe the architecture of SDRAM, but also SRAM. A typical SRAM architecture has a single bank, a data rate of one word per cycle, and a burst length of one word. The clock frequency and data width depends on the design in which it is used. The only timing parameter we are interested in for SRAMs is the clock frequency. The more elaborate definition of the timing behavior of an SDRAM is provided in Definition 3.2.

Definition 3.1 (Memory architecture). The architecture of a memory is defined as (n_{banks}, w_{mem}, f_{mem}, dr, BL), where n_{banks} is the number of banks, w_{mem} is the width of the data bus in bytes, f_{mem} is the clock frequency of the memory in MHz, dr is the number of data words that can be transferred during a clock cycle, and BL is the programmed burst length in words.

Definition 3.2 (SDRAM timings). The timings of an SDRAM, measured in clock cycles, are defined as (tRC, $tRCD$, tCL, tWL, tRP, $tRFC$, $tRAS$, $tRTP$, tWR, $tFAW$, $tRRD$, $tCCD$, $tWTR$, $tREFI$), where the parameters are defined according to Table 3.1.

We proceed by adding definitions for requestors and memory requests. These enable our discussions on memory efficiency in this chapter and the next. The memory is shared between a set of requestors, as stated in Definition 3.3. A requestor generates a sequence of memory requests, defined in Definitions 3.4 and 3.5, that may be either reads or writes. These requests may have variable size, as expressed by Definition 3.6. A read or a write burst is only allowed to start at an address that is an integer multiple of the programmed burst length. The alignment of a request is defined as the offset of the targeted address with respect to the start of the burst, as defined in Definition 3.7.

Definition 3.3 (Set of requestors). The set of requestors sharing the memory is denoted by R.

Definition 3.4 (Set of requests). The set of requests from a requestor $r \in R$ is denoted by Ω_r.

Definition 3.5 (Request). The k:th request ($k \in \mathbb{N}$) from a requestor $r \in R$ is denoted by $\omega_r^k \in \Omega_r$.

Definition 3.6 (Request size (bytes)). The size of a request ω_r^k in bytes is given by $s^{bytes}(\omega_r^k) : \Omega_r \to \mathbb{N}^+$.

Definition 3.7 (Request alignment). The alignment of a request ω_r^k in bytes is given by $a(\omega_r^k) : \Omega_r \to \mathbb{N}$, and is defined as $a(\omega_r^k) = \alpha(\omega_r^k) \mod (BL \cdot w_{mem})$, where $\alpha(\omega_r^k)$ is the address of ω_r^k in bytes.

3.3 Memory Efficiency

The bandwidth offered by a memory ideally corresponds to the product of the width of the memory interface, the clock frequency of the memory, and the data rate. This is referred to as the *peak bandwidth* of the memory, defined in Definition 3.8. Our example DDR2-400 memory has a peak bandwidth of 800 MB/s, since it has a clock frequency of 200 MHz, a data rate of 2 words per clock cycle, and a data bus width of 16 bits. A Zero-bus-turnaround (ZBT) SRAM has no problems with achieving its peak bandwidth, due to its constant access latency. However, the

peak bandwidth of SDRAMs typically cannot be fully utilized, due to stall cycles caused by the timing constraints of the memory. This is captured by the concept of *memory efficiency*. Memory efficiency corresponds to the fraction of cycles data is transferred to and from the memory. A useful classification of memory efficiency into five categories is presented in [136]. The categories are: (1) refresh efficiency, (2) read/write efficiency, (3) bank efficiency, (4) command efficiency, and (5) data efficiency. We continue by explaining each of the categories of memory efficiency, discuss what they depend on, and try to estimate their impact in the general case.

Definition 3.8 (Peak bandwidth). The peak bandwidth of a memory device is denoted by b^{peak}, and is defined as $b^{peak} = f_{mem} \cdot dr \cdot w_{mem}$.

3.3.1 Refresh Efficiency

Refresh efficiency, e^{ref}, accounts for the cycles that are lost due to refreshing the capacitors in the memory array. This efficiency depends on the time required to precharge all banks, the time to complete the refresh command itself, and the refresh period. The refresh command requires $tRFC$ cycles to complete after it has been issued. The value of this parameter is determined by the size of the memory device, as larger devices require more time to refresh. The refresh command must be issued every $tREFI$ cycles on average, corresponding to 7.8 µs for all DDR2 and DDR3 devices at normal operating temperatures. The only uncertainty when determining refresh efficiency is the time required to precharge all banks, which depends on the state of the memory. The refresh efficiency can hence be estimated at design time with reasonable accuracy. Typically, the refresh efficiency is between 95 and 99% for all DDR2 and DDR3 memories.

3.3.2 Read/Write Efficiency

SDRAMs have a bi-directional data bus that requires time to switch from read to write and vice versa. This results in lost cycles as the direction of the data bus is being reversed. To use the data bus of a DDR SDRAM at maximum efficiency, a read or write command must be issued every $BL/2$ cycles. We quantify the cost of switching directions as the number of extra cycles on the command bus before the read or write command can be issued. As an example, the cost of a read/write switch and a write/read switch using our example DDR2-400 is 2 and 4 cycles, respectively. The read/write efficiency, e^{rw}, depends on the number of read/write switches, which cannot typically be determined at design time. However, a formula is presented in [136] that computes the average read/write efficiency, based on a long-term read/write ratio. As an example, the average read/write efficiency for traffic consisting of 70% reads and 30% writes with $BL = 8$ equals 93.8%. Note that

the worst-case read/write efficiency must be considerably lower, since a long-term read/write ratio cannot exclude that there are long intervals where there is a switch after every single burst.

3.3.3 Bank Efficiency

The access time of an SDRAM is highly variable. A read or write command can be issued immediately to columns in the active row. However, if a command targets an inactive row, it first requires a precharge followed by an activate command. This requires at least an additional $tRP + tRCD$ cycles (6 for our example memory) before the read or write command can be issued. The penalty can be even larger, as tRC cycles must separate one activate command from another within the same bank, according to Table 3.1. This overhead is captured by bank efficiency, e^{bank}. Bank efficiency is highly dependent on the target addresses of requests, and how they are mapped to the different rows and banks of the memory. Therefore, it is not possible to give a general estimate on the impact of this efficiency.

3.3.4 Command Efficiency

Even though a DDR device transfers data on both the rising and the falling edge of the clock, commands can only be issued once every clock cycle. Sometimes a required activate or precharge command has to be delayed because another command is already issued in that clock cycle. This results in lost cycles when a read or write command has to be postponed due to a row miss. The impact of this is connected to the burst length, as smaller bursts result in more activate and precharge commands. Command efficiency, e^{cmd}, is traffic dependent and can generally not be calculated at design time, but is estimated in [136] to be between 95 and 100%.

3.3.5 Data Efficiency

Data efficiency, e^{data}, is defined as the fraction of a memory access that actually contains requested data. This can be less than 100%, since SDRAM memories are accessed with a minimum burst length; 4 words for DDR2 and 8 for DDR3 SDRAM (since 4 words is only supported by chopping bursts of 8 words). The problem is not only related to fine-grained requests, but also to how data is aligned with respect to a memory burst. This is because a burst is required to access BL words from an address that is evenly divisible by the burst length. This is illustrated in Fig. 3.3. The data efficiency of a requestor can be computed at design time if the minimum access granularity of the memory, and the size and alignment of requests are known.

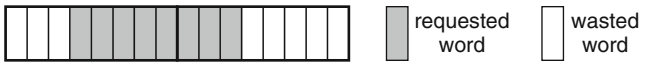

Fig. 3.3 Two bursts of 8 words are required to read or write 8 words that are misaligned

For example, if requests are aligned cache lines of 128 B from an L2 cache then the data efficiency may be 100%. On the other hand, [136] computes a data efficiency of 75% for an MPEG2 stream. However, the overall data efficiency of the memory depends on how many requests from a particular requestor that is scheduled in an interval, which is determined by the arbiter and may hence depend on traffic.

3.3.6 Gross and Net Efficiencies

Having discussed all categories of memory efficiency, we proceed by distinguishing two different types of the concept. Definition 3.9 defines *gross memory efficiency* as the product of all categories of memory efficiency, excluding data efficiency. This metric hence does not care if the data on the bus is wanted by any of the requestors or not. *Gross bandwidth*, defined in Definition 3.10, hence accounts for all data that passes the data bus in an interval. This metric is primarily relevant if the data efficiency is unknown or uninteresting. Note that all categories of gross memory efficiency are traffic dependent, making it very difficult to determine the gross bandwidth at design time in the general case.

Definition 3.9 (Gross memory efficiency). Gross memory efficiency is denoted by e^{gross}, and is defined as $e^{gross} = e^{ref} \cdot e^{rw} \cdot e^{bank} \cdot e^{cmd}$.

Definition 3.10 (Gross bandwidth). The gross bandwidth of a memory device is denoted by b^{gross}, and is defined as $b^{gross} = b^{peak} \cdot e^{gross}$.

Definition 3.11 defines *net memory efficiency* as the product of all categories of memory efficiency, thus including data efficiency. This hence corresponds to the fraction of clock cycles with useful data requested by a requestor on the data bus. Net memory efficiency is used to determine the *net bandwidth* provided by the memory controller, defined in Definition 3.12. Net bandwidth is an important concept, since the bandwidth requirements of the requestors have to be satisfied according to this definition of bandwidth, as stated in Definition 3.13. This implies that the net bandwidth allocated to the requestor in the memory controller, defined in Definition 3.14, must be at least equal to the requested bandwidth.

Definition 3.11 (Net memory efficiency). Net memory efficiency is denoted by e^{net}, and is defined as $e^{net} = e^{ref} \cdot e^{rw} \cdot e^{bank} \cdot e^{cmd} \cdot e^{data}$.

Definition 3.12 (Net bandwidth). The net bandwidth of a memory device is denoted by b^{net}, and is defined as $b^{net} = b^{peak} \cdot e^{net}$.

Table 3.2 Comparison of timing constraints in nanoseconds and clock cycles for a DDR2-400 and a DDR3-1600

	DDR2-400		DDR3-1600	
Constraint	(ns)	(cc)	(ns)	(cc)
tRC (ACT-ACT same bank)	55	11	45	36
$tRRD$ (ACT-ACT diff. banks)	7.5	2	6	5
$tRCD$ (ACT-RD/WR)	15	3	10	8
tRP (PRE-ACT)	15	3	10	8

Definition 3.13 (Requested bandwidth). The net bandwidth requested by a requestor $r \in R$, expressed in MB/s, is denoted by b_r.

Definition 3.14 (Allocated bandwidth). The allocated net bandwidth of a requestor $r \in R$, expressed in MB/s, is denoted by b'_r.

3.3.7 Memory Efficiency Trend

An interesting trend in SDRAMs is that the actual timing behavior does not change much between generations. This trend is clearly visible if the timing constraints, measured in nanoseconds, are compared between newer and older memories. However, newer memories are clocked at higher and higher frequencies, resulting in that the timing constraints, measured in clock cycles, are increasing. We illustrate this point in Table 3.2 by comparing the timings of a DDR2-400 to the four times faster DDR3-1600 in both nanoseconds and clock cycles. We note that the activate-to-activate delay for a bank, tRC, is reduced with 10 ns for the DDR3-1600, corresponding to a reduction of about 20%. However, the same delay measured in clock cycles is more than three times larger than for the DDR2-400! We note that similar trends are visible for the other timing parameters. The bottom line of this trend is that the number of clock cycles with data transfer remains constant as memories get faster, while the number of overhead cycles is increasing. It hence follows that memory efficiency of SDRAMs is decreasing for faster memories. We will later experimentally demonstrate this trend in Sect. 4.7.

3.4 Memory Controllers

There are a large number of memory controller designs with different design goals and distinguishing features. However, most memory controllers consist of the same basic building blocks. In this section, we present an overview of a general SDRAM controller design. Such a controller can be partitioned into two parts: a front-end, and a back-end, as illustrated in Fig. 3.4. The front-end is memory independent and

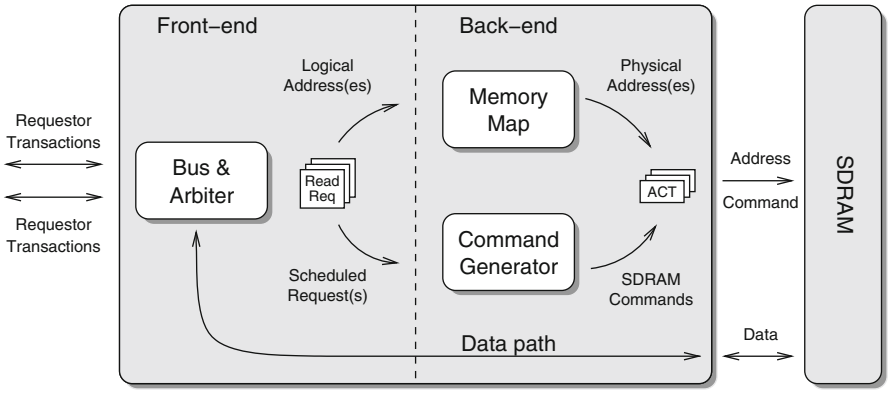

Fig. 3.4 The most important building blocks of a general SDRAM controller

primarily buffers incoming requests from the requestors, schedules access to the back-end, and returns responses. It hence works on the granularity of transactions. The back-end, on the other hand, is dependent on the memory device and needs to be replaced, modified, or reprogrammed if the memory changes. The back-end is primarily responsible for the translation between the protocol of the requestors and that of the memory. The back-end of an SDRAM controller hence works with both requestor transactions and SDRAM commands. There are four main building blocks in an SDRAM controller: (1) a bus and arbiter, (2) a command generator, (3) a memory map, and (4) a data path. The bus and arbiter are located in the front-end, while the command generator and memory map are in the back-end. The data path goes through both the front-end and the back-end. We proceed by describing the function of the first three blocks and explain some important design options that impact important characteristics of the controller, such as the provided net bandwidth and latency. The data path is a necessary part of the controller, but we do not discuss it further, since there are no interesting design considerations that are relevant to this work. We keep an open mind in this section and discuss options without making any decisions for our own design. These decisions are postponed until Chap. 4 when we present our predictable SDRAM back-end.

3.4.1 Bus and Arbiter

The bus is responsible for scheduling access to the back-end according to the policy of the attached arbiter. The arbiter can work in a variety of ways, but typically makes decisions based on bandwidth and latency requirements of the requestors. We categorize arbiters into two classes: static and dynamic, depending on if the scheduling is done at design time or at run time. The main advantage of computing a static schedule at design time is that the maximum number of interfering requests

can be bounded by examining the schedule, making the arbitration predictable. The disadvantage is that they cannot handle latency-sensitive requestors or requestors with small bandwidth requirements without wasting bandwidth. The reason for this wasted bandwidth is that static front-end arbiters are only able to reduce latency of a requestor by assigning it more entries in the schedule, thus unnecessarily increasing its allocated bandwidth. The increasing number of use-cases in contemporary systems furthermore causes difficulties with static arbiters. Multiple use-cases are supported by precomputing and storing a separate schedule per use-case. This may take a long time to compute and require a significant amount of space to store if the number of use-cases is large.

Dynamic front-end arbiters make scheduling decisions at run time, allowing them to use information that is not available at design time. This makes them more flexible, but also more difficult to analyze. Useful bounds on latency have not been successfully derived for many dynamic arbiters, making them unpredictable according to our definition of predictability. The three types of arbiters presented in Sect. 1.1.3, Time-Division Multiplexing (TDM), Round-Robin, and static-priority scheduling, are all examples of dynamic front-end arbiters, since scheduling is done at run time. However, TDM has many properties of static arbitration, since it is based on a schedule that is computed at design time. We elaborate further on this in Chap. 5 when discussing resource arbitration.

3.4.2 Command Generator

The command generator is responsible for generating and scheduling SDRAM commands, such that no timing constraints of the memory are violated. Just like front-end arbiters, command generators can be classified as either static or dynamic, depending on how the scheduling is done. A command generator that uses static scheduling simply issues a schedule of SDRAM commands that is precomputed at design time. The command generator is hence a very simple block that does not have to be aware of the state of the memory, since this becomes the responsibility of the tooling that computes the schedules. These command generators are predictable, since the time to serve a request and the provided gross bandwidth can be derived from the schedule at design time. However, the precomputed schedule makes these controllers unable to adapt to changes in traffic. This limits the applicability to requestors with regular access patterns, where the request sizes and read/write ratio do not change during a use-case. Just like in the case of front-end arbitration, static command scheduling implies that a different SDRAM schedule has to be computed and stored for every use-case.

A command generator that uses dynamic scheduling generates the required SDRAM commands for the memory requests sent to the back-end and schedules them according to some algorithm. A common goal is to schedule the commands to maximize the average gross bandwidth and provide low average latency to sensitive requestors. To achieve high efficiency, requests are often scheduled out

of order, depending on how they fit with the state of the memory. Requests that address open rows may hence be preferred over requests that target closed rows to reduce overhead. Similarly, reads or writes may be preferred depending on the current direction of the data bus. Most dynamic command generators address the requirements of latency-sensitive requestors by incorporating priorities into the scheduling algorithm. It is important that command generators that use dynamic scheduling closely tracks of the state of the memory, such that no timing constraints are violated. The particular timings of the target memory device are often programmed into registers, allowing a single command generator to be used with many different SDRAM devices. Dynamic command scheduling is clearly more complex than the static counterpart, both conceptually and in terms of hardware, but it also gives many additional degrees of freedom. This approach automatically adapts to the incoming requests and can hence handle input-dependent applications. It does furthermore not require reconfiguration between use-cases. While dynamic command generation and scheduling has many advantages, it is not without its share of disadvantages. The increased flexibility may increase the average provided bandwidth and reduce average latency, but at the expense of predictability. The provided net bandwidth and latencies are typically not bounded, due to the complex interactions between different mechanisms, and have to be estimated by simulation. This makes it difficult to satisfy requestor requirements, since the bandwidths and latencies have to be reevaluated every time a requestor is added, removed, or changes behavior.

3.4.3 Memory Map

A memory map provides a translation from the logical memory addresses used by the requestors to the physical addresses (bank, row, column), used by the memory device. There are many possible memory mappings and the choice impacts important properties of the memory, such as the average and worst-case offered bandwidths and latencies. We proceed by discussing two commonly used memory maps and highlight their respective advantages and disadvantages.

3.4.3.1 Continuous Memory Map

A *continuous memory map* maps a sequential address space to successive elements in a single row in a single bank. Thus, the same row is accessed over and over again until the end of the row is met. At this point, the mapping switches to a new bank. When there are no more banks, the next access maps to the next row in the first bank. This is illustrated in Fig. 3.5, where a five-bit logical address space is mapped to a toy memory with four banks, two rows, and four columns. The figure also shows which bits in the logical address are used to index the bank (B), row (R), and column (C), respectively.

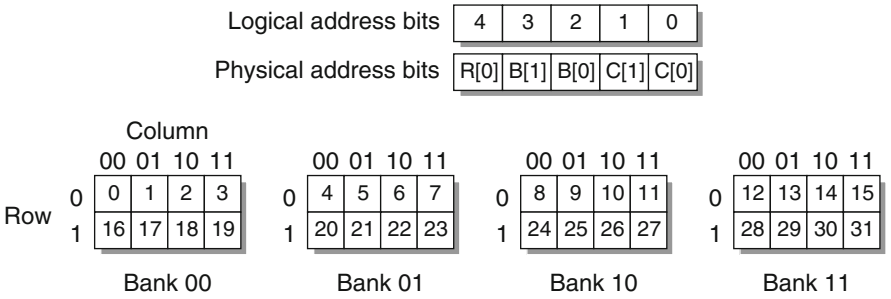

Fig. 3.5 Illustration of a continuous memory map

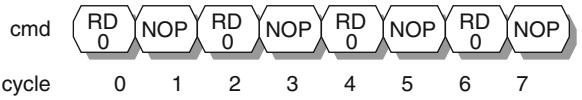

Fig. 3.6 Best case for a requestor reading four independent bursts with $BL = 4$ from a DDR2-400 using a continuous memory map

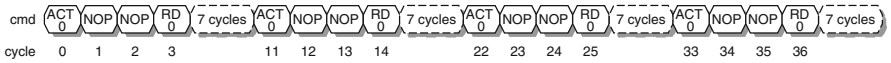

Fig. 3.7 Worst-case for a requestor reading four independent bursts with $BL = 4$ from a DDR2-400 using a continuous memory map

The best case for a continuous memory map is demonstrated in Fig. 3.6 for a single requestor reading four independent bursts with $BL = 4$ using our example DDR2-400 memory. In the best case, all four bursts access the same row in the same bank and the row is already open. The requestor hence reads all four bursts without any need to activate or precharge, finishing all bursts in only 8 cycles. The sequence has a bank efficiency of 100%, since it is eight cycles long and results in eight cycles of data transfer. Note that this command sequence is repeatable and leaves the memory in the same state it was before the four bursts were issued. The next four memory bursts may hence encounter the same best case.

The worst case is when successive bursts target different rows in the same bank. This requires an activate and a precharge command to be issued for every access, as shown in Fig. 3.7. Note that the figure assumes that reads are issued with the auto-precharge flag, since no explicit precharge commands are shown. The activate-to-activate timing constraint for a single bank, tRC, is quite long, significantly increasing latency over the best case. This is seen in Fig. 3.7, where the time required to issue four memory bursts is 44 clock cycles, as opposed to the 8 clock cycles in the best case, corresponding to an increase of 450%! This clearly shows that memory efficiency and latency are highly dependent on spatial locality, and is very high in

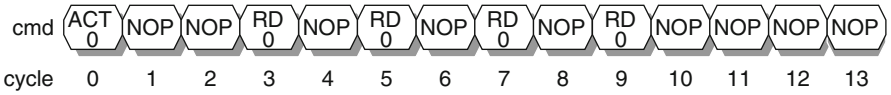

Fig. 3.8 Worst-case command sequence for a request consisting of four bursts to a DDR2-400 using a continuous memory map

the best case, but very poor in worst case. The command sequence in the figure is repeatable, just like for the best case, and the next four bursts may hence encounter the same worst case.

One technique to prevent requestors from ruining each other's spatial locality is to use *bank partitioning* [5]. In this case, each requestor gets exclusive access to one or more memory banks, depending on their required storage capacity, resulting in that their potential spatial locality is preserved. However, a problem with this approach is that it is only guaranteed to work if there is maximally one requestor per bank. Otherwise, the risk of interference between requestors reappears. Since the number of banks is limited to four or eight, this becomes a significant restriction.

Increasing granularity by serving larger requests in a non-preemptive manner is another method of improving worst-case efficiency. Figure 3.8 shows the worst case for requests consisting of four bursts to sequential addresses that are served non-preemptively. In this case, all bursts are guaranteed to access the same row in the same bank, thus only requiring one activate and precharge for every four bursts. Note that there is no explicit precharge command in the figure, since the last read is assumed to be issued with the auto-precharge flag set. The sequence is repeatable and causes data to be transferred during eight cycles, resulting in a bank efficiency of $8/14 = 57\%$. This shows that the bank efficiency increases with the size of the request, as there are more cycles with data transfer to amortize the overhead cycles caused by activating and precharging.

3.4.3.2 Interleaved Memory Map

An *interleaved memory map* is an alternative approach to memory mapping. This mapping sends sequential bursts in the logical address space to different banks. Once all banks have been accessed, bursts are mapped to the following columns in the same rows, until the rows are full. At this time, the interleaving continues over the next rows. This is illustrated in Fig. 3.9, where bursts of two words are interleaved over the banks.

The best and worst cases for a single burst are the same as for the continuous memory map, presented in the previous section. However, the two memory maps behave differently in the worst case if requests are larger than a single burst and are served non-preemptively. The worst-case for a request consisting of four sequential bursts is shown in Fig. 3.10. The time to activate and (auto-)precharge the banks is hidden by the accesses to the other banks. This results in a worst-case bank efficiency of $8/11 = 73\%$, which is higher than the 57% provided by the continuous

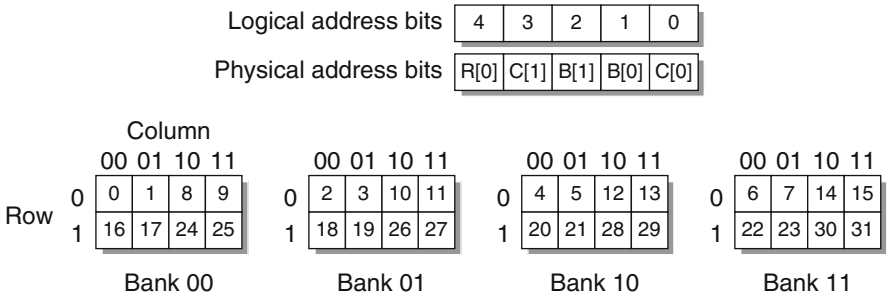

Fig. 3.9 Illustration of an interleaved memory map

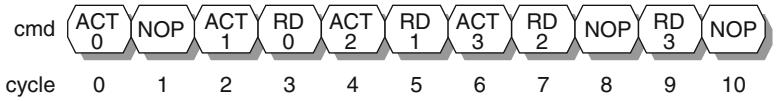

Fig. 3.10 Worst-case command sequence for a request consisting of four bursts to a DDR2-400 using an interleaved memory map

memory map. In fact, an interleaving memory map always provide equal or higher gross memory efficiency than its continuous counterpart, since it exploits bank parallelism. However, a drawback of this memory map is that interleaving over the banks causes more activate and precharge commands to be issued. Both of these commands consume a considerable amount of power [25, 78, 79], resulting in that the total power consumption of the memory device may be higher than with the continuous alternative.

3.5 Summary

Synchronous Dynamic RAM (SDRAM) memories are commonly used as off-chip background memory in contemporary systems, since they provide high storage capacities and reasonable bandwidths at low cost per bit. Many current platforms use Double-Data-Rate (DDR) SDRAM that transfer data elements on both the rising and falling edges of the clock, effectively doubling the bandwidth over its predecessors. This work considers DDR2 and DDR3 SDRAM memories, which are the second and third generations of DDR SDRAM, respectively. These memories are specified from 200 to 800 MHz. The architecture of an SDRAM consists of *banks*, *rows*, and *columns*. Current SDRAM memories have either 4 or 8 banks, which are essentially independent memories, but with shared data, command, and address buses to reduce the number of off-chip pins.

The SDRAM protocol consists of six commands: *activate*, *read*, *write*, *precharge*, *refresh* and *no-operation*. The activate command opens a row in the memory array and stores it in a row buffer. Once the requested row is opened, read and write commands can be issued to access the columns in the row buffer. These bursts have a length of either 4 or 8 words. The precharge command is the converse of the activate command, as it copies the contents of the row buffer back into its place in the memory array. Read and write commands can be issued with an *auto-precharge* flag, resulting in an automatic precharge at the earliest possible moment after the transfer is completed. A DRAM cell stores a bit as a charge in a capacitor. To prevent data from being lost due to leakage, a refresh command must be issued regularly to recharge the capacitor. The last command is the no-operation command, which is issued if no other command is required during a cycle. There are many *timing constraints* that decide which SDRAM commands that can be issued during a particular cycle. These constraints are typically specified as minimum delays between successive commands of different types.

The *peak bandwidth* provided by a memory is determined by the width of its interface, the clock frequency, and the data rate. However, SDRAM memories cannot achieve this bandwidth due to overhead caused by the timing constraints. This is captured by the concept of *memory efficiency*, which is the fraction of the time that the memory controller transfers data. Memory efficiency can be classified into five categories that account for different types of overhead: (1) *refresh efficiency*, (2) *read/write efficiency*, (3) *bank efficiency*, (4) *command efficiency*, and (5) *data efficiency*. All of these categories are traffic dependent for SDRAM and very difficult to bound at design time in the general case. *Gross memory efficiency* is the product of the first four categories of memory efficiency. Multiplying this number with the peak bandwidth determines the *gross bandwidth*, which considers all data that is sent over the data bus of the memory. Similarly, *net memory efficiency* is the product of all categories of memory efficiency. Multiplying this with the peak bandwidth computes the *net bandwidth*, corresponding to the data sent over the data bus that is requested by any of the requestors.

A typical SDRAM controller has three main building blocks: *arbiter*, *command generator*, and *memory map*. The arbiter schedules one or more requests at a time. The command generator generates the appropriate SDRAM commands for the scheduled requests. It also schedules these commands, such that no timing constraints of the memory are violated. The arbiter and command generator either use *static* or *dynamic scheduling*. The advantage of static scheduling is predictability, but it is also less flexible than its dynamic counterpart, resulting in longer latencies and lower memory efficiency. The memory map translates the logical addresses used by the requestors to physical addresses, being the targeted bank, row, and column. Two common ways of doing this is using either a *continuous* or an *interleaved memory map*. An interleaved memory map offers better worst-case bank efficiency than a continuous memory map, as it exploits bank parallelism. However, this benefit comes at the expense of increased power consumption.

Chapter 4
Predictable SDRAM Back-End

Designing a predictable SDRAM controller is challenging. Traditional approaches are based on static scheduling of requests and SDRAM commands. This makes them unsuitable for many applications in contemporary System-on-Chips (SoCs), as they are getting increasingly input dependent and have diverse bandwidth and latency requirements. However, the timing constraints of SDRAM memories make it difficult to support dynamic behavior, since net bandwidth and latencies are traffic dependent and hard to bound at design time. This chapter starts with an overview of our predictable memory controller in Sect. 4.1 by discussing our decisions between static and dynamic arbitration and command generation, and between continuous and interleaved memory maps. The rest of this chapter focuses on the SDRAM back-end, saving the discussion about front-end arbitration for Chap. 5. After the overview, Sect. 4.2 introduces memory patterns, which are a key concept to achieve predictability with SDRAM in our approach. We then show how these patterns enable us to bound gross bandwidth and latency in Sects. 4.3 and 4.4, respectively. Next, three algorithms for automatic memory pattern generation, each offering a different trade-off between memory efficiency and run time, are presented in Sect. 4.5. The architecture and synthesis results of our SDRAM back-end are discussed in Sect. 4.6, before we experimentally evaluate it in Sect. 4.7. Lastly, we conclude the chapter with a summary in Sect. 4.8.

4.1 Overview of Predictable SDRAM Controller

We learned from our discussion about memory controllers in Sect. 3.4 that there are many important design decisions when making a new design, and the right choices are determined by the goals of the design. We are developing a predictable SDRAM controller, and we mentioned in Sect. 2.1.1 that our approach to predictability is based on combining memories and arbitration that are predictable in themselves. The motivation for this decision is that it allows us to combine different memories

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_4, © Springer Science+Business Media, LLC 2011

and arbiters in a transparent manner, thus abstracting from the diversity of memory controllers found in contemporary SoCs. To consider the memory predictable, we require useful bounds on both net bandwidth and the time to serve a scheduled memory request. The predictable arbitration accounts for resource sharing, and here we require a useful bound on the number of interfering memory transactions. Together, these requirements allow us to determine the behavior of a shared memory. In the light of these requirements, we proceed by looking into the design choices made for each of the three major functional blocks in an SDRAM controller.

4.1.1 Arbitration

Our first design decision involves choosing between static and dynamic (front-end) arbitration. We highlighted predictability as a feature of static arbiters in Sect. 3.4.1. However, we also mentioned that these arbiters cannot satisfy the requirements of latency-sensitive requestors, or requestors with low bandwidth requirements without wasting bandwidth. We explained in Sect. 1.1.6 that we consider requestors with these requirements in this work, and that SDRAM bandwidth is a scarce resource that cannot be wasted. We hence decide to use dynamic arbitration for our predictable memory controller design. However, we also mentioned that all dynamic arbiters are not predictable, which requires us to further reduce the design space. For the reasons explained in Sect. 2.2, we choose to limit ourselves to dynamic arbiters in the class of Latency-Rate ($\mathcal{LR}$) servers, which is a subset of predictable dynamic arbiters. In combination with a predictable memory, these arbiters guarantee a requestor a minimum bandwidth, b', after a maximum service latency, Θ^{cc}, thus providing a lower bound on service in an interval of arbitrary length. The class contains many well-known arbiters, such as Weighted Round-Robin [66], Deficit Round-Robin [111], Time-Division Multiplexing (TDM) [90], and several varieties of Fair Queuing [140], suitable for a wide range of requestor requirements. We discuss this further when considering resource arbitration in Chap. 5.

4.1.2 Command Generator

After deciding to use dynamic arbiters in the class of $\mathcal{LR}$ servers to satisfy our requirement on predictable arbitration, we continue by making the memory behave in a predictable manner. We start this process by considering the options of static and dynamic command generation and scheduling. We require useful bounds on the amount of net bandwidth and the time to serve a scheduled memory request. This requirement fits well with the properties of a static command generator. However, many applications in our considered application domains are too dynamic and input dependent to fit with a static schedule. On the other hand, dynamic command generators typically prove too complicated to analyze. We hence decide to take a

middle ground and develop a hybrid approach that combines aspects of static and dynamic command generation and scheduling. We use predictable memory patterns, which are precomputed sequences of commands for read accesses, write accesses, read/write switches, and refresh operations, respectively. These short patterns are then dynamically scheduled by the command generator depending on whether the scheduled request is a read or a write, or if it is time to refresh. We hence reduce the problem of scheduling memory commands to the problem of scheduling memory patterns, which is an easier problem, since patterns have much fewer constraints determining when they can be scheduled. The hybrid approach still has the benefit of being predictable, since the rules for how the patterns can be dynamically combined are relatively straight-forward. The command generator can be kept simple, since the patterns are constructed at design time to prevent timing violations for all valid dynamic combinations. We support an increased level of dynamism compared to fully static approaches, since the decision to schedule a read, a write or a refresh is taken at run time. Lastly, the problem of computing and storing schedules is reduced, since we only have to compute and store patterns, which are a small number of short sub-schedules.

4.1.3 Memory Map

Our choice to use a hybrid approach for the command generator means that bandwidth and latency can be bounded at design time. The next step is to choose a suitable memory map to make the bounds as useful as possible. The properties of the interleaved memory map provides a closer fit with our requirements, since the bound on gross bandwidth is always greater than or equal to that of a continuous memory map. However, it dissipates more power than a continuous memory map, since activates and precharges contribute significantly to the power consumed by the memory device. We accept this drawback with the motivation that we are providing the first predictable memory controller of its kind, and even though reducing power consumption is an important goal in embedded systems, it is not one of the main goals addressed in this book. However, we recognize this as important future work to enable predictable memory controllers in portable devices.

4.2 Memory Patterns

After motivating the design choices for our predictable memory controller, we explain the details of the SDRAM back-end. We start by discussing the predictable memory patterns with interleaving memory accesses. The command generator uses a set of predictable memory patterns, consisting of five patterns sorted into two groups. The read and the write pattern constitute the first group called *access patterns*. The name of the group reflects that these patterns are the only ones

that access the row buffers and modify the contents of the memory. The second group, called *auxiliary patterns*, contains a read/write switching pattern, a write/read switching pattern, and a refresh pattern. These patterns do not access the contents of the memory, but are required to give the data bus time to switch direction, and to prevent the memory from losing data.

4.2.1 Scheduling Rules

The scheduling rules determine how the memory patterns may be dynamically combined by the command generator. These important rules impose requirements on the construction of the patterns, and affect the worst-case latency and gross/net bandwidth. Our approach uses five scheduling rules: (1) Memory patterns are scheduled in a non-preemptive manner, which means that a pattern that has been issued cannot be stopped until it has finished. This rule restricts scheduling to *work on the granularity of patterns*, as opposed to SDRAM commands, greatly simplifying both scheduling and analysis. (2) A read or a write pattern can be scheduled immediately after itself, or when the memory is idle. This rule makes *successive read and write transactions independent of each other*, further simplifying analysis. (3) A write pattern following a read pattern must be preceded by a read/write switching pattern. Similarly, a read pattern following a write pattern must be preceded by a write/read switching pattern. Combined with the first and second rules, this *bounds the interference that can carry over from one memory transaction to another*. It is possible to build enough time into the read and write patterns to allow them to repeat after each other arbitrarily [120]. However, this is equivalent to enforcing a read/write switch after every access pattern, which may unnecessarily increase average latency and waste bandwidth. (4) A read or a write pattern can be scheduled immediately after a refresh pattern. This follows from that the refresh command leaves all banks in a precharged state, suitable for both read and write patterns. (5) A refresh pattern will only be scheduled after a read pattern, a write pattern, another refresh pattern, or if the memory is idle. Technically, it would be possible to schedule a refresh also after a switching pattern, but it does not make sense to spend time switching direction and then schedule a refresh pattern, which can be followed by either a read or a write pattern regardless.

4.2.2 Pattern Descriptions

We now present the structure of the different patterns in a pattern set. There are many possible patterns for each memory device that implement this structure. For now, we keep the discussion general and consider any patterns of the different types that satisfy the scheduling rules and do not violate the timing constraints of the memory device. We refer to these patterns as *valid patterns*. We return in Sect. 4.5 to discuss how to construct valid patterns that are efficient.

We have chosen to use an interleaving memory map for our design. This means that read and write accesses to successive logical addresses map to the different banks in sequence with a granularity of one or more SDRAM bursts. The second scheduling rule in Sect. 4.2.1 states that successive access patterns of the same type must be completely independent of each other. It is hence not possible to assume that the correct rows are open in any of the banks, so an access pattern must contain least one activate command and precharge command for each bank. The pattern also contains a fixed number of SDRAM bursts to every bank. The number of SDRAM bursts to each bank is a pattern parameter and is referred to as the *burst count*, defined in Definition 4.1. The example access patterns in Fig. 2.3 have a burst count of one, since there is only a single SDRAM burst per bank in the patterns.

The reason for having a burst count larger than one is that a single burst requires $BL/2$ clock cycles to complete, which may be less than the minimum time between activates to different banks ($tRRD$). Similarly, a single burst to each bank completes in $BL/2 \cdot n_{banks}$ clock cycles, which may be less than the minimum activate-to-activate delay for the same bank (tRC). In both cases, NOP commands must be inserted to satisfy the timing constraints of the memory. Increasing burst count addresses this problem by increasing the number of cycles that data is transferred during a pattern. However, increasing the burst count also increases the access granularity of the memory, defined in Definition 4.2. More data hence has to be read or written on every access and requests smaller than the access granularity of the pattern are masked or padded to fit. This choice might sound severe, but it is important to realize that no SDRAM controller performs well in the worst case with small memory transactions, due the inherent uncertainty of SDRAM memories. Different approaches push this uncertainty in different directions to put it where it does not interfere with the goals of the design. Our goal is to make a predictable SDRAM controller, so we push this uncertainty into data efficiency where it can be quantified at design time, allowing us to bound the net bandwidth. The impact of this decision becomes apparent when bounding memory efficiency in Sect. 4.3 and when computing net bandwidth for some example memories in Sect. 4.7.

Definition 4.1 (Burst count). The burst count of an access pattern is denoted by BC, and is defined as the number of SDRAM bursts per bank in the pattern.

Definition 4.2 (Access granularity). The access granularity in bytes of an access pattern is denoted by g, and is defined as $g = BC \cdot BL \cdot n_{banks} \cdot w_{mem}$.

The switching patterns are used to provide sufficient time for the SDRAM to reverse the direction of the data bus. These patterns only consist of NOP commands, and the length is determined by the minimum number of cycles required between read and write commands, which are defined by the specification of the memory device. Note that it is possible to have switching patterns with a length of zero cycles if the distance between the last read of a read pattern and the first write of a write pattern, or the other way around, is already sufficient.

The refresh pattern contains a single refresh command, preceded and succeeded by a number of NOPs. There have to be enough NOPs before the refresh command

to allow all banks to (auto-)precharge after the last read or write pattern. After the refresh command is issued, there have to be at least $tRFC$ NOPs to allow the refresh operation to complete before the next pattern is issued.

We conclude our discussion about the general structure of memory patterns by formally defining memory patterns and pattern sets in Definitions 4.3 and 4.4, respectively. Definition 4.4 considers the lengths of the patterns in the set, corresponding to the number of commands in the pattern. One command is issued by the memory controller per clock cycle, which implies that the time to issue a pattern is known at design time. This information is required to bound the bandwidth and latencies provided by the memory controller.

Definition 4.3 (Memory pattern). A memory pattern is defined as a sequence of SDRAM commands in {ACT, RD, WR, PRE, REF, NOP}. The number of commands in the pattern is referred to as the length of the pattern.

Definition 4.4 (Pattern set). A pattern set is defined as (t_{read}, t_{write}, t_{rtw}, t_{wtr}, t_{ref}), where the parameters correspond to the lengths of the read pattern, the write pattern, the read/write switching pattern, the write/read switching pattern, and the refresh pattern, respectively.

4.2.3 Pattern Set Dominance

Before bounding the gross/net bandwidth or latency for a given pattern set, we must determine which sequence of patterns produces the worst results. There are four different possibilities, depending on the relations between the lengths of the patterns in the set. We hence sort pattern sets into four classes: read-dominant, write-dominant, mix-read-dominant, or mix-write-dominant. We proceed by defining these classes and show how to compute the dominance of a pattern set.

A pattern set is classified as *read-dominant* when the read pattern is longer than the write pattern and both the switching patterns put together. This is formally defined in Definition 4.5 and illustrated in Fig. 4.1a. In this case, the lowest bandwidth and longest latency occurs when all interfering transactions are reads, i.e. when only read patterns and an occasional refresh pattern are issued. Conversely, a pattern is considered *write-dominant* if the write pattern is longer than the combined lengths of the read pattern and both the switching patterns. This case is defined in Definition 4.6, and an example is shown in Fig. 4.1b. It follows by the earlier reasoning that the worst-case bandwidth and latency for a write-dominant pattern set occurs when all interfering requests are writes, resulting in that only write patterns and refresh patterns are issued. Pattern sets that are neither read-dominant nor write-dominant are referred to as *mix dominant* sets, defined in Definition 4.7. For these sets, the worst-case bandwidth and latency is provided when interfering requests alternate between reads and writes, causing as many switches as possible. The definitions of the dominance classes are all expressed in terms of the read pattern to clearly show that the classes are mutually exclusive and jointly exhaustive.

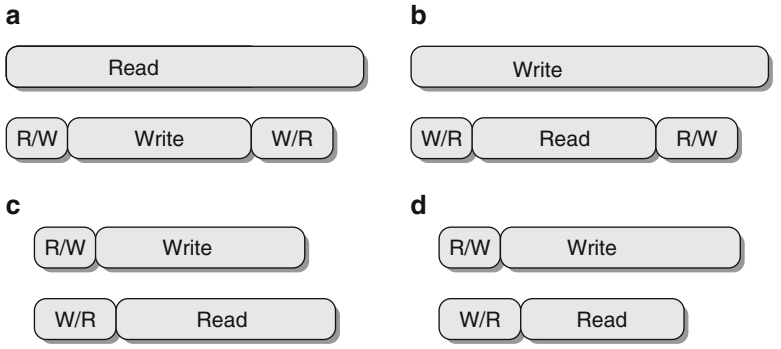

Fig. 4.1 Example pattern sets illustrating the four different dominance classes. (**a**) A read-dominant pattern set. (**b**) A write-dominant pattern set. (**c**) A mix-read-dominant pattern set. (**d**) A mix-write-dominant pattern set

Definition 4.5 (Read-dominant pattern set). A pattern set is defined as read-dominant iff $t_{read} > t_{write} + t_{wtr} + t_{rtw}$.

Definition 4.6 (Write-dominant pattern set). A pattern set is defined as write-dominant iff $t_{write} > t_{read} + t_{wtr} + t_{rtw}$, which is equivalent to $t_{read} < t_{write} - t_{wtr} - t_{rtw}$.

Definition 4.7 (Mix-dominant pattern set). A pattern set is defined as mix-dominant iff $t_{write} - t_{wtr} - t_{rtw} \leq t_{read} \leq t_{write} + t_{wtr} + t_{rtw}$.

The division into three dominance classes is sufficient to bound net bandwidth. However, to accurately determine worst-case latency, mix-dominant pattern sets are further subdivided into two categories: *mix-read-dominant* and *mix-write-dominant* sets. The reason is that we need to know if an odd number of interfering requests result in more read patterns or write patterns in the worst case. A mix-read-dominant pattern set corresponds to a mix-dominant set in which the lengths of the write to read switching pattern and the read pattern is greater than or equal to that of the read to write switching pattern and the write pattern. Otherwise, the pattern set is mix-write-dominant. Mix-read-dominant and mix-write-dominant pattern sets are formally defined in Definitions 4.8 and 4.9, respectively. The corresponding example pattern sets are illustrated in Fig. 4.1c, d.

Definition 4.8 (Mix-read-dominant pattern set). A mix-dominant-pattern set is defined as mix-read-dominant iff $t_{wtr} + t_{read} \geq t_{rtw} + t_{write}$, which is equivalent to $t_{read} \geq t_{write} - t_{wtr} + t_{rtw}$.

Definition 4.9 (Mix-write-dominant pattern set). A mix-dominant pattern set is defined as mix-write-dominant iff $t_{rtw} + t_{write} > t_{wtr} + t_{read}$, which is equivalent to $t_{read} < t_{write} - t_{wtr} + t_{rtw}$.

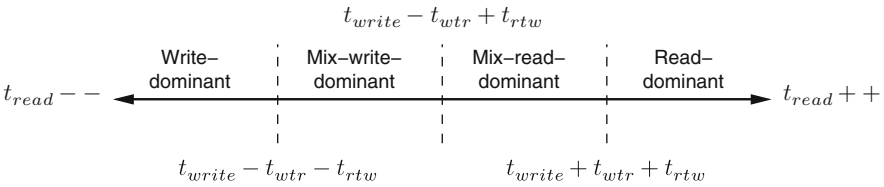

Fig. 4.2 Illustration of how the dominance class of a pattern set changes as t_{read} is incremented or decremented

Having defined all four dominance classes, we illustrate their relation in Fig. 4.2. The figure shows how the dominance class of a pattern changes as t_{read} is scaled up and down while keeping t_{write}, t_{wtr}, and t_{rtw} fixed.

4.3 Memory Efficiency Bound

We have now introduced the concept of predictable memory access patterns and learned how to categorize pattern sets into different dominance classes, based on the situation that triggers the worst-case bandwidth and latency. We now have all the necessary ingredients to lower bound the memory efficiency for all classes of pattern sets, which is an important step towards creating a predictable memory controller. We proceed by walking through each of the efficiency categories presented in Sect. 3.3 and show how predictable memory patterns allow them to be bounded. We illustrate the effects of cumulatively bounding the efficiencies using a running example. Figure 4.3 shows the starting point of this example, which is a sequence of patterns and their associated SDRAM bursts. The gray bursts are the useful bursts that are transferred to and from the requestors. The black bursts correspond to data that is not explicitly requested by the requestors, but is provided anyway due to the minimum access granularity of the patterns.

4.3.1 Refresh Efficiency

We explained in Sect. 3.3.1 that the refresh efficiency depends on the time to precharge all banks and execute a refresh command, and the refresh period. We mentioned that the difficulty with accurately bounding refresh efficiency is to know how long it takes to precharge all banks, since this depends on the state of the memory. This problem is solved in our approach, since we know that the refresh pattern is preceded either a read pattern or a write pattern, and the state of the memory at the end of these patterns is known at design time. This enables

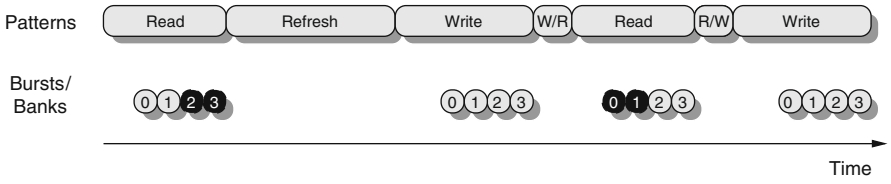

Fig. 4.3 A sequence of patterns and corresponding bursts

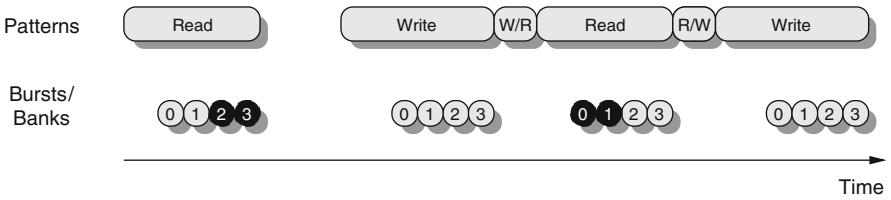

Fig. 4.4 Refresh efficiency accounts for refresh patterns

us to account for the time to precharge all banks by inserting NOPs before the refresh command in the refresh pattern. The length of the refresh pattern, t_{ref}, hence accounts for all time lost due to refresh operations.

The refresh period is controlled by a timer that triggers every $tREFI$ clock cycles (corresponding to 7.8 µs for all DDR2 and DDR3 memories at normal operating temperatures). At this point, the memory controller prepares to schedule a refresh pattern. However, the scheduling rules state that a refresh pattern can only be issued after a read or write pattern has finished. The longest blocking time, t_{block}, before a refresh pattern can be issued is hence determined by the largest sum of a write/read switching pattern and a read pattern, and a read/write switching pattern and a write pattern. This is expressed in (4.1), which is independent of the dominance class of the pattern set. A refresh pattern is hence scheduled every 7.8 µs on average, but with some occasional jitter due to blocking. This jitter does not jeopardize the integrity of the data in the memory array unless it is greater than $8 \cdot tREFI$ [61, 62], which is a very long time in comparison to the time it takes to execute any reasonable pattern. In case the jitter is larger than $tREFI$, multiple refresh events are queued, resulting in that several refresh patterns are executed in sequence. We now bound refresh efficiency according to (4.2). Figure 4.4 illustrates the meaning of bounding refresh efficiency by removing the refresh pattern from the example sequence of patterns.

$$t_{block} = \max(t_{wtr} + t_{read}, t_{rtw} + t_{write}) \tag{4.1}$$

$$e^{ref} = 1 - \frac{t_{ref}}{tREFI} \tag{4.2}$$

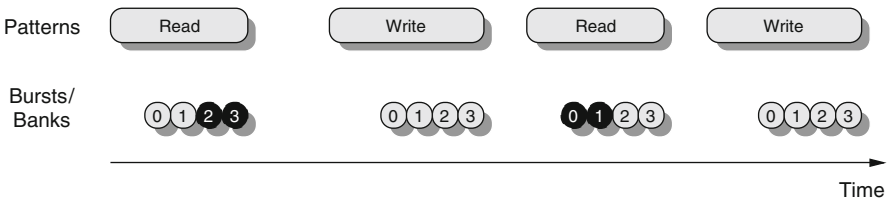

Fig. 4.5 Read/write efficiency accounts for switching patterns

4.3.2 Read/Write Efficiency

The read/write efficiency accounts for the time lost to switching direction of the data bus. Read/write efficiency is often difficult to determine, since the worst-case number of switches in an interval is rarely known at design time. Using predictable memory patterns, we know that the read/write efficiency corresponds to the maximum fraction of time spent executing read/write and write/read switching patterns. This can be determined at design time, since the length of the patterns and the scheduling rules are known. The read/write efficiency is straight-forward to determine for read-dominant and write-dominant pattern sets, since these issue only read, write, and refresh patterns in the worst case. Since the worst case does not contain any switches, it follows that the read/write efficiency is 100% for these sets. However, if the set is mix-dominant, there is a switch after every read and write pattern in the worst case. The read/write efficiency is hence determined by the time required to execute a read and write pattern divided by the time to execute the patterns and their corresponding switches, as shown in (4.3). Bounding read/write efficiency after having already bounded refresh efficiency is illustrated in Fig. 4.5 by also removing all switching patterns from the sequence. All that remains to be considered is the efficiency of the read and write patterns themselves.

$$e^{rw} = \begin{cases} 1 & \text{if read-dominant or write-dominant} \\ \frac{t_{read}+t_{write}}{t_{read}+t_{write}+t_{wtr}+t_{rtw}} & \text{if mix-dominant} \end{cases} \tag{4.3}$$

4.3.3 Bank and Command Efficiency

The bank efficiency accounts for the overhead associated with activating and precharging banks. This term is highly dependent on how the traffic maps to the different rows and banks, as explained in Sect. 3.3.3, and cannot be tightly bounded in the general case. The predictable memory patterns allow us to tightly bound this efficiency by interleaving every memory access over all banks, making the

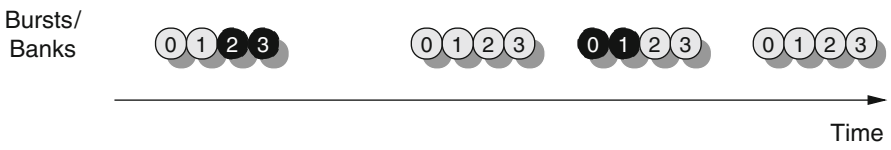

Fig. 4.6 Bank and conflict efficiencies remove overhead within read and write patterns, leaving only data bursts

timings of all activates and precharges known at design time. We compute the bank efficiency by determining the fraction of cycles of a read and a write pattern that data is actually transferred. However, this also accounts for any overhead due to command conflicts that may delay activate or precharge commands and result in a longer read or write pattern. Although it may be possible to distinguish this loss, we conveniently choose to compute bank efficiency and command efficiency as an aggregate. The aggregate bank and command efficiency is computed by first determining the number of cycles that data is transferred during a read pattern or a write pattern, denoted by $t_{transfer}$. This is calculated by considering that there are BC bursts of BL words to each of the n_{banks}, and that two words are transferred every clock cycle to a DDR memory. This is expressed in (4.4). For read-dominant pattern sets, we simply divide the data transfer cycles with the length of the read pattern. Conversely for write-dominant sets, we divide the transfer cycles with the length of the write pattern. Lastly for mix-dominant sets, we consider the fraction of transfer cycles during one read and one write pattern. This is expressed formally in (4.5). Accounting for bank and command efficiency after already considering refresh efficiency and switching efficiency is illustrated in Fig. 4.6. All overhead cycles inside the read and write patterns are removed, leaving only the cycles where data is transferred.

$$t_{transfer} = \frac{BC \cdot BL \cdot n_{banks}}{2} \tag{4.4}$$

$$e^{bank} \cdot e^{cmd} = \begin{cases} \frac{t_{transfer}}{t_{read}} & \text{if read-dominant} \\ \frac{t_{transfer}}{t_{write}} & \text{if write-dominant} \\ \frac{2 \cdot t_{transfer}}{t_{read} + t_{write}} & \text{if mix-dominant} \end{cases} \tag{4.5}$$

4.3.4 Data Efficiency

Data efficiency corresponds to the amount of data that is transferred over the data bus that is useful to the requestors. The data efficiency of a requestor is determined by how the size and alignment of its requests fit with the minimum access granularity of

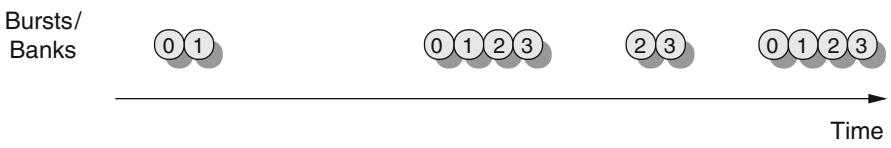

Fig. 4.7 Data efficiency accounts for data that is not useful to requestors, leaving only requested data bursts

the memory, as previously discussed in Sect. 3.3.5. The minimum access granularity in our approach is equal to the granularity of an access pattern, computed according to Definition 4.2. This is a drawback of our approach, since the access granularity of a pattern is significantly larger than that of a single SDRAM burst, which is the minimum access granularity of the memory device itself. This means that some of the efficiency gains provided in the other categories are lost in data efficiency in the presence of small requests. However, a benefit of our approach is that this loss can be quantified, allowing the net bandwidth to be bounded. The data efficiency of a requestor r is computed according to (4.6). As seen in the equation, we use a simple model that uses the worst-case combination of size and alignment that is possible for the requests from a requestor. A more refined model of data efficiency may be possible given a good characterization of the application. We demonstrate (4.6) by applying it to the example in Fig. 3.3, assuming a 16-bit memory interface. The requestor in the figure has a request size of 16 bytes (8 words), an alignment of 6 bytes (3 words), and the minimum access granularity of the memory device is 16 bytes. This results in a data efficiency of 50%, which is accurate, since two accesses of 8 words are required to transfer the request.

$$e_r^{data} = \min_{\forall \omega_r^k \in \Omega_r} \left(\frac{s^{bytes}(\omega_r^k)}{\left\lceil \frac{s^{bytes}(\omega_r^k) + a(\omega_r^k)}{g} \right\rceil \cdot g} \right) \tag{4.6}$$

Equation (4.6) can be used to determine the total data efficiency of the memory in the special case where the request sizes and alignments of all requestors are the same. If this assumption does not hold, then the data efficiency depends on how frequently the different requestors are scheduled, which is determined by the particular front-end arbiter. We return to discuss this issue in Chap. 7. Figure 4.7 illustrates the effect of accounting for data efficiency after all other categories have been considered. All bursts that are not useful to the requestors are removed, leaving only the actual useful bursts in the figure. We have now arrived at the net bandwidth, which concludes our example of bounding memory efficiency.

4.4 Latency Bound

We have shown how to bound gross and net bandwidth, based on how the patterns in a pattern set are dynamically combined in the worst case. We now proceed by showing how to derive the maximum latency of a request, given a maximum number of interfering atomic service units (atoms). An atomic service unit corresponds to a memory transaction with a size equal or less than the access granularity of the access patterns, and it is hence served in a non-preemptive manner. We choose this particular metric, since this is the granularity at which arbitration is done in our architecture, as previously explained in Sect. 2.1.3. We define the maximum latency of an atom as the total length of interfering patterns. This accounts for all switching patterns and access patterns related to different atoms, and to refresh patterns. The own switching pattern and access pattern is not considered a part of this latency.

Our first step towards bounding the worst-case latency of an atom is to disregard of refresh patterns and compute the maximum latency caused by read, write and switching patterns in the presence of x interfering atoms. The maximum latency in this case depends on the dominance of the pattern set, as shown in (4.7). If the set is read-dominant, then all interfering atoms are assumed to be reads. In this case, the worst case contains an initial write/read switch, followed by x read patterns. By the same logic, all interfering atoms are assumed to be writes for write-dominant patterns. The worst-case latency for mix-dominant patterns happens if the interfering atoms alternate between reads and writes, resulting in the maximum number of interfering switching patterns. Which type of access pattern there are more of depends on whether the pattern set is mix-read-dominant or mix-write-dominant, as shown in (4.7). For clarity, Table 4.1 shows the mix of patterns instantiated in the worst case for up to four interfering atoms using mix-dominant patterns.

Table 4.1 Worst-case patterns for mix-dominant pattern sets

x	t_{wtr}	t_{read}	t_{rtw}	t_{write}
Mix-read-dominant pattern sets				
0	0	0	0	0
1	1	1	0	0
2	1	1	1	1
3	2	2	1	1
4	2	2	2	2
Mix-write-dominant pattern sets				
0	0	0	0	0
1	0	0	1	1
2	1	1	1	1
3	1	1	2	2
4	2	2	2	2

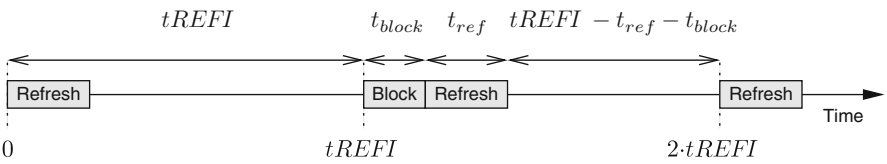

Fig. 4.8 The minimum distance between two refresh patterns

$$t_{aux}(x) = \begin{cases} t_{wtr} + t_{read} \cdot x & \text{if read-dominant} \\ t_{rtw} + t_{write} \cdot x & \text{if write-dominant} \\ \left\lceil \frac{x}{2} \right\rceil \cdot (t_{wtr} + t_{read}) + \left\lfloor \frac{x}{2} \right\rfloor \cdot (t_{rtw} + t_{write}) & \text{if mix-read-dominant} \\ \left\lceil \frac{x}{2} \right\rceil \cdot (t_{rtw} + t_{write}) + \left\lfloor \frac{x}{2} \right\rfloor \cdot (t_{wtr} + t_{read}) & \text{if mix-write-dominant} \end{cases} \tag{4.7}$$

Next, we account for interference due to blocking and refresh, and compute the total worst-case latency, t_{tot}. Blocking occurs because the worst-case latency of a request may start counting from a moment just after a scheduling decision has been taken by the arbiter. This results in that maximally one additional atom may interfere with the requestor due to the non-preemptive nature of memory patterns. We account for this by adding one extra interfering atom to the bound, thus using $t_{aux}(x+1)$ to compute the maximum interference from x atoms. To compute the maximum interference from refresh patterns, we must consider the minimum distance between two of these patterns. This distance occurs if one refresh pattern is maximally blocked (t_{block}) by other patterns, and the following refresh pattern encounters no blocking. In this case, the time between two consecutive refresh patterns is $tREFI - t_{ref} - t_{block}$, as illustrated in Fig. 4.8. For every such interval, we add the time to execute a refresh pattern to the latency from other interfering patterns, as shown in (4.8). This approach is somewhat pessimistic, since two such worst-case intervals cannot occur multiple times in a row. However, we do not attempt to tighten the bound in this work. The equation rounds the number of interfering refresh patterns up, reflecting that a refresh can happen immediately in an arbitrary interval. Hence, all requestors always have at least one refresh pattern in their worst-case latency.

Equation (4.8) provides a hard bound on the interference from other requestors accessing the back-end. A key feature of this equation is that it does not make *any assumptions* about the arbiter, since the number of interfering atoms is left as a parameter. This *separates the analysis of the arbiter and the resource*, as described in Sect. 2.1. The back-end can hence be used in a predictable manner with a variety of arbiters, which is a differentiating feature with respect to the state of the art. Another important feature is that the equation computes the interference from a *sequence of requests*, resulting in less pessimistic latency than when the worst-case

interference from a single request is multiplied with the number of interfering requests. As an example, (4.8) captures that two following memory requests cannot both be interrupted by a refresh and that a two longer write-to-read switches must be separated by a shorter read-to-write switch. The equation can furthermore be used to bound many different definitions of latency. One example is worst-case delay, often used in communication networks [118], which considers the maximum time from a request arrives at a resource until it gets scheduled by the arbiter. Another example is service latency, used by $\mathcal{LR}$ servers, which measures the time from a request is eligible for scheduling at the head of the Request Buffer until it gets scheduled. All that is required to use (4.8) with any of these metrics is that the chosen arbiter can provide a bound on the number of atoms scheduled in this time. This holds for any arbiter in the class of $\mathcal{LR}$ servers.

$$t_{tot}(x) = \left\lceil \frac{t_{aux}(x+1)}{tREFI - t_{ref} - t_{block}} \right\rceil \cdot t_{ref} + t_{aux}(x+1) \tag{4.8}$$

4.5 Memory Pattern Generation

We have now explained how our approach to achieve predictability with SDRAMs allows us to bound gross/net bandwidth and worst-case latency by using memory patterns. However, we have not yet discussed how these patterns are generated. In the early stages of this research [5, 7, 120], memory patterns were derived manually using spreadsheets. Although this was sufficient to illustrate the concept, there are three important reasons to automate the process: (1) Making a pattern set is a time consuming process that must be repeated every time a new combination of memory device, burst length (BL) and burst count (BC) is needed. (2) Making patterns manually is error prone, considering the large number of constraints that must be satisfied for a pattern to be valid. In fact, our automated generators found errors in some of our handmade patterns. (3) It is difficult to ensure that the generated patterns provide (close to) optimal efficiency, considering the huge number of valid patterns for a particular memory device, burst length and burst count. Also here, our automated pattern generators have derived patterns that are more efficient than some of those previously made by hand.

We now proceed by discussing how to automatically compute efficient pattern sets. First, we motivate some design decisions that focus the vast search space on the more efficient sets, while speeding up the computation. We then proceed by explaining the conditions that have to be satisfied for an access pattern to be considered valid and complete. After discussing these preliminaries, we move on to present three pattern generation algorithms, each presenting a different trade-off between the efficiency of the generated pattern sets and run time. Note that we focus our efforts on generating patterns for a given burst count and memory device. We return to discuss how to determine which burst count is best suited to satisfy requestor requirements in Chap. 7.

4.5.1 Design Decisions

The number of possible access patterns for a given burst count and memory device grows exponentially with the pattern length, resulting in a huge design space. To limit the size of the design space, we make five important design decisions: (1) We assume that shorter access patterns provide more bandwidth and lower latencies than longer ones. (2) We do not distinguish the identity of the banks, but cycle through them in ascending order. (3) We always start an access pattern with an activate command. (4) Instead of scheduling precharge commands, the last burst to each bank in an access pattern is issued with the auto-precharge flag. (5) We issue all bursts in an access pattern to one bank before moving on to the next. We proceed by motivating these decisions and explaining their consequences.

The first design decision is that we assume that shorter access patterns result in higher bandwidth and lower latencies. The benefit of this assumption is that it allows the pattern generation algorithms to focus on independently finding the shortest read and write patterns for the given burst count before deriving the corresponding auxiliary patterns. Otherwise, auxiliary patterns have to be derived for every possible pair of access patterns, exploding the design space. The validity of the assumption depends on the dominance class of the pattern set. The assumption typically holds for pattern sets that are read or write-dominant. For these patterns, the lowest bandwidth and longest latencies occur when all transactions are reads or writes, respectively, and hence when there are no read/write switches. The number of words transferred in an access pattern with a given burst count is constant. A shorter pattern hence transfers the same amount of data in less time, which intuitively means increased bandwidth. Expressed more formally, the bank and command efficiency in (4.5) monotonically increases with reducing pattern lengths, because $t_{transfer}$ is constant, while t_{read} and t_{write} are reducing. The problem with the assumption is that a shorter access pattern may result in slightly longer switching patterns and refresh patterns, since NOP commands at the end of access patterns help precharging the banks before auxiliary patterns are issued. This effect, both in terms of bandwidth and latency, is negligible in most cases for refresh patterns, due to their low frequency. Experiments with a variety of memories and burst counts suggest that memory efficiency may reduce with 0.1% and that latencies are unaffected by longer refresh patterns. The effect of longer switching patterns is negligible for read-dominant and write-dominant pattern sets, but may be more significant for their mix-dominant counter parts, since read/write switches occur after every access pattern in the worst case. However, the maximum impact of this is estimated to be less than 1% reduction in memory efficiency and a few clock cycles of latency under any circumstances.

The second design decision is not to distinguish the identity of the banks. This means that we do not consider two access patterns as different if all commands to two banks are swapped. Swapping the commands in this fashion essentially corresponds to consequently changing the identity of the banks, which affects neither bandwidth nor latency. However, this decision has a significant impact on the set of valid patterns, since we do not have to consider identical patterns that access the banks in different orders.

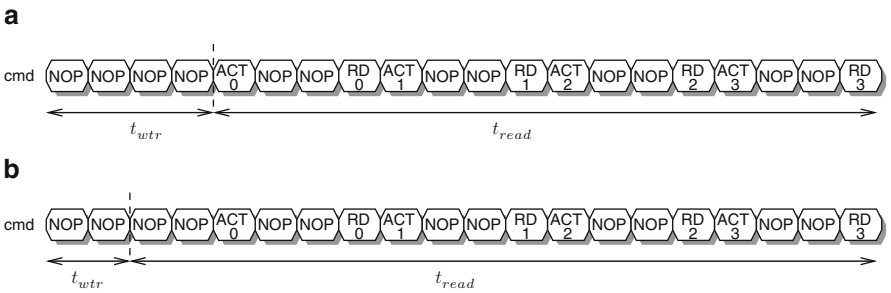

Fig. 4.9 Adding NOPs to the beginning of an access pattern may reduce the length of a switching pattern. (**a**) Original patterns. (**b**) Two NOP commands added to beginning of read pattern

The third design decision states that we always start an access pattern with an activate command. The idea behind this decision is to prune a large number of uninteresting patterns from the design space, which grows exponentially with the length of the pattern. This decision ignores all patterns starting with one or more NOP commands. The rationale behind the decision is that the purpose of an access pattern is to issue a number of read and write bursts to the SDRAM. These bursts cannot be issued until their corresponding banks have been activated. Inserting NOPs in the beginning of an access pattern makes the access pattern longer. This typically reduces bandwidths and increases latencies for read or write-dominant pattern sets, similarly to what we described for the first design decision. For mix-dominant patterns, adding a NOP in the beginning of a read pattern implies that a NOP can be removed from the write/read switching pattern, unless the length of the switching pattern is already zero clock cycles. This is illustrated in Fig. 4.9 for our example memory. Shifting NOP commands from a switching pattern to the beginning of an access pattern does not affect worst-case latency, but it reduces actual-case bandwidth during intervals with better than worst-case switching behavior. However, it does not change the bound on bandwidth for mix-dominant patterns, since the increase in read/write efficiency and decrease in bank and conflict efficiency cancel each other out. This effect can be observed in (4.9), which corresponds to Definition 3.11 with all efficiency terms that depend on the length of the access patterns and switching patterns expanded. Adding n cycles to the length of an access pattern and subtract the same number of cycles from one of the switching patterns does not affect memory efficiency.

$$\begin{aligned} e^{mem} &= e^{ref} \cdot \frac{t_{read} + t_{write}}{t_{read} + t_{write} + t_{wtr} + t_{rtw}} \cdot \frac{2 \cdot t_{transfer}}{t_{read} + t_{write}} \cdot e^{data} \\ &= e^{ref} \cdot \frac{BC \cdot BL \cdot n_{banks}}{t_{read} + t_{write} + t_{wtr} + t_{rtw}} \cdot e^{data} \end{aligned} \tag{4.9}$$

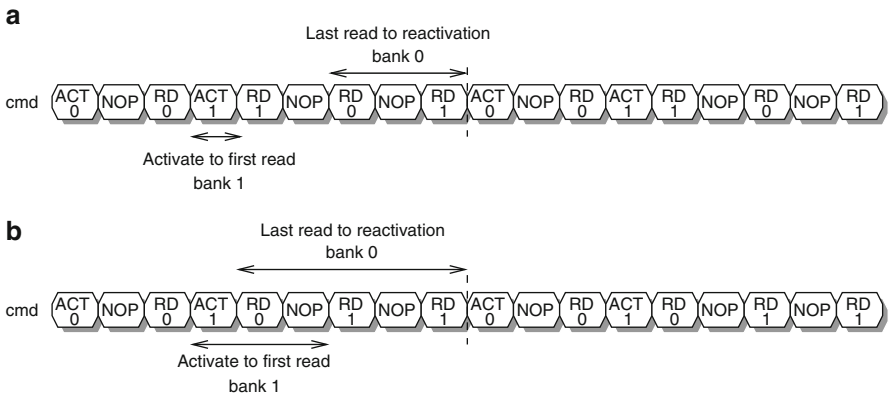

Fig. 4.10 Issuing all bursts to a bank before moving on to the next gives more time between activate and reads/writes, and more time to precharge before reactivating. (**a**) One burst per bank before moving on. (**b**) All bursts to one bank before moving on

The fourth design decision is to issue the last burst to a bank in an access pattern with the auto-precharge flag. This removes the risk of command conflicts when issuing precharge commands, possibly reducing the length of the pattern. It also reduces the number of non-NOP commands in the access patterns, further reducing the design space.

The last design decision is to issue all BC bursts to one bank before proceeding to the next. A bank is ready to receive the next read or write command $BL/2$ cycles after the first. No read or write command can be issued to any other bank before this time, since it would cause a conflict on the data bus. Keeping all bursts to a bank close together may give a bank more time between the activate command and the first read or write command, as well as more time to precharge after the last read or write command before the following activate command. We illustrate this in Fig. 4.10, where a read pattern is repeated after itself. To get a short pattern, we use an *imaginary memory* with very generous timings, two banks, and with $BL = 4$ and $BC = 2$. Figure 4.10a shows the case where only a single read is sent to a bank before moving on to the next. Conversely, the pattern in Fig. 4.10b issues both bursts before moving on to the next bank. We see that the time between the activate to and read to bank 1 is longer in the second figure. Similarly, the time from the last read until the activate command in the second pattern is longer. This makes it easier to satisfy the timing constraints of the memory device, making the set of valid patterns larger and potentially resulting in shorter patterns.

4.5.2 Access Pattern Termination

We now show how to decide when an access pattern is valid and complete, which determines what the access pattern generation algorithms actually have to do. An access pattern is valid and complete when it satisfies the following five criteria: (1) all necessary commands have been scheduled, (2) the activate-to-activate constraint is satisfied for all banks, (3) the four-activate window constraint is satisfied, (4) the data bus constraint is satisfied, and (5) the precharge constraints are satisfied. We proceed by explaining these conditions and how to ensure that they are satisfied.

The first termination condition requires all necessary commands to be included in the pattern. It follows from the structure of access patterns, presented in Sect. 4.2.2, and our design decisions in Sect. 4.5.1 that an access pattern consists of one activate command and BC read or write commands per bank. There are no precharge commands, since the last SDRAM burst in the pattern is issued with the auto-precharge flag. After all commands have been scheduled, NOPs are added to the end of the generated pattern to prevent the following constraints from carrying over into a repeated pattern, violating their independence.

The second condition is that the activate-to-activate constraint must be satisfied for all banks. This condition implies that there must be at least tRC clock cycles between successive activates to a bank when an access pattern is repeated after itself. Since there is only one activate command per bank in an access pattern, this constraint is automatically satisfied if the length of the pattern is greater than or equal to tRC.

The third condition is that any window of $tFAW$ cycles, referred to as a Four-activate window (FAW), can maximally contain four activate commands. This is to ensure that the instantaneous power consumption of a device with eight banks does not exceed that of a device with four banks. This constraint has to be considered during the pattern generation, but NOPs may additionally have to be added at the end of a pattern to allow it to be repeated after itself without violating this constraint.

The fourth termination condition requires that the data produced on the data bus by the last burst in an access pattern does not collide with the data from the first burst in the next. This requirement is satisfied if the corresponding access commands are separated by at least $BL/2$ clock cycles, which is the time required to finish the burst.

The last condition requires that there must be at least tRP clock cycles between the bank is precharged and reactivated. To satisfy this requirement, we must know in which clock cycles the precharges of the banks actually happen. This procedure works differently for read and write patterns. For a read pattern, the precharge cycle of a bank is determined by finding the cycle with its activate command, t_{act}, and the cycle with its last read command, t_{read}^{last}. The precharge cycle is then computed according to (4.10). Note that the precharge cycle is computed with respect to the start of the read pattern and may be greater than the total length of the pattern, indicating that the precharge finishes during the execution of a later pattern. The

procedure is similar for write patterns, although (4.11) is used instead. Both these equations are derived from [61, 62].

$$t_{read}^{pre} = \begin{cases} \max\left(t_{read}^{last} + \frac{BL}{2} + \max(tRTP, 2) - 2, t_{act} + tRAS\right) & \text{DDR2} \\ \max\left(t_{read}^{last} + tRTP, t_{act} + tRAS\right) & \text{DDR3} \end{cases} \tag{4.10}$$

$$t_{write}^{pre} = t_{write}^{last} + tWL + \frac{BL}{2} + tWR \tag{4.11}$$

4.5.3 Branch and Bound

After specifying the task of access pattern generation, we proceed by presenting three algorithms for efficient access pattern generation. The first of the three access pattern generation algorithms is a branch and bound algorithm. This algorithm is based on a depth-first traversal of the set of valid patterns satisfying the design decisions in Sect. 4.5.1. It is guaranteed to find the shortest possible access patterns, as its bounding conditions exclude only longer patterns. We start by giving a brief introduction to the branching part of the algorithm, before explaining how to bound the search space.

The algorithm works by starting an access pattern with an activate command in the first cycle, according to our third design decision. It then looks to see which commands that can be scheduled the following cycle. For each command that respects the timing constraints of the memory, a copy of the pattern is made and each command is appended to the end of a copy. The algorithm repeats this process cycle by cycle until the first pattern is complete. At this point, it stores the completed pattern and returns to one of the remaining copies and continues its search until there are no unfinished copies remaining. An illustration of this algorithm is shown in Fig. 4.11.

The set of valid patterns complying with our design decisions is very large and grows exponentially with the size of the patterns. To speed up execution of the algorithm, we implemented two bounding conditions that limit the size of the design space. The first bounding condition is a sliding cut-off point based on the pattern length. We keep track of the length of the shortest pattern found so far, and stop pursuing any branches longer than this value. This condition significantly reduces the run time and memory use of the algorithm, while trivially not excluding the shortest pattern. The second bounding condition is an extension of the first. Whenever, the algorithm branches, it looks at the list of commands remaining to be scheduled, and performs two quick sanity checks to see if the finished pattern can be shorter than or equal to the current shortest pattern in a best-case scenario. If any of these checks fail, then no further branches along this path is pursued. Just like the first condition, this significantly reduces run time and memory usage of the algorithm. The sanity checks are exact and cannot exclude the shortest pattern from the search space. The first check considers the number of activates that remains to be scheduled, n_{act}. The time required to schedule these commands is guaranteed to be

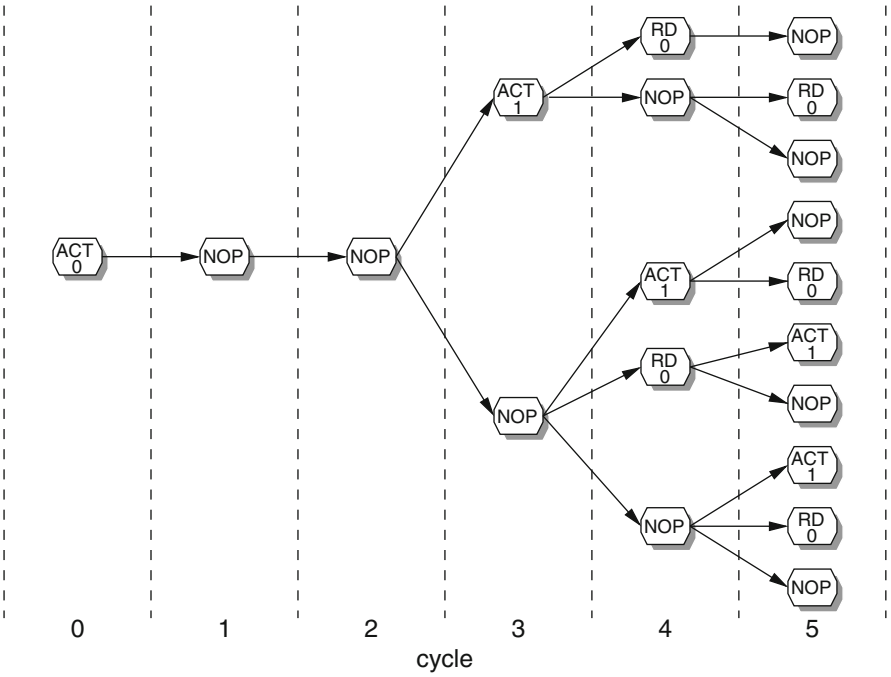

Fig. 4.11 The branch and bound algorithm creates pattern by exploring a tree of SDRAM commands

at least $(n_{act}-1)\cdot tRRD+tRCD$ clock cycles. The $(n_{act}-1)\cdot tRRD$ comes from the fact that activates cannot be scheduled within $tRRD$ cycles of each other, and there are at least $(n_{act}-1)$ full delays between activates remaining. The second part of the sum is $tRCD$. This is included because we know that if there is at least one activate left, there is also at least one read or write command remaining, and $tRCD$ is the minimum delay between an activate and a read or a write command. The first sanity check hence simply determines if the current cycle, t, plus the delays implied by the remaining activates can result in a pattern that is equal to or shorter than the current shortest one, $t_{shortest}$. This is expressed in (4.12)

$$t+(n_{act}-1)\cdot tRRD+tRCD \leq t_{shortest} \tag{4.12}$$

The second sanity check considers the number of remaining read or write commands, n_{acc}. If there are n_{acc} read or write commands remaining, we know that there have to be at least $(n_{acc}-1)\cdot BL/2$ cycles between them to prevent a conflict on the data bus. The data of the last read or write command may overlap with the following pattern and is hence not included. The second check is hence expressed according to (4.13).

$$t+(n_{acc}-1)\cdot BL/2 \leq t_{shortest} \tag{4.13}$$

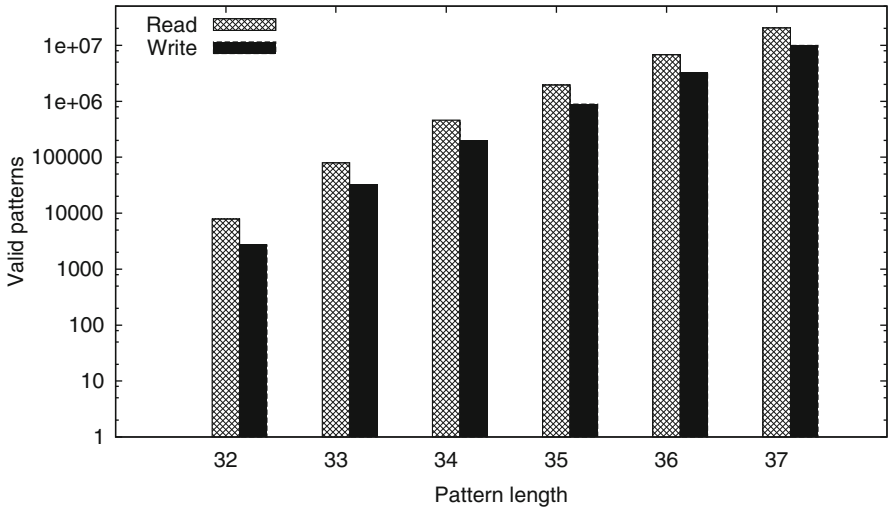

Fig. 4.12 Number of valid patterns fitting our design decisions at $BC = 2$ for a DDR2-400 SDRAM device

After the search is complete, there is at least one access pattern of each type with the shortest length. Out of these, we choose the read and write pattern where the last read or write command is issued as early as possible. This allows the access pattern to hide more of the precharge time, potentially resulting in shorter refresh pattern and switching patterns.

The benefit of the branch and bound algorithm is that it is guaranteed to find the shortest possible access patterns and choose the one of these that provides the shortest auxiliary patterns. The drawback of the algorithm is that it may take a long time to search the design space, despite the help of our two bounding conditions. The complexity of the algorithm begins to show itself as the clock frequency of the memory device increases. This is because the timing constraints become longer, as previously discussed in Sect. 3.3.7, and increase the lengths of the patterns. Similarly, increased burst count increases the number of commands to schedule, creating more options and longer patterns. The size of the design space is seen in Fig. 4.12. The figure shows the number of valid patterns with a particular length that fits with our design decisions for our example DDR2-400 SDRAM memory with $BC = 2$. We note that there are thousands of valid read and write patterns with length 32, which is the minimum possible size. The complexity of the problem becomes apparent when increasing the length of the pattern with five cycles, resulting in that the set of valid patterns grows with three orders of magnitude! For practical purposes, this algorithm is suitable up to DDR3-1600 with $BC = 2$. After this point, the run time of the algorithm moves into months and years. This motivated us to look for a faster algorithm.

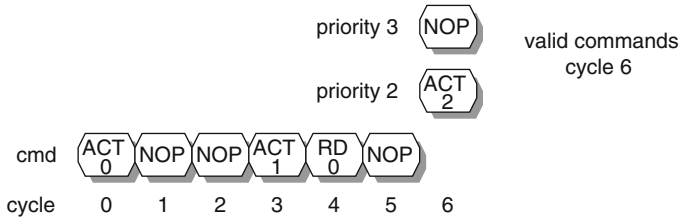

Fig. 4.13 Conceptual illustration of the ASAP scheduling algorithm

4.5.4 As-Soon-As-Possible Scheduling

The second algorithm is a heuristic that attempts to improve run time over the previous algorithm. The idea behind algorithm is to schedule memory commands as-soon-as-possible (ASAP), since this intuitively leads to the shortest access patterns. According to our design decision, the algorithm starts by putting an activate command in the first cycle. It then proceeds one cycle at a time by choosing a command that can be scheduled without violating the timing constraints of the memory. If there are multiple candidate commands, a simple priority scheme is used to make the choice. This contrasts to the previous algorithm that pursues all possible options. This priority scheme first considers read and write commands, since these are the commands that put data on the data bus, thereby increasing efficiency. Activate commands are considered as second, since these enable future read or write commands, and hence future data transfer. However, an activate command is less important than a read or a write, since this can sometimes be postponed without negatively affecting the length of the pattern. If none of these commands are available, a NOP command is scheduled. A conceptual illustration of the ASAP scheduling algorithm is provided in Fig. 4.13, and the pseudo-code is shown in Algorithm 4.1.

Algorithm 4.1 Pseudo-code of the ASAP scheduling algorithm.

```
t ← 0
pattern[t] ← {ACT to bank 0}
while notCompleted(pattern) do
    availableCmds ← getAllowedCmds(pattern, t)
    cmdToSchedule ← pickBestCmd(availableCmds)
    pattern[t] ← cmdToSchedule
    t ← t + 1
end while
```

A consequence of the ASAP scheduling algorithm is that the activate commands are scheduled early in the pattern, as seen in Fig. 4.14a. The reason is that activates

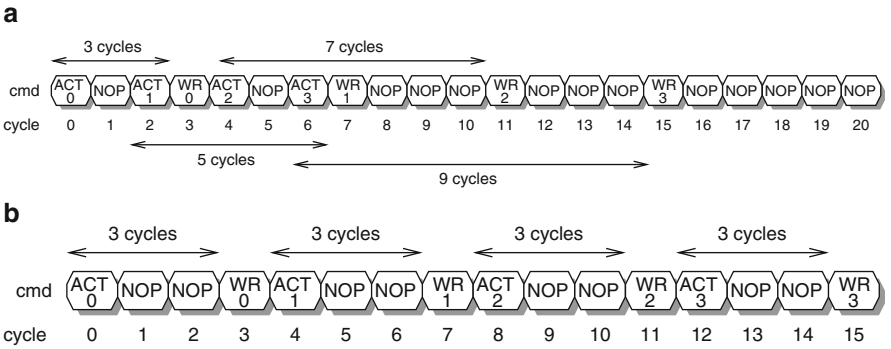

Fig. 4.14 Prematurely scheduled activate commands result in longer access patterns. (**a**) The ASAP algorithm results in increasingly large distances between activate commands and their corresponding write commands. (**b**) A pattern with balanced distances between activate commands and write commands

to different banks can be scheduled every $tRRD$ clock cycles, which is not a very long time. However, the read and write commands must be separated by at least $BL/2$ clock cycles, causing the distance between an activate command and its corresponding read or write to increase, as shown in the figure. This creates a problem, since a bank needs time to precharge after the last read or write command has completed, before it can be reactivated. The earliest reactivation occurs when the pattern is repeated after itself. The critical constraint is hence the time between the last read or write command in the pattern until the activate command in the repeating pattern. The earlier the activate command, the less time available to precharge. This is why the pattern generated by the ASAP scheduling algorithm requires five extra NOP commands to be inserted at the end of the pattern, while the more balanced pattern shown in Fig. 4.14b does not. Clearly, scheduling commands as early as possible is not always beneficial.

The advantage of the ASAP scheduling algorithm is that it runs extremely fast. It generates a schedule in less than a second for any memory and reasonable burst count, clearly addressing the problem with the branch and bound algorithm. However, the advantage in speed comes at the cost of bandwidth, mainly due to the problem with prematurely scheduled activate commands. As a result, the generated patterns provide up to 10% less bandwidth than those generated by the slower algorithm. Although the ASAP scheduling algorithm provides a different trade-off between run time and bandwidth, we consider it rather inefficient, since SDRAM bandwidth is a scarce resource. We hence look into a third algorithm, hoping to find a suitable middle ground.

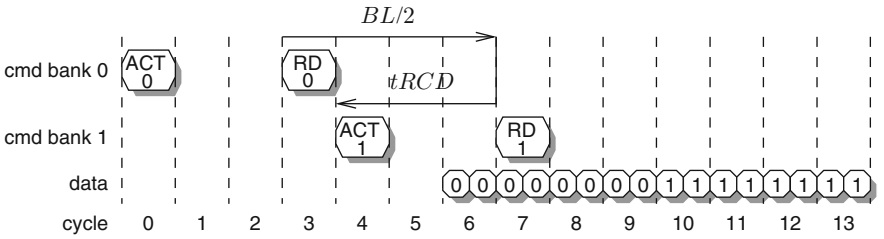

Fig. 4.15 Conceptual illustration of the bank scheduling algorithm for $BC = 1$

4.5.5 Bank Scheduling

The bank scheduling approach is a heuristic that builds on the lessons learned from ASAP scheduling algorithm. The idea behind the algorithm is to keep an activate command as close as possible to its corresponding read or write command, thereby preventing the precharge-to-activate constraint from extending the length of the pattern.

The bank scheduling algorithm works by scheduling one bank at a time. It starts by putting an activate command to the first bank in the first cycle, and a corresponding read or write command at the earliest possible convenience, being $tRCD$ cycles later. Each additional burst to the bank is then scheduled $BL/2$ cycles apart to constantly keep data on the data bus. This finishes the scheduling of the first bank. For each successive bank, the algorithm finds the position of the latest read or write command, and tries to schedule the next read or write $BL/2$ cycles later when the data bus is free. The new read or write command can be scheduled in this position if its activate command can be scheduled $tRCD$ cycles earlier. This depends on whether the cycle already has a scheduled command, and whether the activate-to-activate constraints for different banks and the four-activate window constraint are satisfied. If the activate cannot be scheduled in the requested cycle, the algorithm tries to schedule the read or write command in a later cycle by iteratively repeating this test. Once the first read or write command to the bank has been scheduled, the others follow with a separation of $BL/2$ clock cycles. An illustration of the algorithm is provided in Fig. 4.15 and pseudo-code is presented in Algorithm 4.2. We evaluated an alternative approach to this algorithm, where we let the activate command slide backwards instead of sliding the read or write command forwards. However, the results of this algorithm were at best the same and occasionally provided worse results than the current implementation.

The patterns generated by the bank scheduling algorithms achieve very regular distances between the activates and their corresponding read and write commands, addressing the problem found with the ASAP scheduling approach. In fact, the write pattern shown in Fig. 4.14b was generated using this approach. The run time of the algorithm is similar to the ASAP scheduling algorithm, and hence sufficiently fast. It furthermore generates pattern sets that provide similar bandwidths to those

Algorithm 4.2 Pseudo-code of the bank scheduling algorithm.

```
currentBank ← 0
while currentBank < n_banks do
    // Determine cycle of next activate command
    targetCycleAct ← 0
    if currentBank > 0 then
        cycleLastRead ← getLastRead(pattern)
        targetCycleAct ← cycleLastRead + BL/2 − tRCD
    end if
    while !activateAllowed(targetCycleAct) do
        targetCycleAct ← targetCycleAct + 1
    end while

    // Schedule activate, followed by BC read commands
    pattern[targetCycleAct] ← {ACT to currentBank}
    currentBurst ← 1
    targetCycleRead ← targetCycleAct + tRCD
    while currentBurst ≤ BC do
        pattern[targetCycleRead] ← {RD to currentBank}
        targetCycleRead ← targetCycleRead + BL/2
        currentBurst ← currentBurst + 1
    end while
    currentBank ← currentBank + 1
end while
```

created by the branch and bound algorithm. Bank scheduling hence provides a very favorable trade-off between run time and memory efficiency, compared to the other algorithms.

4.5.6 Computing Auxiliary Patterns

The auxiliary patterns can be computed as soon as the access patterns are calculated by any of the pattern generation algorithms. We start by showing how to generate the refresh pattern, followed by the switching patterns. The refresh pattern starts with a number of NOPs that allow the banks to finish precharging after the latest access pattern. The time required to finishing precharging all banks depends on the distance between the precharge cycle of the last bank, t^{pre}_{read} or t^{pre}_{write}, and the end of the read or write pattern, since this determines how much of the precharging time that is hidden by the access pattern itself. The number of NOPs required to precharge all banks may be different after a read and a write pattern, since the values of t^{pre}_{read} and t^{pre}_{write} are unrelated. It is hence possible to derive two refresh patterns, one that follows read patterns, and one that follows write patterns. However, reducing the refresh pattern for one of these cases with a few clock cycles has very little impact on both bandwidth and latency and is hence not considered in this work. The refresh command is placed in cycle $tRP + (t^{pre}_{read} - t_{read})$, or $tRP + (t^{pre}_{write} - t_{write})$, whichever is larger. This is followed by a refresh command and $tRFC$ NOPs that

are required to satisfy the refresh-to-activate constraint. The equation for computing the length of refresh patterns is therefore:

$$t_{ref} = tRP + tRFC + \max(t^{pre}_{read} - t_{read}, t^{pre}_{write} - t_{write}) \tag{4.14}$$

The switching patterns only consist of NOP commands that allow the direction of the data bus to be reversed. We first explain how to compute the read/write switching pattern and then proceed with the write/read switching pattern. The number of NOPs in the read/write switching pattern depends both on the SDRAM generation and the burst length. For simplicity, we do not consider that DDR3 memories can change the burst length on the fly and assume that the burst length is fixed to either 4 or 8 words for both reads and writes. However, there should be no conceptual problems with supporting read patterns and write patterns with different lengths, as long as it does not change dynamically. Equation (4.15) shows the minimum number of clock cycles between a read and a write command for different memories and burst lengths. This equation is derived from the memory specifications [61, 62]. We compute the number of NOPs in the read/write switching pattern by subtracting the number of cycles between the read and write commands that are already built into the read and the write patterns. The length of the read/write switching pattern is hence computed according to (4.16). The computation of the write/read switching pattern is computed in a similar manner. The minimum delay between the write and the read command is shown in (4.17) and the length of the pattern is determined in (4.18).

$$\delta_{read} = \begin{cases} 4 & \text{DDR2 with } BL = 4 \\ 6 & \text{DDR2 with } BL = 8 \\ tCL + \frac{tCCD}{2} + 2 - tWL & \text{DDR3 with } BL = 4 \\ tCL + tCCD + 2 - tWL & \text{DDR3 with } BL = 8 \end{cases} \tag{4.15}$$

$$t_{rtw} = \max(\delta_{read} - (t^{first}_{write} + t_{read} - t^{last}_{read}), 0) \tag{4.16}$$

$$\delta_{write} = tWL + \frac{BL}{2} + tWTR \tag{4.17}$$

$$t_{wtr} = \max(\delta_{write} - (t^{first}_{read} + t_{write} - t^{last}_{write}), 0) \tag{4.18}$$

4.6 Architecture and Synthesis

The concepts in this chapter are embodied in hardware as an SDRAM back-end, according to the architecture previously shown in Fig. 2.5. The back-end is accessed through a Device Transaction Level (DTL) [101] port, where the scheduled request is presented by the bus in the resource front-end. The back-end consists of two major functional blocks, being a Command Generator and a Memory Map.

The Memory Map decodes the logical memory addresses used by the requestors, into a physical SDRAM address consisting of bank, row and column. The burst are mapped to the banks in an interleaving fashion, as mentioned in Sect. 3.4.3.2. For the example patterns of our example 16-bit DDR2-400 memory with $BL = 8$ and $BC = 1$ shown in Fig. 2.3, this is done by letting bit zero in the logical memory address index the byte, bits 4 to 5 index the bank, 13 to 25 index the row, and 1 to 3 and 6 to 12 index the column.

The Command Generator issues the appropriate memory patterns based on the refresh state, the read/write state, and the scheduled request. The patterns are hard-coded in a finite-state machine inside the Command Generator, which results in a small implementation. However, this also implies that the Command Generator must be modified to change the patterns in response to different use-case requirements, or if a different memory device is used. Although this is sufficient for an initial proof of concept, we consider a configurable Command Generator important future work.

The SDRAM back-end has been implemented in VHDL and tested together with a Verilog model of a Micron DDR2-400 memory [80]. The implemented model is a part of an older version of the proposed memory controller [7, 103], containing an integrated front-end and back-end. Apart from Command Generator and Memory Map, this implementation also contains a bus and a Credit-Controlled Static-Priority (CCSP) arbiter, further discussed in Chap. 5. This older memory controller is no longer maintained, in favor of the new more modular architecture. The design has been synthesized in a 0.13 µm CMOS technology. Synthesis with six ports and a speed target of 200 MHz, suitable for a DDR2-400, resulted in a total cell area of 42,000 μm^2. Note that all synthesis results in this book are obtained before place-and-route and that areas after layout are expected to be higher and maximum frequencies lower. We estimate the size of the current SDRAM back-end without the arbiter to 23,000 μm^2 by subtracting the area of the six port CCSP arbiter instance included in the design. The cell area of the design is small for an SDRAM controller, partly because it does not include buffers to store requests and responses. A second reason is that the design is customized for a particular memory and uses a simple finite-state machine to schedule commands in a way that is guaranteed not to violate any timing constraints. We do hence not require the large amount of registers required to track the state of the memory.

4.7 Experimental Results

We conclude the chapter by experimentally evaluating the proposed SDRAM back-end. We first describe the experimental setup, before conducting three experiments. The first experiment evaluates the three different memory pattern generation algorithms by comparing how much bandwidth they provide for different memories and burst counts. We consider gross bandwidth in this experiment to isolate the results from the influence of different request sizes. The different categories of gross memory efficiency are quantified, enabling us to learn about how efficiency is lost

Table 4.2 List of relevant timing parameters for some different 64 MB x16 (512 Mb) memory devices with page sizes of 2 KB

Parameter	DDR2-400 (cycles)	DDR2-800 (cycles)	DDR3-800 (cycles)	DDR3-1600 (cycles)
tRC	11	22	20	36
$tRCD$	3	4	5	8
tCL	3	4	5	8
tWL	2	3	5	8
tRP	3	4	5	8
$tRFC$	21	42	36	72
$tRAS$	8	18	15	28
$tRTP$	2	3	4	6
tWR	3	6	6	12
$tFAW$	10	18	20	32
$tRRD$	2	4	4	6
$tCCD$	2	2	4	4
$tWTR$	2	3	4	6
$tREFI$	1560	3120	3120	6240

for the different memories. In our second experiment, we take data efficiency into consideration and demonstrate that both burst count and memory device must be chosen carefully to maximize bandwidth in presence of small requests. For our last experiment, we evaluate the tightness of our derived bound on net bandwidth by simulating a SystemC model of our SDRAM back-end. No latency results are presented in this chapter. However, the tightness of the latency bound is evaluated in Chap. 5 when we discuss sharing the SDRAM back-end between multiple requestors.

4.7.1 Experimental Setup

The experiments in this section use our proposed SDRAM back-end together with four different memories with different speeds: DDR2-400, DDR2-800, DDR3-800, DDR3-1600. Each of these memories exists in a number of different speed bins, determining their exact timings, and we have consistently used the fastest possible version of every memory. All memories have a capacity of 512 Mb and 16-bit interfaces. The DDR2 memories have four banks, and the DDR3 memories eight. The relevant timing parameters of these memories are listed in Table 4.2. Brief explanations of the different memory timings are provided in Table 3.1.

4.7.2 Algorithm Evaluation

In our first experiment, we compare the different pattern generation approaches. The idea behind the experiment is to let all three algorithms generate a set of patterns for

Table 4.3 Pattern generation results for the DDR2-400 memory

	B&B & Bank scheduling				ASAP scheduling			
BL/BC	4/1	8/1	8/2	8/4	4/1	8/1	8/2	8/4
Dominance	wr	mix rd	mix rd	mix rd	wr	wr	wr	wr
t_{read}	11	16	32	64	11	16	32	64
t_{write}	13	16	32	64	13	21	37	69
t_{rtw}	0	2	2	2	0	2	2	2
t_{wtr}	0	4	4	4	0	0	0	0
t_{ref}	27	32	32	32	27	27	27	27
e^{gross}	60.5%	82.5%	89.6%	93.6%	60.5%	74.9%	85.0%	91.1%

burst counts 1, 2, and 4 with a burst length of 8 words. To provide a low-latency option, we also generate pattern sets with burst count 1 and burst length 4 for the DDR2 memories. This experiment just exercises the pattern generation tooling, and does not involve any implementation of the SDRAM back-end. To reduce the run time of the branch and bound algorithm, the lengths of the access patterns generated by the bank scheduling algorithm were used as initial shortest patterns. This significantly reduces the search space without the possibility of removing the shortest pattern.

4.7.2.1 DDR2-400

First up is our example DDR2-400 memory. Table 4.3 lists the lengths of the resulting patterns for the different algorithms and the corresponding gross memory efficiencies. We have merged the columns for the branch and bound algorithm and the bank scheduling algorithm, since they consistently provide the exact same pattern lengths for all tested memories. The table shows that all algorithms provide patterns with the same length for $BL = 4$. In fact, they even provide the exact same patterns. The reason is that the low burst count and short burst length results in short patterns, where the memory timings do not allow a lot of options. In contrast with $BL = 8$, we observe that the ASAP scheduling algorithm generates write patterns that are five cycles longer than those generated by the other algorithms. As explained in Sect. 4.5.4, this is because scheduling the activate commands as soon as possible causes the distance to the corresponding write commands to gradually increase, causing a problem with precharges. The generated read patterns all have the same lengths. The ASAP scheduling algorithm still schedules the activate commands much earlier, although this does not affect the length of the pattern, since the memory starts precharging faster after reads. Having a longer write pattern is not completely without advantages. We observe that the patterns generated by the ASAP algorithm often have shorter write/read switching patterns and refresh patterns. The reason is that the five NOPs at the end of the write patterns hide some of the time required to switch direction of the data bus, or to precharge all banks.

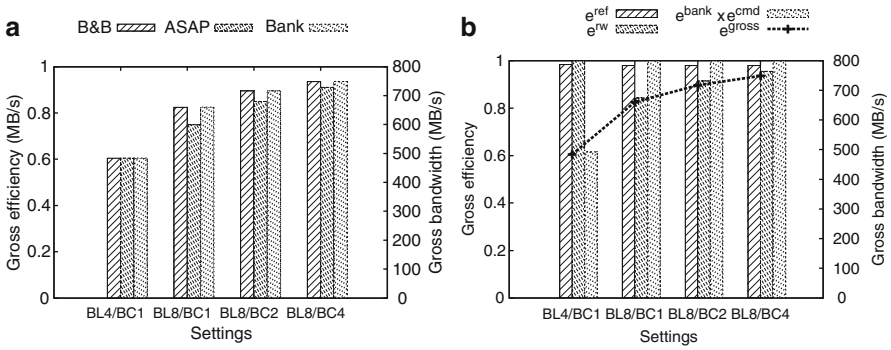

Fig. 4.16 Memory efficiency results for DDR2-400. (**a**) Bounds on gross efficiency and gross bandwidth for the different algorithms. (**b**) Bank scheduling gross efficiency breakdown

Next, we look at how the bounds on gross memory efficiency and gross bandwidth vary between the different algorithms for the DDR2-400 memory. This is shown in Fig. 4.16a. Note that the bars in the plot can be interpreted using either y-axis, depending on the metric of interest. The branch and bound algorithm and the bank scheduling algorithm perform identically, having generated patterns with the same length. However, the patterns with $BL = 8$ generated by the ASAP scheduling algorithm provide slightly less gross bandwidth than the patterns from the other two algorithms. The reason is that the longer write pattern reduces the bank and command efficiencies. The improved read/write efficiency and refresh efficiency (marginal) helps compensating for this drawback, but they do not manage to completely cancel out the effect. The difference is most pronounced for the patterns with $BC = 1$, where ASAP scheduling results in a reduction in memory efficiency by $1 - 0.749/0.825 = 10.2\%$. This shows that the choice of algorithm may have considerable impact on how efficiently the memory controller uses the scarce and expensive SDRAM bandwidth.

Figure 4.16b illustrates the impact of the different categories of gross efficiency for the pattern sets with the shortest access patterns, generated by the branch and bound and bank scheduling algorithms. We note that it is the bank and command efficiencies that cause problems for DDR2-400 with $BL = 4$. This because the access patterns only transfer data during 8 cycles and then have to wait a few cycles before the activate-to-activate constraint or precharge constraints are satisfied. However, these extra cycles completely eliminate the switching patterns, resulting in read/write efficiency of 100%. As the burst count and burst length increases, we note that the bank efficiency is 100% for this memory, as it transfers data during every cycle of the access patterns. Instead, the main loss of efficiency is now due to read/write switches, since this overhead is no longer hidden by the access patterns. The line in the figure clearly shows how the gross efficiency increases with increasing burst count, indicating that longer bursts to all banks help amortizing the switching costs.

Table 4.4 Pattern generation results for the DDR2-800 memory

	B&B & Bank scheduling				ASAP scheduling			
BL/BC	4/1	8/1	8/2	8/4	4/1	8/1	8/2	8/4
Dominance	mix rd	mix rd	mix rd	mix rd	mix rd	mix rd	mix wr	mix wr
t_{read}	22	22	33	65	22	22	33	65
t_{write}	22	22	33	65	22	22	36	68
t_{rtw}	0	0	1	1	0	0	1	1
t_{wtr}	1	3	5	5	1	3	2	2
t_{ref}	57	57	58	58	57	57	55	55
e^{gross}	34.9%	66.8%	87.2%	92.4%	34.9%	66.8%	87.3%	92.4%

As far as the run times of the algorithms are concerned, the ASAP scheduling and bank scheduling algorithms provided all results in a matter of seconds. The branch and bound algorithms managed to produce patterns with low burst counts in comparable time. However, the pattern set with $BC = 4$ took 8 days to generate. Such a long run-time clearly motivates the existence of the heuristic algorithms.

4.7.2.2 DDR2-800

We proceed by looking at the results for the DDR2-800, the fastest device in the generation of DDR2 memories. The patterns generated for this memory and their gross efficiencies are listed in Table 4.4. The difference between the algorithms is that ASAP scheduling again generates longer write patterns for some values of burst count and burst length. The increase is slightly less severe than for the DDR2-400, since the precharging constraints are more favorable for this memory. The minimum spacing between activates to different banks, $tRRD$, is increased from two to four cycles, moving all but the first activate commands further into the access patterns. This gives more time to precharge the banks after an access pattern before they are reactivated in a later pattern. The timing constraints that determine the precharge cycle increase too for this memory, but not enough to cancel out the benefits.

Looking at the gross efficiency for the different algorithms in Fig. 4.17a, we observe that the ASAP scheduling algorithm is not performing worse than the branch and bound algorithm and bank scheduling. In fact, the longer write patterns result in that the gross efficiency is marginally *increased* by 0.1%! This is explained by observing that the patterns are mix-dominant and that increasing the write pattern with three cycles removes three cycles from the write/read switching pattern, eliminating the disadvantage. The slight increase in efficiency stems from that the longer write pattern also allows the refresh pattern to be shorter. This demonstrates that the shortest access patterns do not always provide the best efficiency, although the difference in this case is negligible. Comparing the results of DDR2-800 to our earlier results for DDR2-400 shows that the efficiency of the faster memory is lower for any burst count, as discussed in Sect. 3.3.7. However, the gross bandwidth of DDR2-800 is still higher than for DDR2-400, since the peak bandwidth of the faster memory is twice as high.

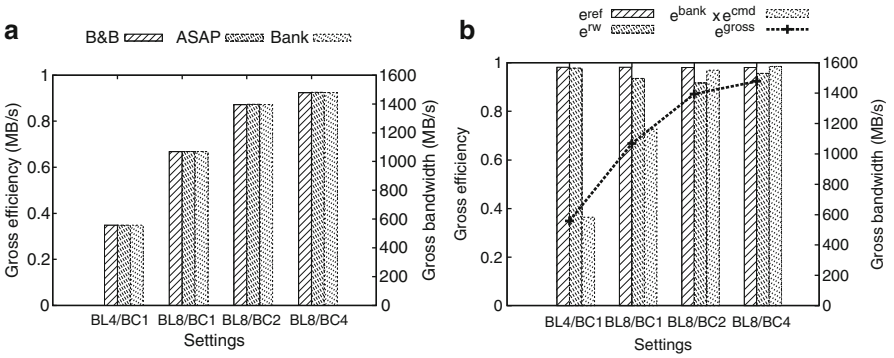

Fig. 4.17 Memory efficiency results for DDR2-800. (**a**) Bounds on gross efficiency and gross bandwidth for the different algorithms. (**b**) Bank scheduling gross efficiency breakdown

The gross efficiency breakdown in Fig. 4.17b reveals that it is the bank and command efficiency that causes the most significant efficiency loss for this memory with lower burst counts. This happens because the time between consecutive activate commands to the same bank, tRC, is 22 cycles, effectively preventing any access pattern from being shorter than that. This in turns allows the access patterns to hide much of the read/write switching time, resulting in high read/write efficiency.

Considering the run-times of the algorithms, just like for DDR2-400, all patterns were generated in a few seconds with the exception of $BC = 4$, which took the branch and bound algorithm 32 min.

4.7.2.3 DDR3-800

The next memory is DDR3-800, the slowest memory in the DDR3 generation. We are interested in this memory, since it provides the same peak bandwidth as the DDR2-800. Apart from the difference in memory generation, our DDR3-800 memory comes with 8 banks instead of 4. We do not evaluate $BL = 4$ for DDR3 memories, since this is only supported by means of a burst chopping mechanism. Bursts of 4 words are hence not much faster than burst of 8 words. The generated patterns for this memory are shown in Table 4.5. The results from all algorithms are merged, since they always provide patterns of the same lengths for this memory. A possible reason for this is that eight banks resolves the precharging problem of the ASAP algorithm, since the last activate command slips further into the pattern. Eight bank memories also have the additional FAW constraint, which limits the number of activate commands in a window of $tFAW$ cycles. This constraint helps spacing the activate commands in the pattern more evenly, further mitigating the precharging issue. However, this constraint does not primarily make patterns shorter. Both access patterns with $BC = 1$ have two NOP commands added in the end to ensure that the FAW constraint is satisfied also when the patterns are repeated after themselves.

Table 4.5 Pattern generation results for the DDR3-800 memory

	All algorithms		
BL/BC	8/1	8/2	8/4[a]
Dominance	mix rd	mix rd	mix rd
t_{read}	40	66	130
t_{write}	40	66	130
t_{rtw}	0	0	0
t_{wtr}	5	7	7
t_{ref}	53	55	55
e^{gross}	74.0%	90.5%	94.2%

[a] The B&B algorithm did not finish in less than 10 days for this setting

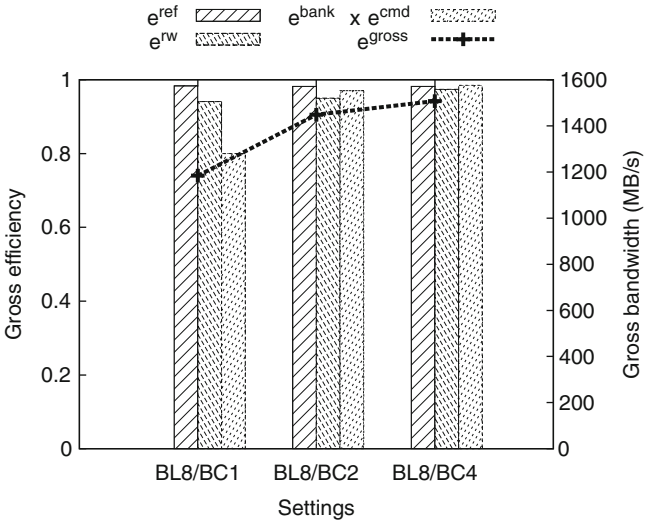

Fig. 4.18 Bank scheduling gross efficiency breakdown for DDR3-800

Since all three algorithms perform identically, we proceed directly to the gross efficiency breakdown in Fig. 4.18. The breakdown is similar to that of DDR2-800. Most of the efficiency loss for $BC = 1$ is due to bank and command efficiencies, which reduce with increasing burst count. Overall, DDR3-800 has a gross efficiency that is a few percent higher than DDR2-800.

The additional banks impact the run time of the branch and bound algorithm. More banks imply more commands to schedule, creating more possible patterns. The pattern set with $BC = 1$ still completed within seconds. However, the patterns with $BC = 2$ took 39 h to complete, and the patterns with $BC = 4$ where still not finished after 10 days when we terminated the experiment.

Table 4.6 Pattern generation results for the DDR3-1600 memory

	All algorithms		
BL/BC	8/1	8/2	8/4[a]
Dominance	mix rd	mix rd	mix rd
t_{read}	64	70	133
t_{write}	64	70	133
t_{rtw}	0	0	0
t_{wtr}	4	9	9
t_{ref}	98	103	103
e^{gross}	47.7%	84.5%	91.6%

[a] The B&B algorithm did not finish in less than 10 days for this setting

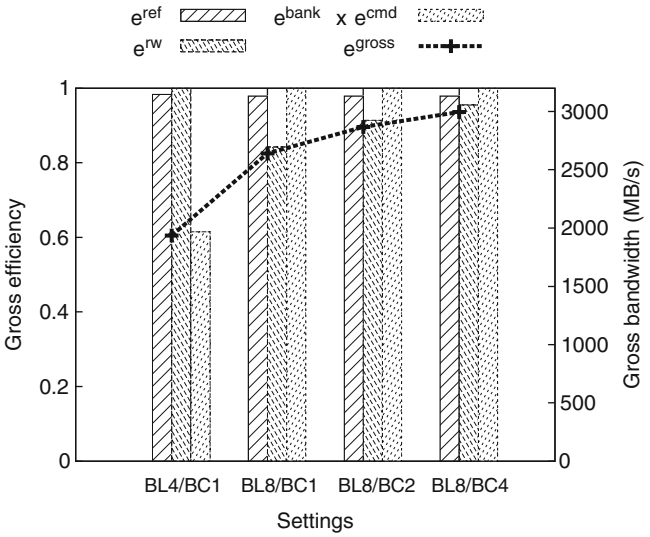

Fig. 4.19 Bank scheduling gross efficiency breakdown for DDR3-1600

4.7.2.4 DDR3-1600

Our last memory in the pattern generation experiment is a DDR3-1600, doubling the peak bandwidth over DDR3-800. Just like for the previous memory, all algorithms perform the same and provide the results shown in Table 4.6. We observe that there is not a big difference in the length of the access patterns with $BC = 1$ and $BC = 2$. The reason is that the FAW constraint of 32 cycles postpones the fifth activate command by eight cycles in both access patterns with $BC = 1$. The same constraint also adds an extra five NOPs at the end of these access patterns to allow them to repeat after themselves.

The gross efficiency breakdown in Fig. 4.19 does not show us much new over DDR3-800. We observe that the gross efficiency is lower for DDR3-1600 than for

DDR3-800, proving the efficiency trend for faster memories yet again. Still, the peak bandwidth is doubled compared to the slower memory, resulting in increased gross bandwidth.

The branch and bound algorithm required 7 days to generate the pattern set with $BC = 1$, although the set with $BC = 2$ was generated in seconds. The algorithm had not successfully generated a pattern set with $BC = 4$ after 10 days when we terminated the experiment. Just like always, the two heuristic algorithms produced all results in just seconds.

4.7.2.5 Conclusions

This experiment allows us to draw several interesting conclusions both regarding memory pattern generation and bandwidth trends in DDR2/DDR3 memories. We first present five conclusions about memory pattern generation: (1) *By considering all patterns generated by our algorithms, we observe that all generated read patterns are shorter than or equal to the corresponding write patterns. Similarly, read/write switching patterns are always shorter than write/read switching patterns.* In both cases, this is related to the fact that a bank requires more time to precharge after a write burst. A result of this relation is that we have not generated any read-dominant pattern sets in this experiment. In fact, it is possible that read-dominant pattern sets cannot be optimal for current DDR2 and DDR3 memories. (2) *The choice of memory pattern generation algorithm matters*. The difference in efficiency between the best algorithm and the worst algorithm is up to 10.2% of net bandwidth. This is a significant improvement, since SDRAM bandwidth is a scarce and expensive resource. (3) Given the large set of valid patterns for modern memories *exhaustive search is not a viable option*, even with the design-space reduction provided by our design decisions. (4) *ASAP scheduling of SDRAM commands is not a good approach for memories with four banks*, as precharge constraints may cause pattern lengths to increase, reducing efficiency. (5) *Bank scheduling provides the same results as the branch and bound algorithm* for all tested memories and burst counts. This suggests that it is a fast heuristic that reproduces the optimal results, given our design decisions.

Next, we draw two conclusions about bandwidth trends in DDR2/DDR3 memories: (1) *Gross efficiency increases with burst count, although the increase becomes smaller for every increment.* This is shown for all tested memories when comparing their results with the bank scheduling algorithm in Fig. 4.20a. (2) *Newer faster memories offer higher peak bandwidths, but lower gross efficiency, due to increasingly severe timing constraints. However, the provided gross bandwidth is still increasing with clock frequency.* Figure 4.20a indicates that gross efficiency is reducing as memories get faster. It also shows that DDR3-800 has higher gross efficiency than DDR2-800. The fact that gross bandwidth is increasing despite the reducing gross efficiency is clearly shown in Fig. 4.20b, where DDR3-1600 provides the highest gross bandwidth.

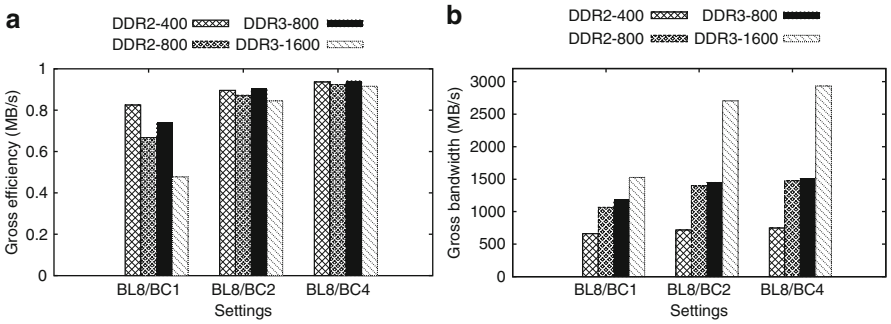

Fig. 4.20 Gross efficiency and gross bandwidth comparisons between different DDR2 and DDR3 memories. (**a**) Gross efficiency comparison. (**b**) Gross bandwidth comparison

4.7.3 Bounding Net Bandwidth

For our second experiment, we take data efficiency into account and bound the net bandwidth offered by the memories. Figure 4.21 shows the bound on net bandwidth provided by the different memories and settings for different request sizes, based on the patterns generated by the bank scheduling algorithm. For simplicity, we assume that the request sizes of all requestors are the same, since this allows us to compute the data efficiency independently of the memory arbiter. The bars in the plot can be read from either y-axis, depending on if net bandwidth or net efficiency is of interest. All graphs have the same scale, allowing the net bandwidths provided by the different memories to be compared. From this experiment, we learn that while increasing burst count consistently increases gross bandwidth, it may reduce net bandwidth. The reason is that increasing burst count also increases the access granularity of the memory, resulting in more waste for small requests. This trend is clearly visible for requests of 256 B (bytes) as the DDR3-1600 memory moves from $BC = 2$ to $BC = 4$. This reduction of net bandwidth is guaranteed to occur no later than when the access granularity of the memory becomes larger than the request size of all requestors. Similarly, increasing the number of banks from 4 to 8 improves bank and command efficiencies, but can still reduce net bandwidth, due to the larger access granularity. A consequence of this behavior is that our DDR2-800 provides more net bandwidth for small requests than the DDR3-800. However, the tables turn as requests become big enough to benefit from the larger granularity. The figure also shows that achieving really high bandwidths with an interleaving memory map fundamentally requires large requests. In fact, the DDR2 memories with 4 banks require requests of 64-128 B to provide a net memory efficiency above 80%, while the DDR3 memories require requests of 256 B to accomplish the same. If the requests in the system are small, there is hence no benefit in using a faster SDRAM memory unless it is cheaper to buy. A good example of this is that the DDR2-800 with four banks provides the most net bandwidth for requests of 32 B.

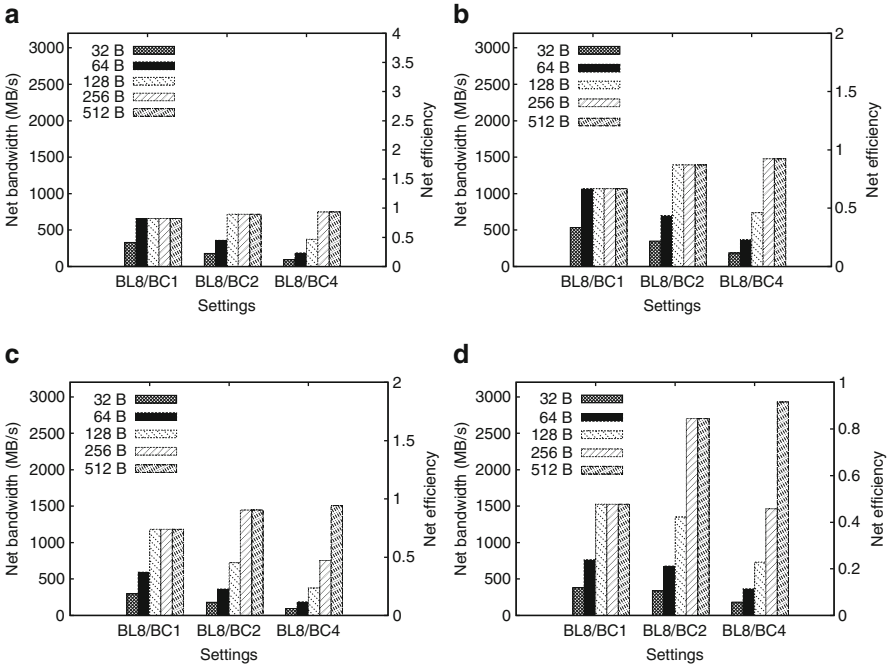

Fig. 4.21 Bound on net bandwidth for different memories and request sizes. (**a**) Net bandwidth with a DDR2-400. (**b**) Net bandwidth with a DDR2-800. (**c**) Net bandwidth with a DDR3-800. (**d**) Net bandwidth with a DDR3-1600

4.7.4 Tightness of Net Bandwidth Bound

In our third and last experiment, we evaluate the tightness of our lower bound on net bandwidth by simulation using a SystemC model of our proposed SDRAM back-end. We measure the running average net bandwidth, which we expect to converge to a value greater than or equal to our derived bound during the simulation. The experiment is conducted by sending an equal mix of read and write requests to our example DDR2-400 memory using the shortest mix-read-dominant pattern set with $BC = 1$, computed in the first experiment. The sizes of the requests are 64 B, which is equal to the access granularity of the pattern, thus providing a data efficiency of 100%. The bound on net bandwidth with this setup is 660 MB/s. We simulate the memory controller back-end twice. In the first simulation, we let read and write requests arrive in a random order. In the second simulation, arriving requests are alternating reads and writes to illustrate what happens during worst-case conditions. The simulation time in both cases is 100 ms. The results of this experiment are shown in Fig. 4.22, where the provided net bandwidth is plotted over time. Figure 4.22a shows the first 16 µs of the simulation, which is just

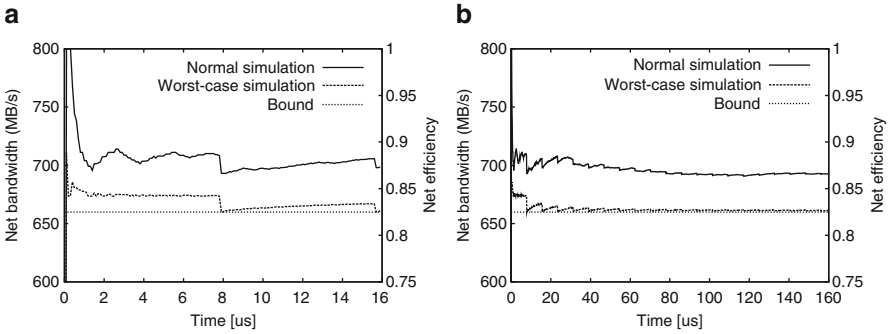

Fig. 4.22 Net bandwidth plotted over time for a DDR2-400 memory with and without worst-case switches. (**a**) The first 16 µs of the simulation. (**b**) The first 160 µs of the simulation

enough to get two interfering refresh patterns. In both simulations, net bandwidth shoots towards 800 MB/s as the first request arrives. This is because the bank and command efficiency of the patterns is 100% and hence that data is transferred on every cycle of the pattern. The efficiency then gradually reduces as read/write switches cause lost cycles on the data bus. We note that the impact of these switches is considerably higher when the worst-case switching behavior is enforced. We see the effects of refresh at 7.8 µs and again at 15.6 µs, where the efficiency reduces due to the 32 cycles required to precharge all banks and refresh the memory. The measured bandwidth is very close to the bound at the end of the refresh pattern, indicating that this is the time at which the memory efficiency calculation "evens out". This is not surprising, considering that all events covered by the bound, such as read/write switches and refresh, have happened at this time. After 100 ms when the simulation ends, the worst-case simulation converges at a net bandwidth of 661.0 MB/s, which is less than 0.2% from the derived bound. This is not completely unexpected, since we have enforced exactly the behavior assumed by the bound. The normal simulation, on the other hand, converges at 694 MB/s, thus providing about 4% extra net bandwidth due to the reduced number of read/write switches. This convergence is visualized in Fig. 4.22b, which shows the efficiency during the first 160 µs of the simulation. This experiment is shown also for DDR2-800, DDR3-800, and DDR3-1600 in [51]. A similar experiment, although without enforcing the worst-case scenario, is furthermore conducted with the VHDL implementation of the back-end together with a Micron DDR2-400 memory model in [103].

4.8 Summary

Our approach to predictability involves combining predictable resources with predictable arbitration. This chapter addressed the first part of this approach by introducing a predictable SDRAM back-end that increases the level of dynamism

compared to previous work. The proposed back-end is shared using predictable *dynamic front-end arbitration* to be able to satisfy diverse latency requirements, while remaining analyzable. The command generator uses a new *hybrid approach* that combines elements of static and dynamic command scheduling, enabling it to accommodate traffic that is not fully known at design time in a predictable fashion. The hybrid approach is based on *memory patterns*, which are precomputed sequences of SDRAM commands that are dynamically instantiated and combined by the command generator at run-time.

A *pattern set* consists of five memory patterns: a *read pattern*, a *write pattern*, a *read/write switching pattern*, a *write/read switching pattern*, and a *refresh pattern*. The read and write patterns access the memory by issuing a fixed number of bursts to each of the banks in an interleaving fashion. The read/write switching patterns and write/read switching patterns are used to give the data bus time to switch direction between a read and a write pattern. The refresh pattern is issued regularly to prevent leakage in the DRAM cells from causing data loss.

A pattern set is classified as either *read-dominant*, *write-dominant*, or *mix-dominant*, depending on which combination of patterns that results in the lowest bandwidths and longest latencies. A pattern set is read or write-dominant if the worst case happens if all interfering requests are either reads or writes, resulting in that only read or write patterns are issued along with an occasional refresh pattern. On the other hand, the worst-case situation for a mix-dominant pattern is if requests alternate between reads and writes, causing the maximum number of read/write switches. A mix-dominant pattern is further classified as *mix-read-dominant* or *mix-write-dominant* depending on if an odd number of requests contain more reads or writes in the worst case. The gross and net bandwidths provided by a pattern set were computed for all dominance types by bounding the five categories of memory efficiency introduced in the previous chapter. We also bounded the maximum time required to serve an arbitrary number of atomic service units, delivering on the requirements for the controller to be predictable.

Three algorithms for automatic memory pattern generation were presented, representing different trade-offs between net bandwidth and the run time of the algorithm. The algorithms try to compute the shortest possible read and write patterns and then generate the accompanying switching and refresh patterns. The first algorithm uses a *branch and bound* approach to exhaustively evaluate all valid patterns, branching only when a given pattern cannot become shorter than the shortest one currently found. This algorithm is guaranteed to find the shortest read and write patterns, but has a run time in the range of weeks or months when the generated patterns are long. The second algorithm uses *as-soon-as-possible (ASAP) scheduling* and tries to schedule SDRAM commands at the earliest possible time, prioritizing read and write commands over activates in case two commands can be scheduled in the same cycle. This algorithm runs in less than a second, but occasionally generates patterns providing 10% less bandwidth than the branch and bound algorithm. The last algorithm is called *bank scheduling*, as it schedules

commands for one bank at a time. This results in patterns offering the same bandwidth as the branch and bound algorithm in all our tests, while having a run time comparable to the ASAP scheduling algorithm.

We experimentally concluded that newer faster memories offer higher peak bandwidths, but lower gross efficiency, due to increasingly severe timing constraints. However, the provided gross bandwidth is still increasing with clock frequency. It was shown that gross memory efficiency increases with burst count, although the increase becomes smaller for every increment. We also concluded that large request sizes are required to achieve high net memory efficiency. A DDR2 memory requires request sizes between 64-128 B to provide a net memory efficiency above 80%, while the DDR3 memories require requests of 256 B to accomplish the same.

Chapter 5
Resource Arbitration

The previous chapter presented a memory controller back-end that makes an SDRAM into a predictable resource, corresponding to the first part of our approach to predictability. The second part of the approach, which is the topic of this chapter, considers sharing this resource among multiple requestors in a predictable manner. The context of this problem was previously shown in Fig. 1.7, where requests arrive in a Request Buffer in front of a resource arbiter and responses are returned in a Response Buffer. Resource arbitration with real-time requirements is in no way a new research field. In fact, research has been conducted in this field during more than half a century already and there exists a plethora of different arbiters. Still, new applications and emerging technologies like heterogeneous multi-core System-on-Chips (SoCs) continue to change the requirements, as they need small and fast arbiters that cater to diverse requirements without wasting scarce resource capacity

We start this chapter in Sect. 5.1 by elaborating on the requirements from the SoC context, and from the requestors in our considered application domains. Section 5.2 then augments our formal model with definitions related to resource arbitration. Section 5.3 uses this model to introduce Latency-Rate ($\mathcal{LR}$) servers, which is a general framework for analyzing and comparing arbiters. This framework abstracts the detailed behavior of predictable arbiters by using a simple common representation of their service guarantee. We then proceed by presenting three arbiters belonging to the class of $\mathcal{LR}$ servers: Time-Division Multiplexing (TDM), Frame-Based Static-Priority (FBSP), and Credit-Controlled Static-Priority (CCSP) in Sects. 5.4, 5.5, and 5.6, respectively. An overview is provided for each arbiter, followed by derivation of bounds on latency and wasted resource capacity. Section 5.7 then practically demonstrates the service guarantee of the shared predictable memory controller and experimentally evaluates the latencies and wasted resources for the three presented arbiters. Lastly, we conclude with a summary in Sect. 5.8.

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_5, © Springer Science+Business Media, LLC 2011

5.1 Arbiter Requirements

An important difference between memory arbitration and, for instance, processor scheduling is that a memory arbiter works at a much finer level of granularity [117]. The execution time of a task may range from microseconds to milliseconds, while a memory request in an SRAM controller is served in a few nanoseconds. This is one important reason why resource arbiters are implemented in hardware instead of software. There are three main requirements on the hardware implementation of the arbiter to make it applicable to this type of resource. (1) It must run at high clock frequency to keep up with the resource and to be able to schedule small requests. (2) It must have a small hardware implementation to limit the impact on area. (3) The arbiter must be able to provide the required service to a requestor without reserving more capacity than required, referred to as *over-allocation*. Limiting over-allocation is imperative, since memory bandwidth is scarce and must be efficiently utilized. The arbiter must not only consider the requirements of the SoC context, but also those of the requestors in our application domains, previously discussed in Sect. 1.1.6. It must hence be able to accommodate diverse bandwidth requirements and both latency-sensitive and latency-tolerant requestors [126].

5.2 Formal Model

We proceed in this section by extending our formal model with definitions and terminology required to deal with resource arbitration. As previously mentioned in Sect. 2.1, our approach to predictability is based on combining independent analyses of the resource and the arbitration. To emphasize the generality of this approach, and its applicability to a wide range of resources, we abstract from a particular target resource. Some of the definitions in this section are hence more general than required by the proposed memory controller. Still, we chose to include these to increase the applicability of the provided theory. However, for simplicity, we limit the discussion to individual independent resources, such as memories. Resources with multiple internal arbiters, such as Network-on-Chips (NoCs), are not addressed here. First, we discuss the requested service model, which considers the behavior of the requestors and the resource, but not the arbiter. We then present the provided service model, covering resource arbitration. We remind the reader that a complete list of symbols along with brief descriptions and page references to the definitions are found in Appendix B.2.

5.2.1 Requested Service Model

Our independent arbiter analysis uses an abstract resource view, where a *service unit* corresponds to the access granularity of the resource. For a typical SRAM, the

access granularity is a single word, and for the proposed SDRAM back-end it is the granularity of a read or write pattern, g, previously defined in Definition 4.2. The size of a request in service units is hence computed according to Definition 5.1. Note that in the architecture of the proposed memory controller, previously shown in Fig. 2.9, the Request Buffers are located inside the Delay Block, and hence after the Atomizer. This means that the arriving requests are atoms and are hence guaranteed to have a size of a single service unit by definition. Just like in previous chapters, time is discrete and counts from zero. A time unit, referred to as a *service cycle*, is defined as the time required to serve a request with the size of one service unit. The length of a service cycle, measured in clock cycles, is expressed according to Definition 5.2. A Zero-bus-turnaround (ZBT) SRAM has a constant service cycle length of one clock cycle. On the other hand, an SDRAM has a highly variable service cycle length that depends on whether the request is a read or a write and the state of the memory at the time it is scheduled. Multiplying a latency in service cycles with the maximum service cycle length, which is known and bounded for predictable resources, always provides a conservative latency in absolute time. While this approach works well for an SRAM, it is too pessimistic for our SDRAM back-end, since it considers an interfering read/write or write/read switch and a refresh for every single request. Instead, we use the specialized latency $t_{tot}(x)$ from (4.8) for this particular resource. This equation accurately accounts for the maximum possible interfering read/write switches and refreshes for a sequence of requests.

Definition 5.1 (Request size (service units)). The size of a request ω_r^k in service units is given by $s(\omega_r^k) : \Omega_r \rightarrow \mathbb{N}^+$, and is defined as $s(\omega_r^k) = \lceil s^{bytes}(\omega_r^k)/g \rceil$.

Definition 5.2 (Service cycle length). The length of the service cycle, measured in clock cycles, when servicing a request ω_r^k at time t is given by $\lambda(\omega_r^k, t) : \Omega_r \times \mathbb{N} \rightarrow \mathbb{N}$.

We use service curves [22] to model the interaction between the resource and the requestors. These service curves are typically cumulative and monotonically non-decreasing in time. We start by defining an operator for retrieving the value of a service curve in Definition 5.3. We use closed discrete time intervals throughout this book. The interval $[\tau, t]$ hence includes all service cycles in the sequence $\langle \tau, \tau+1, ..., t-1, t \rangle$. Definition 5.4 defines a compact notation for expressing the difference in values between the endpoints of such an interval.

Definition 5.3 (Value of a service curve). The value of a service curve ξ in service units at service cycle t is given by $\xi(t) : \mathbb{N} \rightarrow \mathbb{N}$.

Definition 5.4 (Difference in values between endpoints of an interval). The difference in values between the endpoints of the closed interval $[\tau, t]$, where $t \geq \tau$, of a service curve ξ is given by $\xi(\tau, t) : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, and is defined as $\xi(\tau, t) = \xi(t+1) - \xi(\tau)$.

A requestor generates requests according to a requested service rate, as defined in Definition 5.5. This rate expresses the requested fraction of the total service units provided by the resource, and is defined as $\rho_r = b_r / b^{net}$ for the special case where

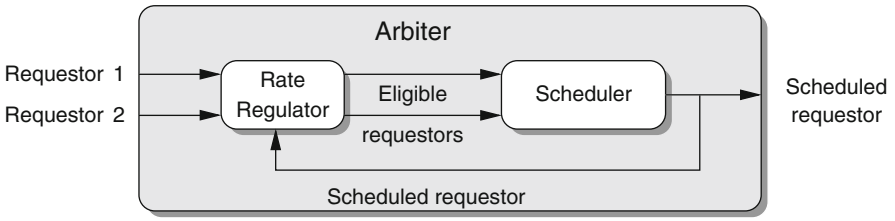

Fig. 5.1 An arbiter comprising a rate regulator and a scheduler

data efficiency is 100%, making gross and net bandwidth identical. The general case is discussed in Chap. 7. A request is considered to arrive at a resource when: (1) the last bit of the request has arrived in the Request Buffer, and (2) there is enough space in the Response Buffer to store a potential response, as stated by Definition 5.6. This is captured by the requested service curve, w, defined in Definition 5.7. For clarity, it is assumed that only a single request arrives per requestor in any service cycle, although this is straight-forward to generalize. Note that Definitions 5.6 and 5.7 state that a requested service curve at time $t+1$ accounts for a request with arrival time $t+1$.

Definition 5.5 (Requested service rate). The requested service rate of a requestor $r \in R$, expressed in service units/service cycle, is denoted ρ_r.

Definition 5.6 (Arrival time). The arrival time of a request ω_r^k from a requestor $r \in R$ is given by $t_a(\omega_r^k) : \Omega_r \to \mathbb{N}^+$, and is defined as the smallest t at which the last bit of ω_r^k has arrived in the Request Buffer and there is enough free space in the Response Buffer to store a potential response.

Definition 5.7 (Requested service curve). The requested service curve of a requestor $r \in R$ is given by $w_r(t) : \mathbb{N} \to \mathbb{N}$, where $w_r(0) = 0$ and

$$w_r(t+1) = \begin{cases} w_r(t) + s(\omega_r^k) & \exists \omega_r^k : \ t_a(\omega_r^k) = t+1 \\ w_r(t) & \nexists \omega_r^k : \ t_a(\omega_r^k) = t+1 \end{cases}$$

5.2.2 *Provided Service Model*

Having discussed how the requested service model captures the behavior of the requestors and the resource, we present the provided service model that covers the arbiter. Most arbiters can be discussed in terms of two main components [141], a *rate regulator* and a *scheduler*, as shown in Fig. 5.1. The purpose of a rate regulator is to protect requestors that do not ask for more service than they are allocated from the ones that do. This is done by determining which requests are *eligible* for scheduling at a particular time. It is then the responsibility of the scheduler to

choose which eligible requestor to schedule, based on its particular policy. We first discuss terminology and concepts related to rate regulators and then proceed with schedulers.

A rate regulator typically uses an *accounting* (budgeting) mechanism to determine which requestors are eligible for scheduling. The accounting is often related to the allocated rate of the requestor, defined in Definition 5.8, which determines its allocated fraction of the resource capacity. This enables a guaranteed minimum service to be provided if paired with an *enforcement* mechanism that ensures that requestors that are out of budget (not eligible) cannot be scheduled while there are requestors with remaining budget waiting to access the resource. This protection of eligible requestors is a key property in providing guaranteed service to requestors with timing constraints [140].

Definition 5.8 (Allocated rate). The allocated rate of a requestor $r \in R$ is denoted $\rho'_r \in \mathbb{R}^+$. For a valid allocation it holds that $\forall r \in R : \rho'_r \geq \rho_r$ and $\sum_{\forall r \in R} \rho'_r \leq 1$.

Definition 5.8 states two constraints that must be satisfied in order for an allocation to be valid: (1) the allocated rate must be at least equal to the average request rate of the requestor, ρ, to satisfy its service requirement, and (2) it is not possible to allocate more service to the requestors than what is offered by the resource. Note that the allocated rate is a real value, possibly requiring infinite precision to be accurately represented.

An arbiter may not always be able to accurately represent the allocated rate, causing it to be discretized. This discretization may either be inherent to the design of the allocation mechanism in the rate regulator, or be a result of limited precision in its hardware implementation. We will see examples of both cases later in this chapter. To capture this discretization, we associate each requestor with a discrete allocated rate, denoted by ρ'', that *conservatively approximates* the real-valued allocated rate, ρ'. Note that similarly to the total allocated rate, the total discrete allocated rate may not exceed the total capacity of the resource. The discrete allocated rate is formally defined in Definition 5.9. The conservative approximation of the allocated rate may cause it to be over-allocated. We define the over-allocated rate of a requestor according to Definition 5.10. This definition shows us how much of the resource capacity is wasted when service is allocated to a requestor.

Definition 5.9 (Discrete allocated rate). The discrete allocated rate of a requestor $r \in R$ is denoted $\rho''_r \in \mathbb{Q}^+$. For a valid discrete allocation it holds that $\forall r \in R : \rho''_r \geq \rho'_r$ and $\sum_{\forall r \in R} \rho''_r \leq 1$.

Definition 5.10 (Over-allocated rate). The over-allocated rate of a requestor $r \in R$ is given by $o_\rho(\rho''_r, \rho'_r) : \mathbb{Q}^+ \times \mathbb{R}^+ \to \mathbb{R}$, and is defined according to $o_\rho(\rho''_r, \rho'_r) = \rho''_r - \rho'_r$.

Having introduced important concepts and terminology of rate regulators, we proceed by briefly discussing schedulers. A scheduler may be either *work-conserving* or *non-work-conserving*. The difference between these types of

schedulers is that a work-conserving scheduler never idles when there are waiting requests from a requestor. Instead, it schedules a waiting requestor according to a slack management policy without reducing its budget. The benefit of work-conservation is that it reduces average latency of requestors and increases resource utilization without negatively impacting worst-case latency. However, a potential drawback of work conservation is that the service provided to a requestor may become burstier, potentially increasing buffer requirements in systems that are not allowed to stall due to overflowing buffers [28, 118, 141]. Note that both work-conserving and non-work-conserving schedulers always prefer scheduling eligible requestors and the same service guarantee is provided in both cases.

Another important property of a scheduler is whether or not it is *preemptive*. A non-preemptive scheduler that schedules a request cannot schedule another request before the first request finished receiving service. In contrast, a preemptive scheduler may suspend a request receiving service, save its state, and schedule another request. When the suspended request is scheduled at a later point in time, its state is loaded, and service is resumed. Preemption can occur either at arbitrary points in time, or be limited to pre-defined preemption points. The benefit of the latter is that preemption can be limited to points in time when there is little state to save, such as between complete service units. Preemption points are furthermore required by some resources, since they cannot be preempted at any given time. This is the case for most SDRAM memories, which cannot be preempted arbitrarily when transferring a burst. The benefit of preemption is that it reduces blocking from large requests. However, preemption complicates the implementation of the scheduler, since a mechanism to save and load state has to be added. Latencies may furthermore increase if preemption incurs overhead, as is typically the case for schedulers implemented in software.

This work considers arbitration with preemption points on the granularity of service units. This applies to many arbiters in general and to all arbiters in our architecture, due to the presence of the Atomizer. The arbiter schedules a requestor every service cycle according to its particular scheduling policy, as stated in Definition 5.11. Note that a request ω^k has to be scheduled $s(\omega^k)$ times before it is finished.

Definition 5.11 (Scheduled requestor). The scheduled requestor at time t is given by $\gamma(t) : \mathbb{N} \rightarrow R \cup \{\varnothing\}$, where $\varnothing$ denotes that no requestor is scheduled.

The provided service curve, w', defined in Definition 5.12, reflects the number of service units provided by the resource to a requestor. A service unit takes one service cycle to serve. This is reflected in that the provided service is increased at $t+1$ if a requestor is scheduled at t. An illustration of a requested service curve and a provided service curve is provided in Fig. 5.2. For reasons of clarity, the curves in the figure are drawn as continuous functions, although their values are only defined at discrete points in time.

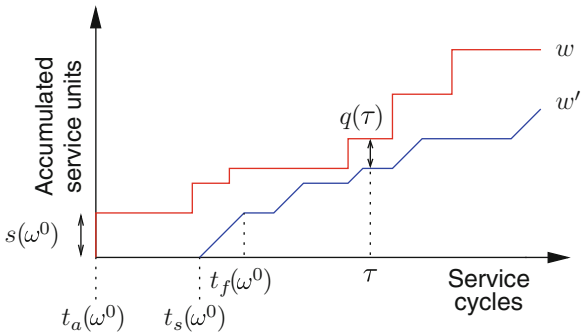

Fig. 5.2 A requested service curve, w, a provided service curve, w', and representations of the related concepts

Definition 5.12 (Provided service curve). The provided service curve of a requestor $r \in R$ is given by $w'_r(t) : \mathbb{N} \to \mathbb{N}$, where $w'_r(0) = 0$ and

$$w'_r(t+1) = \begin{cases} w'_r(t)+1 & \gamma(t) = r \\ w'_r(t) & \gamma(t) \neq r \end{cases}$$

The backlog of a requestor, defined in Definition 5.13, corresponds to the number of requested service units that has not yet been served at a particular time. A requestor that has a backlog greater than zero has outstanding requests, and is referred to as a backlogged requestor. The graphical interpretation of backlog is shown in Fig. 5.2.

Definition 5.13 (Backlog). The backlog of a requestor $r \in R$ at a time t is given by $q_r(t) : \mathbb{N} \to \mathbb{N}$, and is defined as $q_r(t) = w_r(t) - w'_r(t)$.

We previously defined the arrival time of a request in Definition 5.6. Now we define two more events in the life of a request, being the starting time and the finishing time, respectively. The starting time, $t_s(\omega^k)$, is the first service cycle in which ω^k is scheduled by the arbiter. Since atoms in the proposed memory controller have a request size of one service unit, it follows that the starting time corresponds to the one and only time it is scheduled. The finishing time of a request corresponds to the first service cycle in which a request is completely served and available in the Response Buffer, formally defined in Definition 5.15.

Definition 5.14 (Starting time of a request). The starting time of a request ω_r^k is given by $t_s(\omega_r^k) : \Omega_r \to \mathbb{N}^+$, and is defined as the smallest t at which ω_r^k is scheduled.

Definition 5.15 (Finishing time of a request). The finishing time of a request ω_r^k is given by $t_f(\omega_r^k) : \Omega_r \to \mathbb{N}^+$, and is defined as $t_f(\omega_r^k) = \min(\{t \mid w'_r(t) = w'_r(t_s(\omega_r^k)) + s(\omega_r^k)\})$.

5.3 Latency-Rate Servers

The formal model presented how service curves are used to capture the behaviors of requestors, the resource, and the resource arbiter. However, the model only describes the actual service provided by the shared resource, while formal verification requires knowledge about the worst-case service to bound starting times and finishing times. The worst-case provided service depends on the choice of arbiter, typically resulting in that only a single arbiter is supported by the resource to simplify analysis.

This section introduces the concept of $\mathcal{LR}$ [118] servers as a shared-resource abstraction. $\mathcal{LR}$ servers is a general frame-work for analyzing scheduling algorithms that characterizes the service provided by an arbiter with a lower linear bound that depends on two parameters, being service latency and allocated rate, respectively. The service provided to a requestor by any arbiter belonging to the class of $\mathcal{LR}$ can hence be described by the values of these two parameters. Deriving a bound on starting times and finishing times based on these parameters hence ensures that the bounds are valid for any arbiter belonging to the class, such as Weighted Round-Robin [66], Deficit Round-Robin [111], TDM [90], and several varieties of Fair Queuing [140]. This $\mathcal{LR}$ server abstraction is hence useful to capture the diversity of arbiters found in contemporary systems.

A benefit of the $\mathcal{LR}$ server abstraction is that it supports formal performance analysis using approaches based on network calculus [28] or data-flow analysis [113]. Both of these frameworks provide analysis tools that enable formal verification of real-time requirements and buffer sizing in systems comprising multiple resources shared by arbiters in the class of $\mathcal{LR}$ servers. The mathematical formalism of $\mathcal{LR}$ servers is designed to fit with the concept of service curves that are used to characterize applications in verification approaches based on network calculus, such as [55, 126, 128]. It also fits with data-flow analysis by using the data-flow component proposed in [134] that models the behavior of a $\mathcal{LR}$ server. This component enables the shared resource to be included in a data-flow graph that represents both the task graph of the application and the platform resources it uses in a single framework, as done in [49, 121].

We start by defining a $\mathcal{LR}$ server and its associated service guarantee in our formal model. We then use the service guarantee to derive bounds on the starting times and finishing times of the requests relative to the arrival time. We use the definitions from [118], adapted to fit with our use of discrete, as opposed to continuous, time. The concept of *busy periods*, defined in Definition 5.16 is central to the definition of $\mathcal{LR}$ servers. A busy period is intuitively understood as a period in which a requestor requests more service on average than it is allocated. Definition 5.17 defines a $\mathcal{LR}$ server as a server that guarantees a busy requestor its allocated service rate, ρ', after a maximum service latency, Θ. This guarantee results in a lower bound on provided service, $\check{w}'$, as illustrated in Fig. 5.3. The requestor in the figure is busy from τ_1 until τ_2, since it is above the dash-dotted reference line with slope ρ' that we informally refer to as the *busy line*. A second busy period starts at τ_3 and lasts throughout the rest of the shown interval.

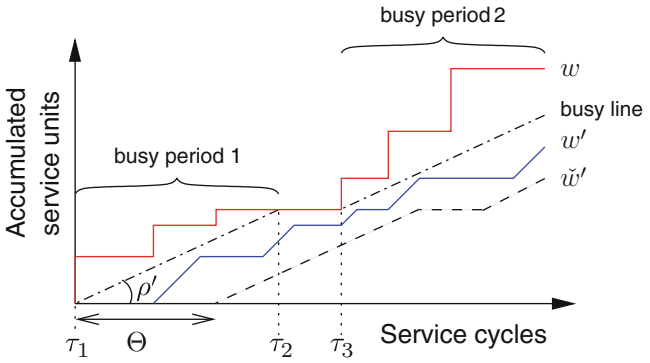

Fig. 5.3 Example service curves in a $\mathcal{LR}$ server

Definition 5.16 (Busy period). A busy period of a requestor $r \in R$ is defined as a maximum interval $[\tau_1, \tau_2]$, such that $\forall t \in [\tau_1, \tau_2] : \; w_r(\tau_1 - 1, t-1) \geq \rho'_r \cdot (t - \tau_1 + 1)$. Requestor r is busy $\forall t \in [\tau_1, \tau_2]$.

Definition 5.17 ($\mathcal{LR}$ server). A server is a $\mathcal{LR}$ server if and only if a non-negative service latency Θ_r can be found such that (5.1) holds during a busy period $[\tau_1, \tau_2]$ of a requestor r. The minimum non-negative constant Θ_r satisfying the equation is the service latency of the server.

$$\forall t \in [\tau_1, \tau_2] : \; \check{w}'_r(\tau_1, t) = \max(0, \rho'_r \cdot (t - \tau_1 + 1 - \Theta_r)) \tag{5.1}$$

We bound the starting times and finishing times using the $\mathcal{LR}$ server abstraction, which enables our solution to work with any arbiter belonging to the class. From the work in [134], we derive that the worst-case starting time of a request is expressed according to (5.2). We see that it is determined by the service latency of the arbiter, Θ, or by the worst-case finishing time of the previous request from the requestor, whichever is larger. The first case happens if the arrival of the request triggers a new busy period, and the second case if the requestor is already busy. This can be observed in Fig. 5.4, where the arrival of ω^k triggers the start of a new busy period and hence $\hat{t_s}(\omega^k) = t_a(\omega^k) + \Theta$. On the other hand, ω^{k+1} arrives during a busy period, resulting in $\hat{t_s}(\omega^{k+1}) = \hat{t_f}(\omega^k)$.

$$\hat{t_s}(\omega_r^k) = \max(t_a(\omega_r^k) + \Theta_r, \hat{t_f}(\omega_r^{k-1})) \tag{5.2}$$

Next, Definition 5.18 defines the time it takes for a request that is scheduled at the worst-case starting time to finish receiving service as the completion latency of the request. The bound on completion latency stated in the definition follows immediately from the service guarantee provided by a $\mathcal{LR}$ server. The graphical interpretation of completion latency is also shown in Fig. 5.4.

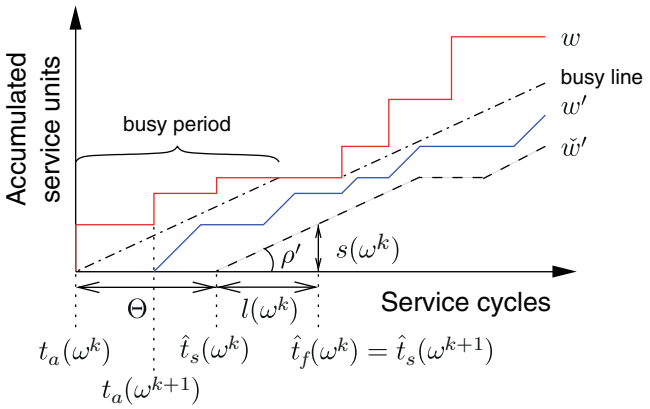

Fig. 5.4 Illustration of worst-case starting time and finishing time in a $\mathcal{LR}$ server

Definition 5.18 (Completion latency). The completion latency of a request ω_r^k from a requestor $r \in R$ is given by $l(\omega_r^k) : \Omega_r \to \mathbb{N}^+$, and is defined according to $l(\omega_r^k) = \hat{t}_f(\omega_r^k) - \hat{t_s}(\omega_r^k)$, which is equal to $s(\omega_r^k)/\rho_r'$.

We have now defined everything we need to compute an upper bound on the finishing time of a request in a $\mathcal{LR}$ server. It follows directly from Definition 5.18 that the worst-case finishing time is computed according to (5.3), which is equivalent to the result presented in [134]. This equation is visualized for request ω^k in Fig. 5.4.

$$\hat{t}_f(\omega_r^k) = \hat{t_s}(\omega_r^k) + l(\omega_r^k) \tag{5.3}$$

5.4 Time-Division Multiplexing

We have now bounded the starting time and finishing time of a request relative to its arrival time using the service guarantee of $\mathcal{LR}$ servers. The derived bound is general and applies to any arbiter belonging to the class. It furthermore shows that all arbiters in the class of $\mathcal{LR}$ servers are predictable, since it bounds the number of service cycles before a request is scheduled and finished. In the following sections, we examine three $\mathcal{LR}$ arbiters in more detail to determine the types of requirements they can satisfy. All considered arbiters have a small and fast hardware implementation, but different abilities to satisfy bandwidth and latency requirements. This section starts by discussing the TDM arbiter, since it is a commonly used arbiter that is easy to conceptually understand and analyze. We start with an overview of the arbiter and then proceed by analyzing it to derive the service latency and over-allocated rate.

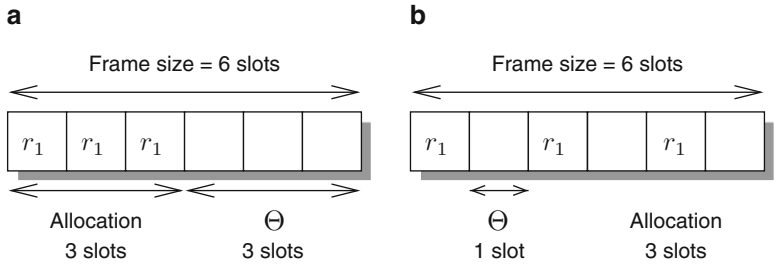

Fig. 5.5 Centralized and distributed slot assignment strategies for TDM. (**a**) Centralized assignment strategy. (**b**) Distributed assignment strategy

5.4.1 Overview

The TDM rate regulator is based on a repeating statically computed schedule. The schedule comprises a number of *slots*, each corresponding to a resource access of one service unit. The size of the schedule is determined by the number of slots and is referred as the *frame size*, *f*, of the arbiter. Each slot in the schedule is either empty or statically assigned to a particular requestor. The allocated rate of a requestor $r \in R$ is hence determined by the number of slots, ϕ_r, the requestor is allocated in the schedule.

The scheduler of a TDM arbiter is simple and static. A scheduling decision is made by examining the next slot in the schedule and providing resource access to the requestor assigned to that slot. The behaviors of work-conserving and non-work-conserving instances of the scheduler are the same, except for how they manage empty slots. A work-conserving instance that encounters an empty slot in the schedule schedules any requestor with a pending request during an empty slot. In contrast, a non-work-conserving instance does not schedule a requestor, but idles during the service cycle of the empty slot.

For a TDM arbiter, not only the number of slots is statically assigned to a requestor at design time, but also their locations in the schedule. This assignment can be done according to different strategies and the choice of strategy impacts the service latency provided by the arbiter. We illustrate this by discussing two slot assignment strategies [90], called *continuous* and *distributed* slot assignment, respectively. The continuous strategy is very simple as it clusters the slots allocated to a particular requestor next to each other in the schedule, starting from the first empty slot. This is illustrated in Fig. 5.5a. The main benefit of this strategy is that it is very easy to implement, but it may result in long service latencies, as we will show in Sect. 5.7.3. In contrast, the distributed assignment strategy prevents clustering by trying to place slots equidistantly in the schedule, as shown in Fig. 5.5b. This strategy performs better with respect to latency, as we will show later in this section, but is more difficult to implement. The main challenge with this strategy is that it is often not possible to achieve equidistant spacing between slots assigned to a requestor.

There are two main reasons for this. Firstly, the number of allocated slots may not be evenly divisible by the frame size. This inevitably results in that some slots are placed closer together in the schedule than others. Secondly, in the unlikely event that the number of allocated slots for all requestors is evenly divisible by the frame size, a distributed allocation may require that the same slot is assigned to multiple requestors, causing all but one of them to be scheduled either earlier or later.

5.4.2 Analysis

This section analyzes the TDM arbiter. First, we study the allocation properties of the rate regulator before computing the service latency for the two presented slot assignment strategies.

The number of slots allocated to a requestor is determined according to (5.4). This ensures that a requestor gets the minimum number of slots that satisfies its intended allocated rate, minimizing over-allocation. The discrete allocated rate of a requestor is determined by computing the fraction of the total service units the requestor may use. For TDM, this is done by dividing the number of slots allocated to a requestor within a frame with the frame size, as shown in (5.5). It is now possible to determine the over-allocated rate of a requestor by subtracting the discrete rate allocated by the rate regulator from the intended allocated rate, as previously defined in Definition 5.10. This is done for the TDM rate regulator in (5.6). The equation shows that the maximum over-allocation of a requestor is inversely proportional to the frame size, implying that a large frame size is required to provide an efficient allocation.

$$\phi_{r_i} = \lceil \rho'_{r_i} \cdot f \rceil \tag{5.4}$$

$$\rho''_{r_i} = \frac{\phi_{r_i}}{f} \tag{5.5}$$

$$o_{\rho}^{tdm}(\rho''_{r_i}, \rho'_{r_i}) = \rho''_{r_i} - \rho'_{r_i} = \frac{\lceil \rho'_{r_i} \cdot f \rceil}{f} - \rho'_{r_i} < \frac{1}{f} \tag{5.6}$$

The service latency of a TDM arbiter depends on the slot assignment strategy. Equation (5.7) shows how to compute the service latency for the continuous slot assignment strategy, explained above. In the worst-case situation, a request arrives and starts a busy period just after the cluster of ϕ_{r_i} slots assigned to the requestor in the schedule. This forces the requestor to wait $f - \phi_{r_i}$ service cycles while the arbiter is serving other requestors. The last expression in the equation uses the relation between the parameters stated in (5.5) to show that the service latency of TDM using the continuous assignment strategy is determined by the allocated rate of the requestor and the frame size of the arbiter.

$$\Theta_{r_i}^{continuous} = f - \phi_{r_i} = f - \lceil \rho'_{r_i} \cdot f \rceil \leq f - f \cdot \rho'_{r_i} = f \cdot (1 - \rho'_{r_i}) \tag{5.7}$$

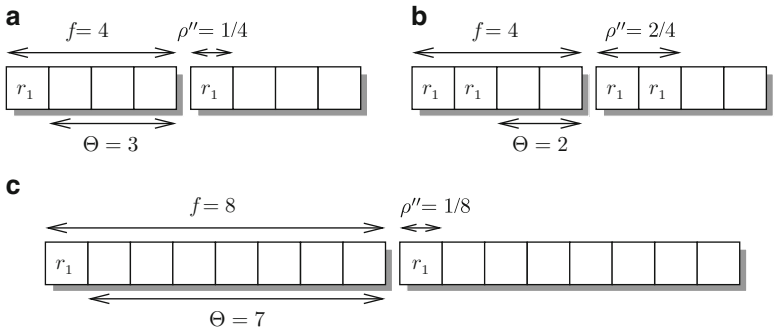

Fig. 5.6 Example of coupling between allocation granularity, latency, and allocated bandwidth. (**a**) Initial case. (**b**) Increasing allocated rate reduces service latency. (**c**) Increasing frame size reduces over-allocation, but increases service latency

The service latency of the distributed slot assignment strategy is shown in (5.8). For simplicity, this section assumes an ideal distributed assignment where all slots allocated to a requestor are placed equidistantly in the frame. A method for computing the service latency in the more complex general case is discussed in [90]. The worst-case situation with the distributed assignment strategy is that a request arrives and starts a new busy period just after a slot belonging to its requestor and has to wait for the next allocated slot. The distance between slots is f/ϕ_{r_i}, resulting in a service latency of $f/\phi_{r_i} - 1$ service cycles. The last expression in the equation shows that the service latency of this strategy depends on the allocated rate of the requestor, but not on the frame size. This is a great advantage of the distributed assignment strategy, as we will later explain. Note that this section only analyzes two assignment strategies out of the many possibilities. However, the ideal distributed assignment and the centralized assignment are the two extreme cases in terms of service latency and all other possible assignments are bounded by these [90].

$$\Theta_{r_i}^{distributed} = \frac{f}{\phi_{r_i}} - 1 = \frac{f}{\lceil \rho'_{r_i} \cdot f \rceil} - 1 \leq \frac{f}{\rho'_{r_i} \cdot f} - 1 = \frac{1}{\rho'_{r_i}} - 1 \tag{5.8}$$

A common problem for many arbiters is that they have unwanted couplings between essential properties, such as allocation granularity, latency, and rate. These problems are particularly common to arbiters with frame-based rate regulators, such as TDM. We illustrate this problem with an example in Fig. 5.6. Figure 5.6a shows an example where the rate regulator has a frame size of four slots. A requestor is allocated $\rho' = 0.25$ and is provided with one out of four slots in the frame. This results in a discrete allocation $\rho'' = 1/4 = 0.25$, as shown in (5.5). There is

hence no over-allocation, according to Definition 5.10. Since the requestor is only assigned a single slot, the choice of assignment strategy has no impact on latency. The figure shows that the requestor is assigned the first slot in the frame. The worst-case latency hence happens if a request arrives in the second slot. In this case, the requestor has to wait three service cycles before being guaranteed its allocated fraction of the resource, which is consistent with both (5.7) and (5.8). Note that there is nothing we can do to reduce latency, except increasing the allocated rate of the requestor, wasting bandwidth. This is shown in Fig. 5.6b, where the requestor is allocated an additional slot in the frame, reducing the worst-case latency to two slots with a continuous assignment. However, this latency reduction comes at expense of allocating another 25% of the total bandwidth. The example illustrates that *latency and rate are coupled*, and that one is traded for the other.

The coupling between allocation granularity and latency becomes a problem if the requestor is only allocated 10% of the total bandwidth. It still gets one out of four slots and hence 25% of the total bandwidth by (5.4), since it is not possible to allocate fractional slots. However, this means that we are wasting 15% of the actually provided 25%, due to discretization effects. We can address this problem by doubling the frame size, changing the discrete allocation of the requestor to $\rho'' = 1/8 = 12.5\%$ of the bandwidth. This reduces the over-allocation from 15 to 2.5% of the bandwidth allocated to the requestor. However, as seen in Fig. 5.6c, this means that the service latency also increases from three slots to seven. The coupling between allocation granularity and latency hence implies a trade-off between over-allocation and latency. This is visible in our earlier analysis, since a large frame reduces over-allocation by (5.6) and increases latency for the continuous assignment strategy by (5.7). The latter problem does not exist for ideal distributed slot assignments, making this the preferred assignment strategy.

We proceed by discussing some important properties of the TDM arbiter. Equations (5.5), (5.7), and (5.8) show how to compute the discrete allocated rate as well as the service latencies for TDM for both the continuous and the distributed assignment strategies. This indicates that TDM belongs to the class of $\mathcal{LR}$ servers, making it a predictable arbiter. This holds for both work-conserving and non-work-conserving instances of the arbiter, since the service latency is bounded for non-work-conserving instances and a work-conserving instance cannot result in longer service latency. Non-work-conserving instances of TDM are furthermore composable, since the absence or presence of one requestor cannot affect the behavior of another. However, work-conserving instances of TDM are not composable, since the absence of requests from one requestor may cause another requestor to be scheduled earlier, altering its starting time and hence also its finishing time and possibly arrival times of later requests. In the rest of this work, we assume that all instances of TDM are non-work-conserving.

5.5 Frame-Based Static-Priority Arbitration

The discussion on the TDM arbiter concluded that it is a predictable and composable arbiter, albeit with couplings between allocation granularity, latency, and rate. A common way to resolve the coupling between latency and rate is to use priorities. However, a regular static-priority arbiter is unpredictable, since it is possible for a requestor to starve everyone with lower priority by continuously requesting service. Priority-based arbiters hence require rate regulators to ensure predictable resource access for all requestors. This section discusses a Frame-Based Static-Priority (FBSP) arbiter, which is designed according to this principle. Just like for TDM, we start with an overview of the arbiter before analyzing its allocation behavior and service latency.

5.5.1 Overview

The FBSP arbiter comprises a frame-based rate regulator and a static-priority arbiter. The FBSP rate regulator is the same as in TDM with a common repeating frame with size f in which each requestor is allocated a number of slots, ϕ_r. The budgets of the requestors are reset to the number of allocated slots at the end of each frame, making the frame size the *replenishment interval* of the requestors. Left-over budget is not preserved between frames to prevent high-priority requestors from building up large budgets when they are idle and later starve low-priority requestors.

Unlike TDM, the assignment of slots within the frame is *dynamically determined* by the static-priority scheduler. The scheduler continuously selects the highest priority requestor that is eligible, i.e. that has a waiting request and has not used more than its allocated slots within the frame. If there is no eligible requestor, a work-conserving instance of the arbiter schedules any requestor with waiting requests without reducing its budget. Just like for TDM, a non-work-conserving instance of the arbiter idles and does not schedule a requestor if they are all out of budget.

5.5.2 Analysis

Since the FBSP rate regulator is the same as for TDM, it follows that the number of allocated slots and the discrete allocated rate are determined by (5.4) and (5.5), respectively. This implies that the maximum over-allocation of a requestor is computed according to (5.6) and is inversely proportional to the frame size.

The worst-case for a requestor r_i under FBSP arbitration is if a request arrives starting a busy period at the same time as all requestors in the set of higher priority requestors, $R_{r_i}^+$. If this happens $\sum_{\forall r_j \in R_{r_i}^+} \phi_{r_j}$ slots from the end of the frame, the higher priority requestors interfere maximally just before the frame repeats,

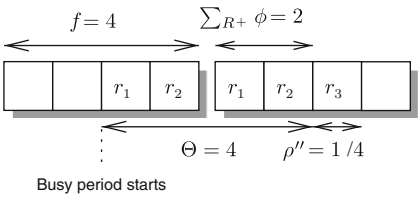

Fig. 5.7 Worst-case situation under FBSP arbitration

replenishing their budgets and enabling them to interfere maximally again before r_i gets to access the resource. The service latency hence considers two times the maximum interference from higher priority requestors. This is expressed in (5.9), where $|R^+_{r_i}|$ is the number of requestors with higher priority than r_i. The worst-case situation is illustrated in Fig. 5.7, where r_3 and its higher priority requestors r_1 and r_2 start their busy period in the third service cycle of the frame. The higher priority requestors have a total allocation of two slots and interfere two service units in the current frame and two additional slots in the following frame, before requestor r_3 is scheduled.

Comparing to the latency equations of the continuous assignment strategy for TDM, previously presented in (5.7), we make two interesting observations. Firstly, that the introduction of priorities results in that a requestor only waits for higher priority requestors, as opposed to all other requestors. Secondly, that the dynamism that enables a requestor to get different slots in different frames, causes the latency to increase by a factor two. This implies that although high-priority requestors enjoy lower service latencies with FBSP than TDM, since they do not have to wait for all other requestors, low-priority requestors are likely to receive longer latencies. This suggests that the choice of arbiter has to reflect the service latency requirements of the requestors in the requestors in the resource.

$$\Theta^{fbsp}_{r_i} = 2 \cdot \sum_{\forall r_j \in R^+_{r_i}} \phi_{r_j} = 2 \cdot \sum_{\forall r_j \in R^+_{r_i}} \lceil f \cdot \rho'_{r_j} \rceil < |R^+_{r_i}| + 2 \cdot f \cdot \sum_{\forall r_j \in R^+_{r_i}} \rho'_{r_j} \tag{5.9}$$

Equation (5.9) reveals two interesting properties with respect to couplings in the arbiter. (1) The service latency of FBSP is proportional to the frame size, while the maximum over-allocation in (5.6) is inversely proportional. This implies that FBSP couples allocation granularity and latency, just like TDM. (2) Latency and rate are decoupled, since increasing priority results in monotonically decreasing latency. Unlike TDM, this enables FBSP to provide low latency to sensitive requestors without wasting resources.

Lastly, we discuss the FBSP arbiter from the perspective of predictability and composability. FBSP belongs to the class of $\mathcal{LR}$ servers and is hence predictable. However, neither work-conserving instances, nor non-work-conserving ones, are composable, since the absence or presence of a request from a requestor affects the starting times and finishing times of requests from lower priority requestors.

5.6 Credit-Controlled Static-Priority Arbitration

The FBSP arbiter demonstrated that priorities are an effective manner of decoupling latency and rate. However, it still couples allocation granularity and latency through the use of a frame-based rate regulator. This section presents a Credit-Controlled Static-Priority (CCSP) arbiter that addresses this issue by combining a more refined rate regulator with the static-priority scheduler. First, we present an overview of the arbiter. We then examine the rate regulator in more detail, before concluding with an analysis of allocation properties and service latency.

5.6.1 Overview

The distinguishing feature of the CCSP rate regulator is that it does not base budget replenishment on a common frame. Instead, it uses a *continuous replenishment* strategy that gives every requestor their allocated fraction of a service unit, ρ', every service cycle. In terms of frame-based regulators, this can be understood as having a frame size of one and allocate fractional slots to the requestors. However, budgets are carried between frames to enable requestors to accumulate sufficient budget for complete slots, allowing them to be scheduled. As previously mentioned in Sect. 5.5, carrying budgets between frames comes with the problem that high-priority requestors can be idle for a long time and accumulate a large budget and then starve lower priority requestors. This is prevented by only replenishing requestors while they are *active*, which intuitively means that they are either backlogged or have recently been asking for service. This restriction bounds the maximum budget a requestor can accumulate, making the arbiter predictable.

The allocated service in a CCSP arbiter consists of two parameters. In addition to the allocated rate used by both TDM and FBSP, a requestor is associated with an *allocated burstiness*, σ'. The allocated burstiness determines the initial budget of a requestor when it starts an active period, while the allocated rate decides the speed with which the budget is replenished during an active period. Together these two parameters determine the upper bound on provided service of a requestor, $\hat{w}'$, as illustrated in Fig. 5.8. A high allocated burstiness entitles a requestor to more service before exhausting its budget, forcing it to surrender the resource to lower priority requestors. The ability to allocate burstiness and service rate separately with CCSP is an important differentiating feature from the frame-based rate regulators of TDM and FBSP that only have a single allocation parameter. The burstiness of these arbiters follows implicitly from the number of allocated slots, which are determined by the allocated rate and the frame size, as previously stated in (5.4). Rate and burstiness are hence coupled in these arbiters, invariably resulting in massive interference from high-priority requestors with high rate requirements.

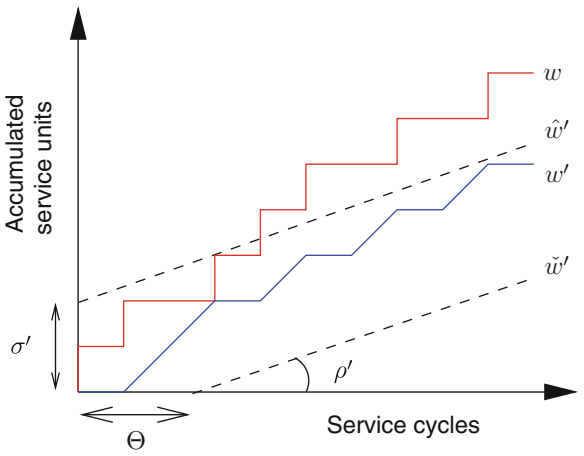

Fig. 5.8 The upper bound on provided service, $\hat{w}'$, in a CCSP arbiter is determined by the allocated rate and the allocated burstiness

5.6.2 *Active Period Rate Regulation*

This section presents the CCSP rate regulator in more detail, which is required to enable analysis of its allocation properties and service latency. We start by formally defining the allocated burstiness of a requestor, discussed in the previous section, in Definition 5.19. Note that the definition states that the allocated burstiness must be at least equal to one service unit. This ensures that a requestor starting an active period is immediately eligible for scheduling, a requirement for the service latency of the arbiter to be valid.

Definition 5.19 (Allocated burstiness). The allocated burstiness of a requestor $r \in R$ is denoted $\sigma'_r \in \mathbb{R}^+$. For a valid allocation it holds that $\forall r \in R : \sigma'_r \geq 1$.

As previously mentioned, CCSP regulates provided service based on the notion of active periods to prevent high-priority requestors from starving low-priority requestors after a long period of idleness. Definition 5.20 states that a requestor is active at t if it is either live at t (Definition 5.21), backlogged at t, or both. Definition 5.21 states that a requestor must on average have requested service according to its allocated rate since the start of the active period to be considered live at a time t. We denote the set of requestors that are active at t with R^a_t.

Definition 5.20 (Active period). An active period of a requestor $r \in R$ is defined as the maximum interval $[\tau_1, \tau_2]$, such that $\forall t \in [\tau_1, \tau_2] : w_r(\tau_1 - 1, t-1) \geq \rho'_r \cdot (t - \tau_1 + 1) \vee q_r(t) > 0$. Requestor r is active $\forall t \in [\tau_1, \tau_2]$.

Definition 5.21 (Live requestor). A requestor $r \in R$ is defined as live at a time t during an active period $[\tau_1, \tau_2]$ if $w_r(\tau_1 - 1, t-1) \geq \rho'_r \cdot (t - \tau_1 + 1)$.

Figure 5.9 illustrates the relation between being live, backlogged and active. Three requests arrive starting from τ_1, keeping the requestor live until τ_3. The requestor is initially both live and backlogged, but the provided service curve catches

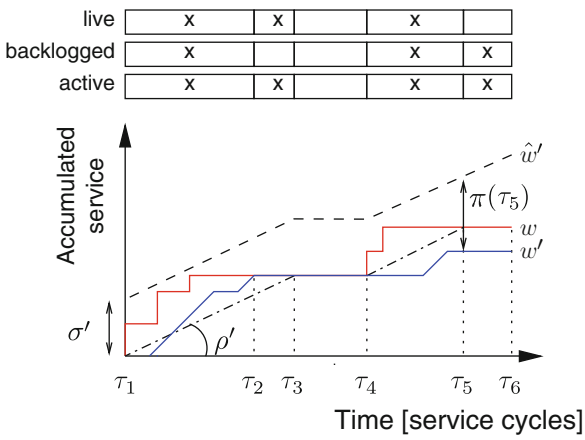

Fig. 5.9 Illustration of the relation between being live, backlogged, and active

up with the requested service curve at τ_2. This puts the requestor in a live and not backlogged state until τ_3. The requestor is neither live nor backlogged between τ_3 and τ_4, as no additional requests arrive at the resource. The requestor becomes live and backlogged again at τ_4, since two additional requests arrive within a small period of time. The requestor stays in this state until τ_5, since not enough service is provided to remove the backlog. The requestor is hence backlogged, but not live at τ_5, and remains such until the end of the shown interval. The requestor in Fig. 5.9 is active between τ_1 and τ_3 and from τ_4 and onwards, according to Definition 5.20. Note from this example that a live requestor is not necessarily backlogged, nor vice versa.

Active periods are related to the busy periods used to describe the service in a $\mathcal{LR}$ server, previously defined in Definition 5.16. However, the relation is complex and is not discussed further in this work. The interested reader is referred to [6]. The CCSP implementation is based on active periods as opposed to busy periods, as this simplifies the implementation of the arbiter. The problem with rate regulation based on busy periods is that it is not possible to determine whether or not a requestor is busy without observing all requests entering the Request Buffer. In contrast, the CCSP arbiter determines if a requestor is active or not by simply observing its internal state and if there is a request pending at the head of the Request Buffer [11]. The CCSP arbiter is hence only aware of the front of the Request Buffer, as opposed to both the front and the back. This provides a better fit with the general context previously shown in Fig. 1.7 and allows the arbiter to use the same physical interface as both TDM and FBSP.

The enforced upper bound on provided service, $\hat{w}'$, is defined according to Definition 5.22. The intuition behind the definition is that the bound of an active requestor increases according to the allocated rate every service cycle, as shown in Fig. 5.9. Conversely, for an inactive requestor, the bound is limited to $w'(t) + \sigma'$, a value that depends on the allocated burstiness. This prevents a requestor that has been inactive for an extended period of time from increasing its bound, possibly resulting in starvation of other requestors once it becomes active again.

Definition 5.22 (Provided service bound). The enforced upper bound on provided service of a requestor $r \in R$ is given by $\hat{w}'_r(t) : \mathbb{N} \to \mathbb{R}^+$, where $\hat{w}'_r(0) = \sigma'_r$ and

$$\hat{w}'_r(t+1) = \begin{cases} \hat{w}'_r(t) + \rho'_r & r \in R^a_t \\ w'_r(t) + \sigma'_r & r \notin R^a_t \end{cases} \tag{5.10}$$

It is not possible to perform accounting and enforcement in hardware based on $\hat{w}'$, since $\lim_{t\to\infty} \hat{w}'(t) = \infty$, which cannot be represented in an implementation with finite precision. Instead, the accounting is based on the *potential* of a requestor, defined as $\pi_r(t) = \hat{w}'_r(t) - w'_r(t)$. This corresponds to the remaining service units that can be provided to a requestor before it hits its upper bound. The potential of a requestor is bounded since the arbiter guarantees a lower bound on provided service. We arrive at the potential-based accounting mechanism in Definition 5.23 by subtracting $w'_r(t+1)$ from both sides in (5.10) and applying Definition 5.12, as shown in [10]. The graphical interpretation of potential is shown in Fig. 5.9.

Definition 5.23 (Potential-based accounting). The accounted potential of a requestor $r \in R$ is given by $\pi_r(t) : \mathbb{N} \to \mathbb{R}$, where $\pi_r(0) = \sigma'_r$ and

$$\pi_r(t+1) = \begin{cases} \pi_r(t) + \rho'_r - 1 & r \in R^a_t \wedge \gamma(t) = r \\ \pi_r(t) + \rho'_r & r \in R^a_t \wedge \gamma(t) \neq r \\ \sigma'_r & r \notin R^a_t \wedge \gamma(t) \neq r \end{cases}$$

Enforcement in the rate regulator takes place before the accounting is updated in a service cycle, and is performed by determining if a request from a requestor is eligible for scheduling. A request is defined as eligible if the following two conditions are satisfied: (1) the requestor is backlogged, and (2) the requestor has at least enough potential to serve one service unit, including the service earned when the accounting is updated, i.e. $\pi(t) \geq 1 - \rho'$. The eligibility information is used by the static-priority scheduler that schedules the highest priority eligible requestor every service cycle.

The hardware implementation of the rate regulator represents the allocated rate of a requestor as a fraction of integers with a numerator, n, and denominator, d. The precision of the arbiter is defined as the number of bits, β, used to represent these integers in the implementation. This is expressed in (5.11), which determines the discrete allocated rate of a requestor. The integer-based mechanism is also used to represent the allocated burstiness of a requestor, causing it to be discretized just like the allocated rate. The discrete allocated burstiness in CCSP is defined according to Definition 5.24. Similarly to the allocated rate, discretization causes burstiness to be over-allocated, potentially increasing in increased service latency of lower priority requestors. The over-allocated burstiness is defined in Definition 5.25.

$$\rho''_r = \frac{n_r}{d_r}, \text{ where } n_r \leq d_r < 2^\beta \tag{5.11}$$

Definition 5.24 (Discrete allocated burstiness). The discrete allocated burstiness of a requestor $r \in R$ is denoted $\sigma''_r \in \mathbb{Q}^+$, and is defined as $\sigma''_r = \frac{\lceil \sigma'_r \cdot d_r \rceil}{d_r}$.

Definition 5.25 (Over-allocated burstiness). The over-allocated burstiness of a requestor $r \in R$ is denoted $o_\sigma(\sigma''_r, \sigma'_r) : \mathbb{Q}^+ \times \mathbb{R}^+ \to \mathbb{R}$, and is defined according to $o_\sigma(\sigma''_r, \sigma'_r) = \sigma''_r - \sigma'_r$.

The accounting used by the hardware implementation of the rate regulator is presented in Definition 5.26, and the formal proof of correctness is provided in [11]. It is in essence a discrete implementation of the potential-based accounting mechanism in Definition 5.23, based on the integer representation of the allocated service. Note that the accounting is simple and only needs to know the current credit state (discrete potential), $c(t)$, of each requestor, if they are backlogged or not, and which requestor was scheduled in the service cycle when updating the state.

Definition 5.26 (Credit-based accounting). The number of credits of a requestor $r \in R$ is given by $c_r(t) : \mathbb{N} \to \mathbb{N}$, where $c_r(0) = \sigma''_r \cdot d_r$ and

$$c_r(t+1) = \begin{cases} c_r(t) + n_r - d_r & \gamma(t) = r \\ c_r(t) + n_r & \gamma(t) \neq r \wedge q_r(t) > 0 \\ \min(c_r(t) + n_r, c_r(0)) & \gamma(t) \neq r \wedge q_r(t) = 0 \end{cases}$$

5.6.3 Analysis

The allocation properties and service latencies provided by a CCSP arbiter depends on the strategy used to select the n and d parameters [11]. Our strategy is to choose these parameters independently per requestor, such that ρ'' is the minimum rate that satisfies $\rho'' \geq \rho'$. If there are multiple n and d pairs providing equal approximations of the allocated rate (e.g. $\frac{1}{2} = \frac{2}{4}$), the one with the largest d is preferred to improve the approximation of the allocated burstiness. This assignment strategy minimizes the over-allocated rate and the impact of discretization on the service latency. It has been shown in [11] that this results in that the over-allocated rate and burstiness of a requestor in a CCSP arbiter with a precision of β bits is upper bounded according to (5.12) and (5.13), respectively.

$$o_\rho^{ccsp}(\rho''_r, \rho'_r) < \frac{1}{2^\beta - 1} \tag{5.12}$$

$$o_\sigma^{ccsp}(\sigma''_r, \sigma'_r) < \frac{2}{2^\beta - 1} \tag{5.13}$$

It has been shown in [12] that the maximum interference in a CCSP arbiter occurs when all higher priority requestors start their active periods at the same time. The service latency for this case is computed according to (5.14). The intuition behind

this expression can be explained in two steps, corresponding to the numerator and the denominator in the equation, respectively. Firstly the numerator states that all higher priority requestors may use the resource until their burstiness allocations are exhausted. Secondly, the denominator adds that the budgets of the higher priority requestors are continuously replenished while they are exhausting their burstiness allowance, prolonging the time they occupy the resource.

$$\Theta_{r_i} = \frac{\sum_{\forall r_j \in R_{r_i}^+} \sigma''_{r_j}}{1 - \sum_{\forall r_j \in R_{r_i}^+} \rho''_{r_j}} \tag{5.14}$$

We proceed by discussing the couplings of the CCSP arbiter, based on the bounds presented in this section. Equations (5.12) and (5.13) show that the over-allocated rate and burstiness monotonically reduce with increased precision. Unlike TDM and FBSP, increasing precision does not negatively impact the service latency of requestors, since there is no dependence on frame size. In contrast, increasing precision *decreases* the service latencies of the requestors by (5.14), since the discrete allocated rates and burstinesses are reduced. Equation (5.14) furthermore shows that latency is decoupled from rate using priorities and decreases monotonically with increasing priority, just like for FBSP. We hence conclude that the CCSP arbiter does not have any couplings between allocation granularity, latency, and rate. Burstiness is furthermore configured independently from rate and frame size. This enables the time to serve a sequence of requests from a requestor with a given priority level to be traded for latency of lower priority requestors without increasing the allocated rate.

Considering predictability and composability, it is shown in [12] that CCSP belongs to the class of $\mathcal{LR}$ servers, making it a predictable arbiter. However, similarly to FBSP, neither work-conserving instances, nor non-work-conserving ones, are composable due to the dynamism in the scheduler.

5.7 Experimental Results

The time has come to experimentally evaluate the theory presented in this chapter. First, we explain the experimental setup where the three presented arbiters take turns providing access to a shared SDRAM memory. Then, the service guarantee of the shared memory controller is evaluated, both in the presence of well-behaved requestors, and when a malfunctioning requestor is asking for more bandwidth than specified. We then proceed by studying the bounds on service latencies provided by the three arbiters for two different use-cases to get an idea of the kinds of latency requirements they can satisfy. The tightness of the bound on service latency is then evaluated for the CCSP arbiter, both when it is expressed in abstract service cycles and actual clock cycles. Lastly, we experimentally compare the allocation properties of FBSP and CCSP and practically demonstrate the difference between frame-based and continuous replenishment strategies.

5.7.1 Experimental Setup

The experimental setup consists of a SystemC simulation model of the CoMPSoC platform. The processor tiles [82], previously shown in Fig. 1.11, are represented by traffic generators that generate requests according to a normal distribution. The average time between requests is determined by the generated bandwidth and request size, and a variance of 10 ns is used to prevent requests from being issued periodically. The memory controller architecture used in these experiments corresponds to the setup previously shown in Fig. 2.6. A predictable arbiter provides access to the predictable SDRAM back-end, previously presented in Chap. 4. The back-end interfaces to our example 16-bit DDR2-400 memory, using the memory pattern set with $BL = 8$ and $BC = 1$ generated by the bank scheduling algorithm, previously shown in Table 4.3. This pattern set has an access granularity of 64 B and guarantees a minimum gross bandwidth of 660 MB/s. An Atomizer chops arriving requests into atoms, whose size are equal to the access granularity of the memory. An arriving request with a size of 256 B is hence be split up into four requests with size 64 B that arrive back-to-back in the Request Buffer, waiting to be scheduled. The processing elements communicate with the memory through the Æthereal [38, 47] NoC. The network is both predictable and composable and hence provides isolated connections that guarantee a minimum bandwidth and a maximum latency. Arbitration in the network is by means of pipelined TDM, which may add a small amount of jitter to the issued requests before they arrive at the memory controller.

5.7.2 Evaluation of Service Guarantee

The first experiment evaluates the service guarantee provided by the SDRAM back-end when shared by a predictable arbiter, in this case CCSP. Table 5.1 presents a simple use-case with four requestors. Two of the requestors only issue read requests, and the other two only issue write requests. Three of the requestors process rather large quantities of data, and request bandwidth according to $b_r = 210$ MB/s, while 20 MB/s suffices for the last requestor. The requestors have different request sizes, but all requests are aligned and an integer multiple of the access granularity of the memory. Data efficiency is hence 100%, making the provided gross and net bandwidths the same. The requested service rates of the requestors, ρ_r, are determined by dividing the requested net bandwidths with the total net bandwidth provided by the memory. The allocated service rates of all requestors are set equal to the requested service rate, i.e. $\rho'_r = \rho_r$. In total, 98.8% of the net bandwidth is allocated to the requestors, including over-allocation, indicating a high load. For all requestors, $\sigma'_r = 1.0$ service units (su), which is the smallest valid allocation according to Definition 5.19. We return to experiment with this parameter later. The allocated rates, ρ'_r, and the allocated burstinesses, σ'_r, may suffer from

Table 5.1 Requestor configuration and service latency bounds

Requestor	Type	b_r (MB/s)	Size (B)	σ''_r (su)	ρ''_r (su/sc)
r_0	Read	210.0	512	1.0	0.319
r_1	Write	210.0	128	1.0	0.319
r_2	Read	210.0	64	1.0	0.319
r_3	Write	20.0	256	1.0	0.031

Table 5.2 Bandwidth and service latency results

Requestor	p_r	b_r (MB/s)	$\dot{b}_r$ (MB/s)	$\max \Theta_r$ (sc)	Θ_r (sc)
r_0	3	210.0	210.0	5	9
r_1	2	210.0	210.0	2	3
r_2	1	210.0	210.0	1	1
r_3	0	20.0	20.0	0	0

Table 5.3 Bandwidth and service latency results with malfunctioning requestor using a regular static-priority arbiter

Requestor	p_r	b_r (MB/s)	$\dot{b}_r$ (MB/s)	$\max \Theta_r$ (sc)	Θ_r (sc)
r_0	3	210.0	0.0	N/A	9
r_1	2	210.0	173.2	10	3
r_2	1	210.0	210.0	3	1
r_3	0	400.0	323.2	0	0

over-allocation due to discretization. This results in the discrete allocated rates, ρ''_r and the allocated burstinesses, σ''_r, that are used in the experiment. We will not discuss this further for now, but we return to this in later experiments.

The use-case in Table 5.1 is simulated during 100 ms and the actual provided bandwidths, $\dot{b}_r$, and maximum latencies, $\max \Theta_r$, are measured and compared to the requested bandwidths, b_r, and the bounds on service latency, Θ_r. Priority levels, p_r, are assigned in descending order with 0 and 3 being the highest and lowest levels, respectively. The results of this experiment, shown in Table 5.2, indicate that all requestors get their requested net bandwidth and that the maximum measured service latency is less or equal to the computed bound. *This experiment hence suggests that the SDRAM back-end combined with a predictable arbiter delivers on its service guarantee.*

A limitation of our evaluation so far is that all requestors in the use-case are well-behaved and do not ask for more bandwidth than specified. To evaluate the robustness of the service guarantee, we modify the highest priority requestor, r_3, to ask for 400 MB/s instead of 20 MB/s, without changing its resource allocation in the network or the memory controller. Table 5.3 shows what happens when this modified use-case is simulated during 100 ms with a regular static-priority arbiter that does not have a rate regulator. We see that r_3 gets 323.2 MB/s out of the requested 400 MB/s, since there is no rate regulator to enforce the allocated 20 MB/s. The reason that the requestor is not getting its full 400 MB/s is because the network connection acts as a bottleneck, since it is not dimensioned for 400 MB/s. Requestor r_2 gets its requested bandwidth, but the memory cannot supply enough bandwidth to deliver on the requirements of r_1 and r_0. In fact, r_0, is completely

Table 5.4 Requestor specification

	Equal rates	Diverse rates
Requestor	ρ'_r (su/sc)	ρ'_r (su/sc)
r_0	0.2	0.05
r_1	0.2	0.10
r_2	0.2	0.15
r_3	0.2	0.2
r_4	0.2	0.5
Total	1.0	1.0

starved by other requestors and does not receive any bandwidth at all! The results also show that the service latency bounds of all requestors except r_3 are violated in this experiment. We repeated the same experiment with the CCSP arbiter, which features a rate regulator. The results of this experiment are essentially equivalent to the results previously shown in Table 5.2. The only difference is that the bandwidth provided to r_3 increases from 20.0 to 22.3 MB/s. The reason is that the memory offers slightly more bandwidth than suggested by its bound, since the simulation contains fewer read/write switches than the worst case. *From this experiment, we conclude that rate regulation is essential to provide a service guarantee that is reliable also in the presence of misbehaving requestors.*

5.7.3 Latency Distributions

The discussion about different arbiters earlier in this chapter suggested that the choice of arbiter must reflect the bandwidth and latency requirements of the requestors. This second experiment shows the service latency distributions provided by the derived bounds for the TDM, FBSP, and CCSP arbiters, and discusses the types of latency requirements they can satisfy. It is not possible to discuss latencies without considering particular use-cases. For this purpose, we define two use-cases with five requestors, shown in Table 5.4. These particular use-cases are chosen because they have distinct characteristics, allowing us to make our points. In the first use-case, all requestors have equal allocated rates, and in the second use-case, they have diverse rate allocations. Both use-cases have an allocated load of 100% of the resource capacity.

5.7.3.1 Time-Division Multiplexing

First, we examine the service latency distributions of TDM for the two presented assignment strategies. Figure 5.10a shows the results for the use-case with equal allocated rates for different assignment strategies and frame sizes. The figure shows that equal allocated rates result in equal service latencies for all requestors for both the continuous and the distributed allocation strategies. It is also shown that

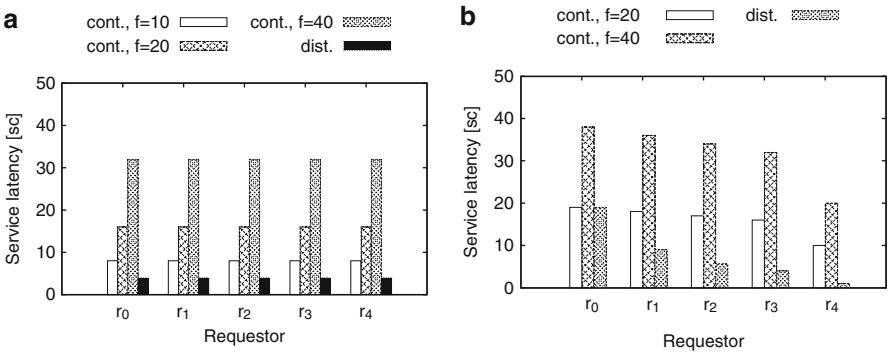

Fig. 5.10 TDM latency distribution for two assignment strategies. (**a**) Use-case with equal allocated rates. (**b**) Use-case with diverse allocated rates

latency reduces with frame size for the continuous assignment strategy, as stated by (5.7), and approaches the latency provided by the distributed assignment strategy. Remember that the continuous and the ideal distributed assignment strategies are two extreme cases and that the results of any other strategy end up in between.

The results for the use-case with diverse allocated rates are shown in Fig. 5.10b. We see that the earlier conclusions about latency changing with frame size and that the distributed slot assignment strategy provides better results remain valid. The main difference with Fig. 5.10a is that the latencies of the requestors are no longer equal, but decreases with increased allocated rate. No results are presented with frame size 10, since the large allocation granularity makes it impossible to allocate sufficient slots to satisfy the rate requirements of all requestors. The allocation granularity is 10% causing the required rates of requestor r_0 and r_2 to be over-allocated with 5% each. This results in a total allocation of 110% of the resource capacity, which is not a valid allocation according to Definition 5.9.

5.7.3.2 Frame-Based Static-Priority Arbitration

We proceed by looking at the latencies provided by the FBSP arbiter with different frame sizes for the use-case with equal allocated rates. The requestors have descending priorities, making r_0 the highest priority requestor and r_4 the lowest. The results, shown in Fig. 5.11, clearly show that the priority levels decouple latencies from rate, as the latencies increase monotonically with reducing priority. The latency increase is linear with decreasing precision, reflecting that all requestors have equal allocated rates. Just like for TDM, the figure also shows that increasing the frame size results in increasing latencies, illustrating that allocation granularity is coupled to latency.

Next, we consider the use-case with diverse allocated rates. Just like for TDM, we only consider frame sizes of 20 and 40, since a frame size of 10 is insufficient

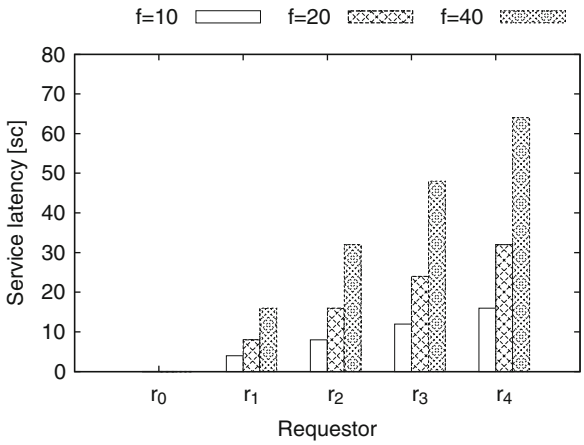

Fig. 5.11 FBSP latency distribution for use-case with equal allocated rates

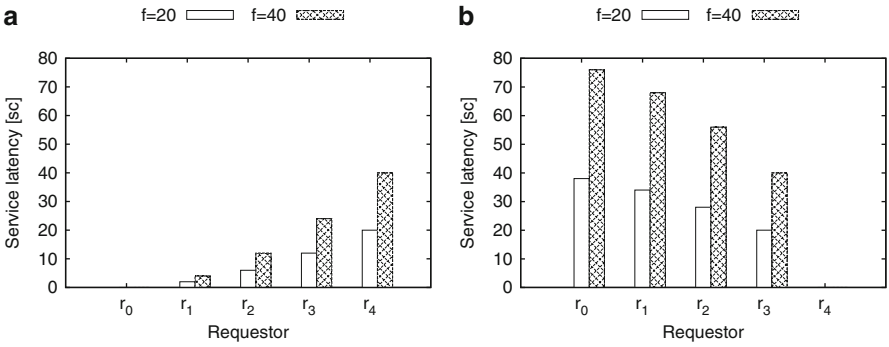

Fig. 5.12 FBSP latency distribution for use-case with diverse allocated rates. (**a**) Descending priorities. (**b**) Ascending priorities

to successfully allocate the use-case. Figure 5.12 presents results with priorities arranged both in ascending and descending order, since the priority assignment heavily impacts the provided latencies for diverse allocated rates. Figure 5.12a shows the latency distribution with descending priorities, which gives high priorities to the requestors with low allocated rates. This priority assignment results in that low latency is provided to several requestors, since the accumulated allocated rates of higher priority requestors remain low for the first couple of requestors. As an example, the three requestors with highest priorities all have latencies less than 15 service cycles with a frame size of 40. Reversing the priority assignment results in that high priorities are given to the requestors with the highest allocated rates. Figure 5.12b shows that this causes the latencies to increase quickly along with the accumulated rate of higher priority requestors. In this case, only the highest priority requestor has a latency less than 15 service cycles. From this observation, we conclude that priority-based arbiters enable low latency to be provided to requestors

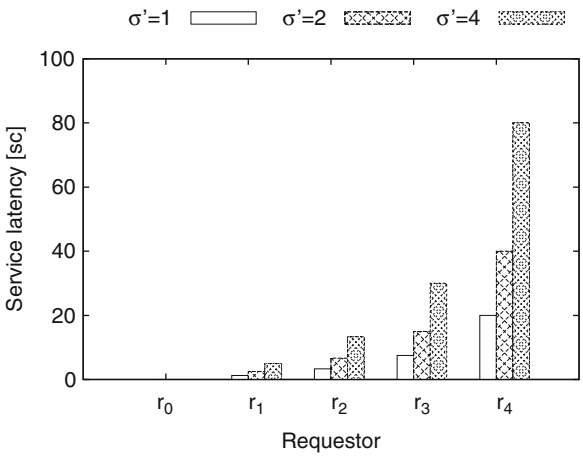

Fig. 5.13 CCSP latency distribution for use-case with equal allocated rates

with low latency requirements, but that priorities have to be carefully assigned to satisfy the latency requirements of a set of requestors. We later discuss how to optimally assign priorities in Sect. 7.4.

5.7.3.3 Credit-Controlled Static-Priority Arbitration

The last arbiter considered in this experiment is a CCSP instance using five bits to represent the n and d parameters of the requestors. The chosen precision is sufficient to reduce both the over-allocated rate and over-allocated burstiness to zero for both use-cases. Note that unlike TDM and FBSP, the precision can be adjusted in this manner without affecting latency, since allocation granularity and latency are decoupled. However, increasing precision increases the area of the arbiter implementation, resulting in a trade-off between over-allocation and area. This trade-off is explored in [11].

The latency distribution for CCSP for the use-case with equal allocated rates and burstinesses is shown in Fig. 5.13. Just like for FBSP, we clearly see that priority levels decouple latencies from rate. The figure also shows that simultaneously increasing the allocated burstiness of all requestors increases the latencies of all requestors except the one with the highest priority. This shows that although the burstiness parameter enables a reduction in the time to serve a sequence of requests without increasing the priority level or the allocated rate, it comes with a latency increase for lower priority requestors.

The results for the use-case with diverse allocated rates with both ascending and descending priorities are shown in Fig. 5.14. The overall trends and conclusions are similar to the results for FBSP, although there are some interesting differences in the provided latencies. The latency distribution for CCSP is different from that of FBSP in that it provides lower latency to high-priority requestors, but it increases faster for low priorities. There are two reasons for this behavior. High-priority requestors have

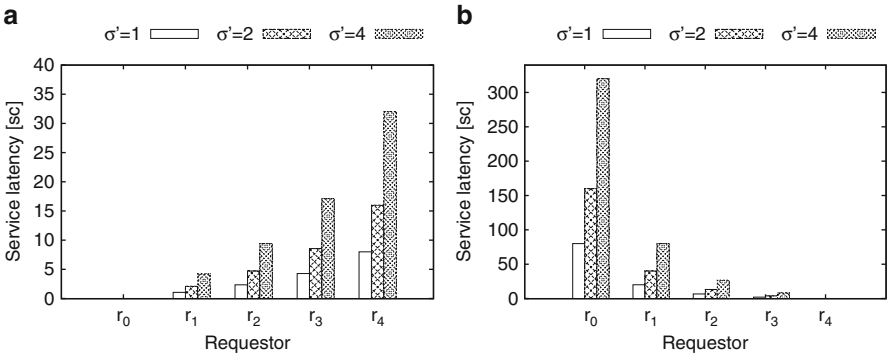

Fig. 5.14 CCSP latency distribution for use-case with diverse allocated rates. (**a**) Descending priorities. (**b**) Ascending priorities

lower latencies because the explicitly allocated burstinesses of requestors in CCSP are typically lower those that are implicitly provided by FBSP for realistic frame sizes. The latencies for low-priority requestors under FBSP increase proportionally to the allocated rates of higher priority requestors, as previously expressed in (5.9). However, the equation is bounded by $2 \cdot (f-1)$, since the lowest priority requestor must have at least one allocated slot in the frame. The concept of frames combined with the restriction that budgets cannot be carried between frames hence bounds the maximum latency that can be provided to a requestor. In contrast, the CCSP rate regulator does not have a notion of frames and continuously replenishes requestors. A high-priority requestor that has exhausted its budget may hence become eligible again several times before a low-priority requestor accesses the resource. This is what causes the latencies provided by (5.14) to increase faster and faster with lower priority levels and approach infinity for fully loaded resources as the allocated rate of the lowest priority requestor approaches zero. Figure 5.14b illustrates this behavior, as the requestors with higher priority than r_0 have allocated a total of 95% of the resource capacity, resulting in a service latency of over 300 service cycles for r_0.

5.7.3.4 Comparison

We conclude this experiment by putting the different latency distributions together for the considered use-cases and priority assignments. For each arbiter, we have chosen the minimum setting that results in a successful allocation. This implies a frame size of 20 for TDM and FBSP, and an allocated burstiness of one for all requestors in CCSP. The results for the use-case with even allocated rates are presented in Fig. 5.15. We see that the arbiters provide quite different latency results for the same use-case and that the choice of arbiter must be dictated by the latency requirements of the requestors. TDM performs very well in this simple use-case, since it is possible to find an ideal distributed assignment. However, a priority-based

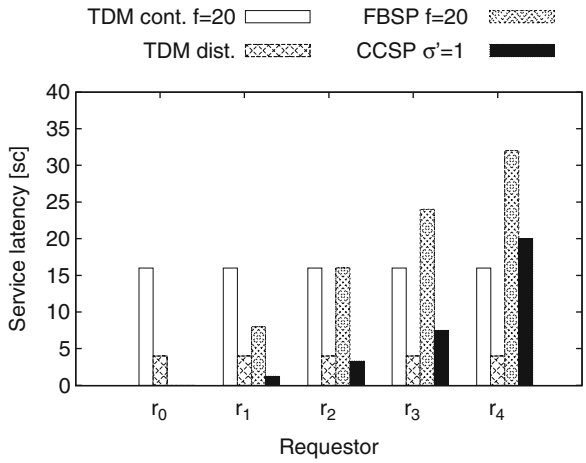

Fig. 5.15 Arbiter latency distribution for use-case with equal allocated rates

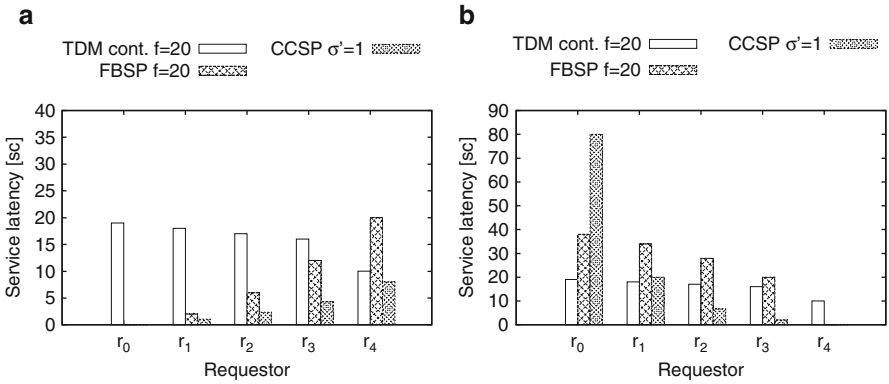

Fig. 5.16 Arbiter latency distribution for use-case with diverse allocated rates. (**a**) Descending priorities. (**b**) Ascending priorities

solution is required in case a requestor has a latency requirement of less than four service cycles, since there is no left-over capacity to allocate in order to reduce the latency with TDM. For this particular use-case, CCSP provides better results than FBSP for all requestors, since the load of higher priority requestors does not accumulate enough for any requestor to make the latency explode.

Results for the use-case with diverse allocated rates are shown in Fig. 5.16. No results are presented for the distributed assignment strategy for TDM, since it is not possible to create an ideal assignment with equidistant spacing between slots for all requestors. This implies that some requestors would have higher latencies than suggested by the bound in (5.8). Figure 5.16a shows the latency distribution with descending priorities. We note that TDM is at a disadvantage due to the inability to make an ideal distributed assignment. FBSP provides lower latency than TDM to

high-priority requestors, but is only a feasible option if the lowest priority requestor can afford a latency increase. Just like in the previous use-case, CCSP provides lower latency than FBSP for all requestors. In this case, it also outperforms TDM, although this would not be the case if a bound would be derived for TDM with the best possible distributed assignment.

Reversing the priorities highlights the differences between CCSP and FBSP. Figure 5.16b shows that CCSP still offers lower latencies to high-priority requestors, but that the latency of low-priority requestors becomes much larger. *From this experiment, we conclude that different arbiters provide different service latency distributions and that it is important that the choice of arbiter and configuration settings, such as frame sizes and priority levels, reflect the requirements of the requestors.*

5.7.4 Tightness of Service Latency Bound

The third experiment evaluates the tightness of the service latency bounds both in abstract service units and in clock cycles. For this experiment, the SDRAM back-end is shared using a CCSP arbiter. We use the use-case in Table 5.1 with descending priorities as a starting point, and uniformly vary the discrete allocated burstinesses, σ''_r, of all requestors in the range [1, 5]. Again, the system is simulated during 100 ms. The maximum measured service latencies and the analytical bounds of the requestors are shown in Fig. 5.17. Three observations are made from the results in the figure. (1) The measured service latency and the bound for r_3 are both zero cycles for all values of σ''. The bound is hence both conservative and perfectly tight for the highest priority requestor. (2) The service latency bound becomes less tight with decreasing priority. There are two main reasons for this behavior. The first reason is that the service latency bound in (5.14) does not take into account that service is provided in a discrete manner. Two requestors providing 1.5 service cycles of interference in an interval hence results in a total interference of 3 service cycles. The actual maximum interference is 2 service cycles, since there is no such thing as half service cycles. The bound is hence over-estimated with up to one service cycle per higher priority requestor, resulting in less tight bounds. This issue is inherent to how the bound is computed. The second reason is that it becomes increasingly unlikely with lower priority that all requestors display their worst-case behavior at the same time. This issue is not related to CCSP, but rather an effect of that all requestors are not constantly backlogged, much like in a realistic use-case. (3) The service latency bound becomes less tight as the allocated burstinesses increase. This happens because the requestors do not ask for service in a bursty enough manner to fully use their allocation. This is understood by realizing that requests are issued almost periodically, although with some jitter from the requestor itself and from the network. The largest request is 256 B, not considering the lowest priority requestor that cannot interfere with anyone. 256 B correspond to 4 service units given the access granularity of the pattern set. Allocating a higher burstiness

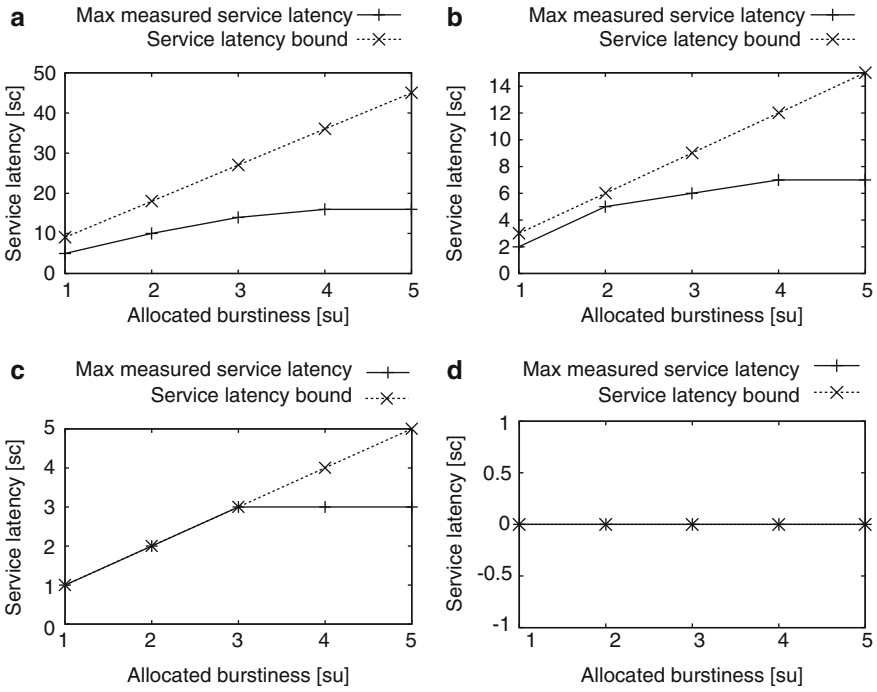

Fig. 5.17 Maximum measured latency and bound, expressed in service cycles, for the requestors in the use-case. (**a**) Maximum measured service latency and bound for r_0. (**b**) Maximum measured service latency and bound for r_1. (**c**) Maximum measured service latency and bound for r_2. (**d**) Maximum measured service latency and bound for r_3

than this to the requestors hence only affects the bound of lower priority requestors, according to (5.14), but not the actual interference. Some of the other requestors have smaller request sizes than 256 B, contributing to less tight bounds of lower priority requestors as the allocated burstiness increases. *Based on this experiment, we conclude that the service latency bound of CCSP is tight for high priority requestors, but becomes less tight with decreasing priority and increasing allocated burstiness.*

This far, we have only reasoned in abstract service cycles. However, we are actually interested in latencies measured in clock cycles. We hence repeat the simulation, but now we use (4.8) to convert the results into clock cycles at 200 MHz. The results are shown in Fig. 5.18. At a first glance, we see the same general trends in tightness as discussed in the previous experiment; the bounds become less tight with decreasing priority and with increasing allocated burstinesses. However, we also note a bigger difference between the maximum measured service latency and the bound. This is especially apparent for r_2 and r_3, whose service latency bounds, measured in service cycles, are relatively small. There are three reasons why the bounds expressed in clock cycles are less tight than the bounds in service cycles.

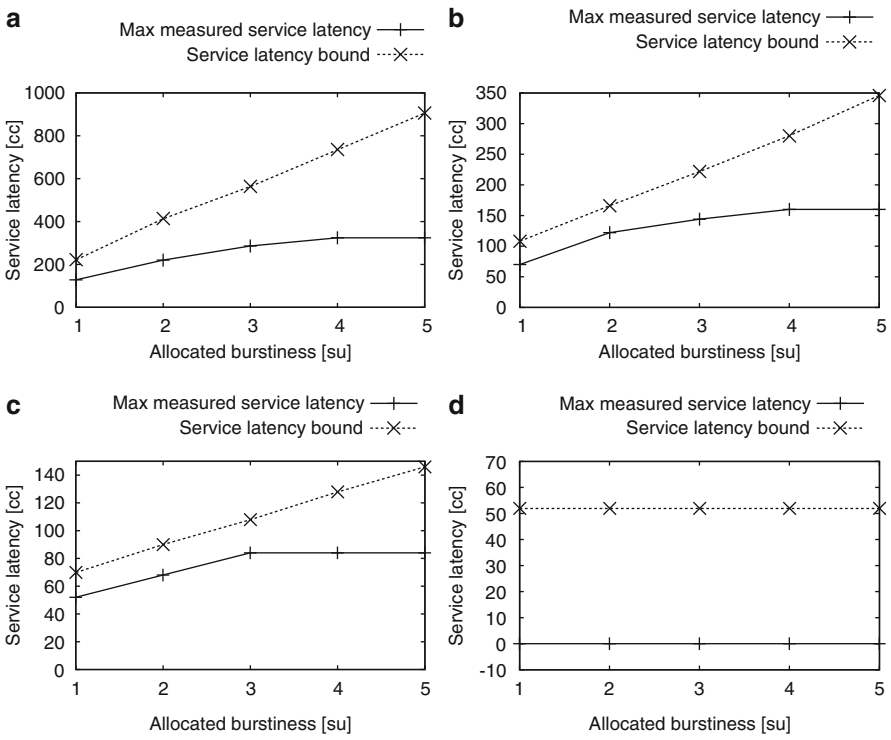

Fig. 5.18 Maximum measured latency and bound, expressed in clock cycles at 200 MHz, for the requestors in the use-case. (**a**) Maximum measured service latency and bound for r_0. (**b**) Maximum measured service latency and bound for r_1. (**c**) Maximum measured service latency and bound for r_2. (**d**) Maximum measured service latency and bound for r_3

(1) The actual simulation may have fewer read/write switches than assumed by the bound. (2) The bound in service cycles, $t_{tot}(x)$ in (4.8), adds an extra service unit to the interference to account for blocking when a request arrives just after a scheduling decision has been taken. However, the actual blocking time may be shorter. Limitations in our instrumentation furthermore prevent us from measuring interference due to blocking. The actual maximum service latencies may hence be up to 20 clock cycles longer with this pattern set. (3) The bound in clock cycles accounts for worst-case interference from refresh, although the actual case may perform better. Our instrumentation captures refresh interference in the general case, but does not include refreshes after the request has been scheduled by the arbiter. It hence does not cover the special case of a requestor that is always scheduled in zero service units, such as r_3 in this use-case. This explains why the service latency bound for r_3 is 52 clock cycles, although the maximum measured value is zero clock cycles! Blocking accounts for 20 clock cycles out of the 52, and refresh for the other 32, neither which can be measured with our instrumentation. Looking past the two

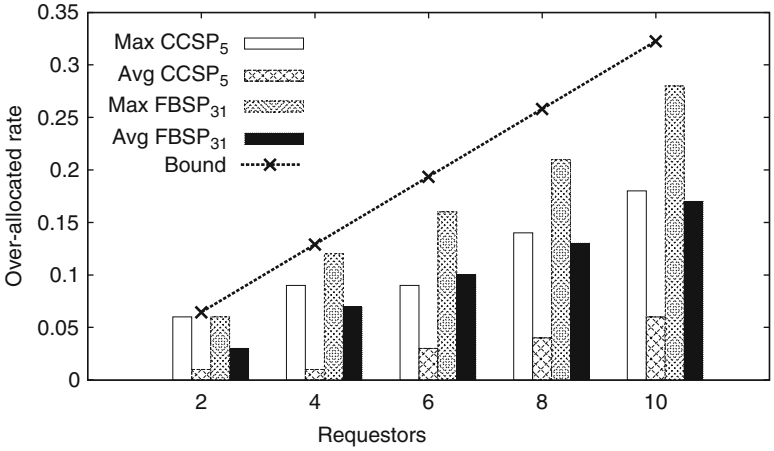

Fig. 5.19 Over-allocated rate for CCSP and FBSP

limitations of our measurements, the results in Fig. 5.18 are very similar to those in Fig. 5.17. *We hence conclude that* (4.8) *performs a useful conversion of bounds in service cycles to clock cycles.*

5.7.5 Allocation Properties

For our final experiment, we compare the allocation properties of CCSP and FBSP. We start by looking at the average and maximum measured over-allocated rates and comparing these to the analytical bounds computed in (5.6) and (5.12). For each number of requestors in 2, 4, 6, and 8, we randomly generate 1000 synthetic use-cases with uniformly distributed loads in the interval [0, 100]%. We are interested in the total over-allocation of all requestors and hence sum their individual over-allocated rates. Similarly, all derived bounds are multiplied with the number of requestors in the use-case to capture the total over-allocation. Five bits of precision ($\beta = 5$) are used for CCSP and a frame size of 31 for FBSP ($f = 31$), since these settings provide the same bounds on over-allocated rate. The results are shown in Fig. 5.19.

We see in the figure that the CCSP results in lower average over-allocated rate than FBSP. In fact, CCSP reduces the average over-allocated rate with a factor of three compared to FBSP. This shows that the allocation mechanism in the rate regulator may have a significant impact on the wasted bandwidth. For example, the average over-allocated rate in the use-cases with six requestors is 3% of the bandwidth for CCSP, while it is 10% with FBSP. The corresponding maximum values are 9 and 16%, respectively, while the bound for both arbiters is 19% with the chosen settings. The reason that CCSP performs better than FBSP is that the

CCSP rate regulator uses two free parameters, n and d, to represent the allocated rate. In contrast, FBSP uses only a single parameter, ϕ, since the frame size is the same for all requestors. The maximum measured over-allocated rate is close to the analytical bound for both arbiters for use-cases with two requestors, although the difference increases with the number of requestors. This reflects that the worst-case over-allocation becomes increasingly unlikely as the number of requestors increases. In particular, we note that the difference between the maximum over-allocation and the bound becomes very large for CCSP. The reason is that the worst-case over-allocation for CCSP only happens if all requestors have allocated rates very close to zero [11], which is very unlikely both for randomly generated and realistic use-cases.

Next, we show the impact of these allocation properties on use-cases with high loads and service latency requirements, which are often found in SDRAM controllers. The use-cases all have six requestors and are randomly generated with the total load divided in a number of bins (91, 93, 95, 97, and 99%, respectively). 1000 use-cases are generated for each bin. The service latency requirements of the requestors are uniformly distributed in the interval [0, 10000] ns. This range is chosen as it provides requirements that are feasible to satisfy with our SDRAM back-end with the considered memory and loads. The requirements are then transformed from ns to service cycles using the inverse of the latency functions of the SDRAM back-end, presented later in Sect. 7.3. This results in requirements varying in the range [0, 120] service cycles. In addition to allocating service for the use-case, priorities are assigned in an attempt to satisfy the service latency requirements of the requestors. For this purpose, we use an optimal priority assignment algorithm, further discussed in Sect. 7.4. We compare the allocation properties of the arbiters by measuring the percentage of use-cases in which the rate requirements of all requestors are satisfied and the total discrete allocated rate is less than 100%, indicating successful allocation. Additionally, we compare the percentage of use-cases where the service latency requirements of all requestors are satisfied. Lastly, we study the total success rate, being the percentage of use-cases where both service allocation and priority assignment are successful, indicating that both rate and latency requirements are satisfied. The results of this experiment are shown in Fig. 5.20.

We note that all use-cases with up to 93% load, and 98.7% of the use-cases with 95% load, are successfully allocated when using CCSP. The success rate is reduced to 88.9 and 53.7% for use-cases with 97 and 99% loads, respectively. As expected, FBSP performs worse, and only allocates 63.7% of the use-cases with 91% load successfully. The success rate is significantly reduced for higher loads and reaches zero for loads higher than 95%. We see that CCSP also performs better when priorities are assigned to satisfy the service latency requirements. The latency requirements are satisfied for 93.3% of the use-cases with 91% load and drops towards 82.8% for use-cases with 99%. FBSP displays a slightly different behavior, starting at 78.3% for 91% load and ending at 75.5% for loads of 99%. The ability to satisfy service latency requirements is hence not as good as for CCSP, but it degrades slower with increasing load. This result is consistent with our earlier

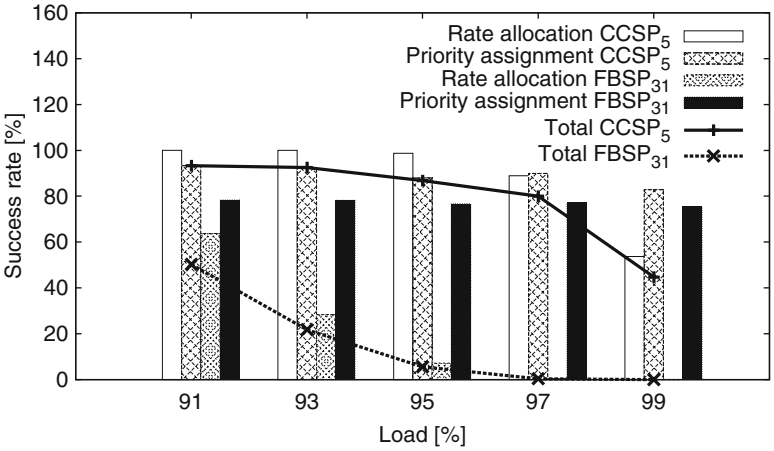

Fig. 5.20 Successful allocations and priority assignments for CCSP and FBSP

observation that latencies provided by the CCSP arbiter increase faster than for FBSP as the accumulated rates of higher priority requestors gets close to 100% of the resource capacity.

The total success rate shows that the CCSP performs better than FBSP for all tested loads, primarily because the smaller over-allocated rate allows more use-cases to be successfully allocated. On average, CCSP results in more than five times as many use-cases with high loads having both their rate and service latency requirements satisfied compared to FBSP. *We conclude from this experiment that having a close approximation of the allocated rate is essential to manage heavily loaded resources.*

We proceed by studying the effects of increasing precision to achieve a finer allocation granularity. Use-cases are randomly generated in the same manner as before, but we now compare CCSP with five and six bits, respectively.

As seen in Fig. 5.21, increasing precision improves both the number of successful allocations and priority assignments. This is because both the over-allocated rate and burstiness of CCSP are monotonically reduced with increased precision, as explained in Sect. 5.6. We experimentally compare this behavior to that of FBSP in Fig. 5.22, where the frame size, f, is increased from 31 to 63. Again, these particular frame sizes are chosen, as they provide the same bounds on over-allocated rate as for CCSP with five and six bits of precision, used in Fig. 5.21. The results show that doubling the frame size to increase precision results in a significant improvement in the percentage of successful allocation for loads up to 95%, all being above 80%. However, this causes the percentage of successful priority assignments to be less than 20% for all loads. This is because both the allocation granularity and the service latency depend on the frame size, causing one to be traded for the other. *We conclude from this experiment that having an allocation granularity that is decoupled from latency is essential when sharing highly loaded resources in the presence of applications with real-time requirements.*

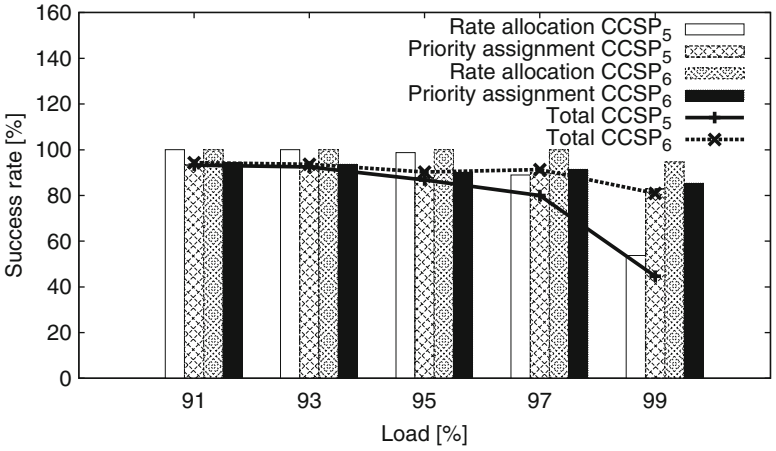

Fig. 5.21 Success rate when increasing precision with CCSP

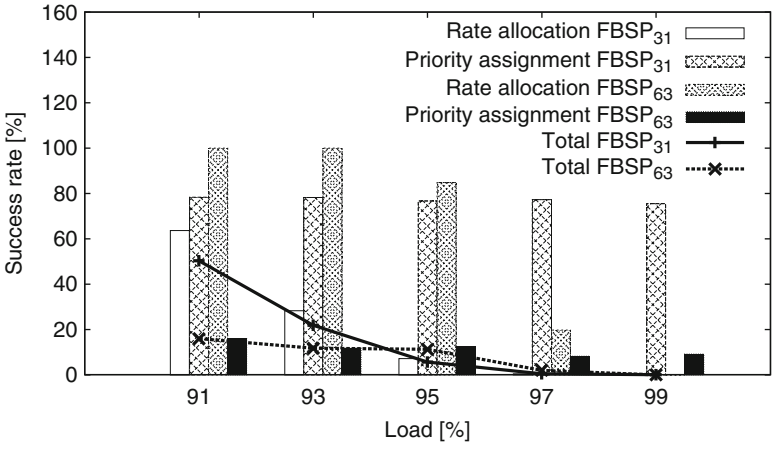

Fig. 5.22 Success rate when increasing precision with FBSP

5.8 Summary

Predictability in our approach is achieved by combining predictable resources with predictable arbitration. The previous chapter addressed the first part by showing how to design a memory controller back-end that makes an SDRAM behave in a predictable manner. This chapter discussed the second part, namely how to share a predictable resource among multiple requestors using a predictable arbiter.

There are three main requirements on the hardware implementation of an arbiter to make it generally applicable in the System-on-Chip (SoC) context. (1) It must run at *high clock frequency* to keep up with the resource and allow scheduling at a fine level of granularity. (2) It must have a *small hardware implementation*. (3) It must

be able to provide the required service to a requestor without *over-allocating*, which means reserving more capacity than required. To fit with the requirements from our application domains it must furthermore be able to *accommodate diverse bandwidth requirements and both latency-sensitive and latency-tolerant requestors*.

We introduced terminology related to resource arbitration and explained how an arbiter can be described in terms of two main parts, being a *rate regulator* and a *scheduler*, respectively. The purpose of a rate regulator is to protect requestors that do not ask for more service than they are allocated from the ones that do. This is done by determining which requests are *eligible* for scheduling at a particular time. It is then the responsibility of the scheduler to choose which eligible requestor to schedule, based on its particular policy.

The Latency-Rate ($\mathcal{LR}$) server model was introduced as a shared-resource abstraction to address the diversity of arbiters present in contemporary SoCs in a transparent manner. A $\mathcal{LR}$ server is an abstraction of predictability that uses two parameters, being a *service latency* and an *allocated rate*, to describe a lower linear bound on the amount of data that is transferred in an interval. It has been shown that many well-known arbiters, such as several varieties of the Round-Robin and Fair Queuing algorithms, are $\mathcal{LR}$ servers. A request is scheduled at its *starting time* and finishes receiving service at its *finishing time*. As a part of satisfying our abstraction requirement, general bounds on these times were derived that are valid for any arbiter in the class of $\mathcal{LR}$ servers.

Three arbiters in the class of $\mathcal{LR}$ servers were presented, Time-Division Multiplexing (TDM), Frame-Based Static-Priority (FBSP), and Credit-Controlled Static-Priority (CCSP). For each arbiter, we described its operation in terms of its rate regulation and scheduling mechanisms and derived their service latencies and allocated rates, respectively. We also discussed their respective abilities to satisfy diverse bandwidth and service latency requirements without over-allocating the resource.

Experimental results showed that the shared predictable memory controller provides its guaranteed service, both in the presence of well-behaved and non-cooperative requestors. The service latency distributions provided by the three presented arbiters were shown for two different use-cases, and we concluded that it is important that the choice of arbiter and configuration settings reflect the bandwidth and latency requirements of the requestors. We experimentally evaluated the tightness of the service latency bound of the CCSP arbiter and concluded that it is tight for high-priority requestors, but becomes less tight with decreasing priority. We also compared the allocation strategies of the CCSP and FBSP arbiters and concluded that CCSP enables fine-grained resource allocation that reduces over-allocation without negatively impacting latency. This makes CCSP a suitable arbiter for highly loaded resources with diverse bandwidth and latency requirements, such as SDRAM controllers.

Chapter 6
Composable Resource Front-End

We have now arrived at a point where we have a predictable SDRAM back-end that offers hard real-time guarantees on net bandwidth and on the time to serve a scheduled memory request. We have also presented three predictable arbiters in the class of Latency-Rate ($\mathcal{LR}$) servers that allow the memory to be shared in a way that bounds the time until a request is scheduled. We have shown that the arbiters are useful for different types of bandwidth and latency requirements, although the Credit-Controlled Static-Priority (CCSP) arbiter distinguishes latency-sensitive and latency-tolerant requestors without wasting scarce bandwidth, fitting with the requirements on SDRAM controllers from Sect. 1.1.6. Together, the combination of predictable resource and predictable arbiter enable formal verification of throughput and latency requirements at the application level. However, this requires a performance model of the application, which is not always available. Some applications have behaviors that cannot be accurately modeled, while others are written in ways that make modeling very complicated. An example of the latter, are applications that communicate through shared memory using a programming model where communication is not explicit. To deal with these applications, we require a complementary verification approach that does not have any restrictions on the application. For this purpose, we rely on simulation-based verification. However, to manage the increasing verification complexity due to the growing amount of use-cases in embedded systems, we require composable service to enable independent verification of applications, as previously explained in Sect. 1.3.3.

There are currently three approaches to composable system design. The first involves not sharing any resources, which is trivially composable, but prohibitively expensive for systems not running safety-critical applications. The second is to statically schedule all interaction between components in the system [69] at design time. This approach requires a global notion of time and is limited to applications and hardware that can be statically scheduled. Although our SDRAM back-end is predictable and can be statically scheduled, we want to ensure that our approach applies to all applications, in particular to those that cannot be verified using formal methods. The third approach is to dynamically share resources at run-time, which is limited to combinations of inherently composable resources and

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_6, © Springer Science+Business Media, LLC 2011

arbiters [17, 48]. It is hence not suited to handle SDRAM memories, since the time to serve a request is variable and depends on other requestors. We have previously mentioned Time-Division Multiplexing (TDM) as an example of an inherently composable arbitration scheme. However, we concluded in Sect. 5.4 that TDM cannot distinguish latency-sensitive requestors without wasting bandwidth. We hence require a new approach to composable resource sharing that is more general and works in combination with our proposed SDRAM back-end and priority-based arbiters, such as Frame-Based Static-Priority (FBSP) and CCSP.

In this chapter, we present a fourth approach to composable resource sharing that works with any combination of predictable resource and $\mathcal{LR}$ arbiter without any restrictions on the application, thus widely extending the class of systems that can offer composable service. We start in Sect. 6.1 by providing an overview of our approach. Our formal model is then extended in Sect. 6.2, allowing us to provide a definition of composable service. We then show how $\mathcal{LR}$ servers can be used to provide service according to this definition, both for resources with constant and variable service cycle times. In Sect. 6.3, we propose an architecture for a resource front-end that implements the presented concepts when combined with any predictable resource. We experimentally show in Sect. 6.4 that our front-end fitted with a CCSP arbiter provides composable service when paired with both a simple SRAM controller and with our SDRAM back-end. The chapter is concluded with a summary in Sect. 6.5.

6.1 Overview of Approach

We explained in Sect. 1.3.3 that composability means that applications cannot influence each other's temporal behavior by even a single clock cycle. The problem with providing composable service in the general case is that requestors interfere with each other by changing the state of stateful resources and arbiters. This interference results in jitter in the provided service that causes both arrival times and finishing times of a requestor to change, due to the behavior of others. The key idea behind our approach is to make the provided service composable by removing this jitter. This is accomplished by delaying all signals sent from the resource to a requestor to always emulate worst-case interference from other requestors. This creates an interface towards each requestor that is independent from others in the temporal domain, as shown in Fig. 6.1. The figure shows that the resource communicates with the requestors in two ways. The first one is through the flow-control signal that accepts incoming requests. The second one is via responses that are returned. We hence need to make sure that both of these signals display composable behavior. This makes the system composable at the level of requestors, which is a sufficient condition for it to be composable at the level of applications. A drawback of making the system composable at this level is that it is not possible to benefit from slack that is generated within the application. The approach is, on the other hand, less complex to implement, since requestors do not require a notion of to which application they belong.

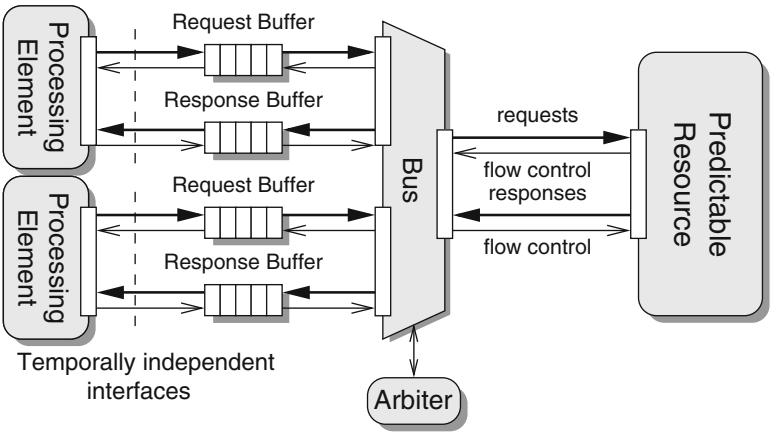

Fig. 6.1 Temporally independent interfaces are created by delaying responses and flow control

A benefit of our approach to composability is that it can be dynamically enabled or disabled per requestor at run time by turning the emulation of worst-case interference on or off. This introduces the notion of *partially composable systems*, where some applications are free from interference and others are not. The advantage of a partially composable system is that slack can be used to improve performance of requestors that do not require composable service, such as non-real-time requestors, or those belonging to applications that are verified using formal approaches.

Our approach to composable resource sharing relies on predictability, since it is not possible to emulate worst-case interference unless it is known and bounded. More specifically, we require predictable resources, where the time to serve a scheduled request is upper bounded, such as an SRAM or our proposed SDRAM back-end. We furthermore require an upper bound on interference from other requestors. Given a predictable resource, this requirement can be satisfied in three ways: (1) By characterizing the amount of service requested by each requestor in an interval and upper bounding the size of a request. This allows any predictable arbiter to be used, but is not robust in case the characterization is wrong or a requestor malfunctions. (2) Preempt a request in service after a maximum time. This solution is robust and can handle requests whose sizes are initially unknown, but is limited to predictable preemptive arbiters. (3) Use a hardware block to split up requests into *atomic service units*, referred to as atoms, with known maximum service time, as proposed in [48]. Both the second and the third solution assume that the resource supports serving requests in smaller pieces, which is typically the case for transaction-based resources like memories and peripherals. We choose this option for our approach, since it enables preemption of requests at the granularity of atoms using any predictable arbiter, thus providing maximum flexibility.

The *main benefit* of our approach is that it is built on the $\mathcal{LR}$ server abstraction. This enables composable service to be provided for *any combination* of arbiter in the

class of $\mathcal{LR}$ servers and predictable resource. An additional benefit of $\mathcal{LR}$ servers is that the latency metric, service latency (Θ), accounts for worst-case interference from other requestors, but not for *self interference*, which is the time a request waits for other requests from its own requestor. This separation is advantageous, since composability only requires us to eliminate the effects of interference from others. An implication of this is that the maximum time between the arrival time and finishing time is not constant for all requests, but changes depending on the number of requests in the Request Buffer of the requestor. Enforcing a constant delay from arrival time to finishing time requires a conservative bound on the requested service, using for instance a (σ, ρ) characterization [28], to compute the worst-case self interference for every request. This results in very pessimistic finishing times for pipelined processing elements, supporting multiple outstanding requests, as we will see in Sect. 6.4. It is furthermore very difficult to obtain an accurate characterization without unnecessarily restricting the application, which does not fit with our approach to composability. We hence choose to compute the worst-case starting times and finishing times dynamically at run time.

6.2 Formal Model

Our discussion on composability requires us to further extend our formal model, starting with a definition of the concept. The provided service is considered composable if the arrival times and finishing times of all requests from a requestor are independent of other requestors. This definition is suitable, since it describes the temporal behavior of requests at the interface of the resource. Note that the arrival time is defined with respect to available space in both the Request Buffer and the Response Buffer, and is hence not naturally independent of other requestors. This creates a dependence between the arrival time and both the starting and finishing times. Composable service according to Definition 6.1 is hence implemented by assuring that both the starting times and finishing times emulate worst-case behavior.

Definition 6.1 (Composable service). The service provided to a requestor $r \in R$ is defined as composable if $\forall \omega_r \in \Omega_r \; t_s(\omega_r)$ and $t_f(\omega_r)$ are independent of other requestors.

To provide composable service with our approach, we need to emulate worst-case interference by delaying flow-control signals and responses. This is achieved by emulating worst-case starting times and finishing times, which were previously bounded in abstract service cycles in Sect. 5.3. We now proceed by discussing how to convert these bounds from service cycles to clock cycles in an efficient manner for resources with variable service cycle length, such as our SDRAM back-end. Remember that a complete list of symbols along with a brief descriptions and page references to the definitions are found in Appendix B.2.

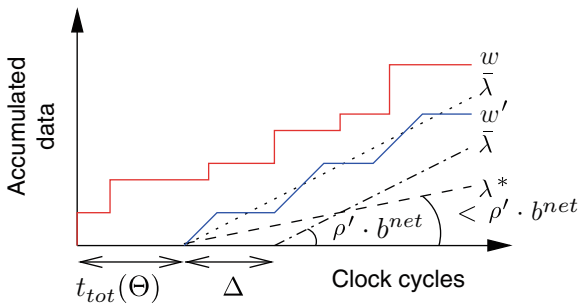

Fig. 6.2 The trade-off between service latency and net bandwidth

The service latency and completion latency have to be converted from service cycles to clock cycles to be of any practical use in a hardware implementation. For a resource with constant service cycle length, such as a Zero-bus-turnaround (ZBT) SRAM, this is easily done by multiplying the values in service cycles with the service cycle length. For our SDRAM back-end, we use (4.8) to convert the service latency. However, this solution does not work for the completion latency, as it would account for an interfering refresh for every request. This would result in extremely pessimistic finishing times, and more seriously, it would reduce the net bandwidth provided to the requestors. This leaves us with two options. The first option is to program multiple completion latencies, e.g. one for reads and one for writes, and choose among them dynamically at run time. However, this option has the drawback of making the implementation dependent on the particular resource, since different resources may have a different number of interesting cases. Instead, we opt for the second option, which is to use a single completion latency that is consistent with our computation of net bandwidth. We accomplish this by using the *average service cycle length during worst-case conditions*. We conveniently refer to this as the average service cycle length and denote it $\bar{\lambda}$. However, using the average service cycle length to convert the completion latency to clock cycles may result in non-composable behavior, since some actual service cycles are longer. This is illustrated in Fig. 6.2, where the provided service curve, w', is often behind the dotted lower bound on provided service starting at $t_{tot}(\Theta)$. This can be solved by enforcing a longer service cycle length $\lambda^* > \bar{\lambda}$, as shown in Fig. 6.2, although this reduces the bandwidth provided to the requestor. Instead, we enforce the dash-dotted line denoted $\bar{\lambda}$ that is based on the average service cycle length, but has an additional latency offset, Δ. This enables us to provide the intended bandwidth by increasing latency. We proceed by explaining how to compute the average service cycle length and the required service latency offset.

The average service cycle length is defined according to Definition 6.2. The intuition behind the definition is that gross memory efficiency is the average fraction of time during which requested data is transferred to and from the memory. The product of gross efficiency and the average service cycle length should hence correspond to the average numbers of cycles with data transfer during a service cycle. A service cycle only transfers data during the access pattern, making the

average number of cycles with data transfer constant and equal to $t_{transfer}$, previously computed in (4.4). Gross memory efficiency can hence be expressed according to $e^{gross} = \frac{t_{transfer}}{\bar{\lambda}}$. By solving for the average service cycle length, we arrive at the expression in Definition 6.2. Note that the length of the service cycle is independent of whether or not the data is requested by a requestor, and does hence not depend on data efficiency. Intuitively, the average service cycle length works like a savings account. The length of every service cycle budgets a constant amount of time to pay for overhead, such as read/write switches or refresh. Saving this amount of time for every service unit during $tREFI$ cycles asserts that the provided gross bandwidth equals b^{gross}, that all possible overhead due to read/write switches is paid for, and that there are t_{ref} clock cycles left to pay for the refresh.

Definition 6.2 (Average service cycle length during worst-case conditions). The average service cycle length during worst-case conditions, expressed in clock cycles, is denoted by $\bar{\lambda} \in \mathbb{R}^+$, and is defined as $\bar{\lambda} = \frac{t_{transfer}}{e^{gross}}$.

The service latency offset, Δ, must assert that the lower bound on provided service remains valid, despite the use of the average service cycle length. The offset corresponds to the difference between the maximum and the average service cycle length, as expressed in Definition 6.3. This offset is tight if a requestor receives worst-case interference as predicted by its service latency bound, starting with a refresh, and is scheduled close to the next refresh, (i.e. service latency is slightly less than $2 \cdot tREFI$). In this case, the second refresh is not included in the service latency and there has not been any time for $\bar{\lambda}$ to amortize it, hence requiring the offset in Definition 6.3. However, if the bound on service latency is short and only contains a blocking atom and a refresh, then $\bar{\lambda}$ almost entirely amortizes the following refresh by the time it happens, making the service latency offset appear pessimistic. This is experimentally shown in Sect. 6.4.

Definition 6.3 (Service latency offset). The service cycle offset, expressed in clock cycles, is denoted by $\Delta \in \mathbb{N}$, and is defined as

$$\Delta = \begin{cases} t_{ref} + t_{wtr} + t_{read} - \lceil \bar{\lambda} \rceil & \text{if read-dominant or mix-read-dominant} \\ t_{ref} + t_{rtw} + t_{write} - \lceil \bar{\lambda} \rceil & \text{if write-dominant or mix-write-dominant} \end{cases}$$

For the computed finishing times to be correct, the number of pipeline stages in the architecture between the Request Buffer and the Response Buffer, n_{pipe}, must be considered. The pipeline stages add a constant delay to the finishing time and are hence included in the service latency of the requestor. All pieces are now in place to define service latency and completion latency, expressed in clock cycles. This is done in Definitions 6.4 and 6.5, respectively. The completion latency is defined as a real number, resulting in real starting times and finishing times in the implementation. We return to address this issue in Sect. 6.3.3.

Definition 6.4 (Service latency (clock cycles)). The service latency of a requestor $r \in R$, expressed in clock cycles, is denoted by $\Theta_r^{cc} \in \mathbb{N}$, and is defined as $\Theta_r^{cc} = t_{tot}(\Theta_r) + \Delta + n_{pipe}$.

Definition 6.5 (Completion latency (clock cycles)). The completion latency of a requestor $r \in R$, expressed in clock cycles, is denoted by $l_r^{cc} \in \mathbb{R}^+$, and is defined as $l_r^{cc} = \bar{\lambda} \cdot l(\omega_r^k)$, which is equivalent to $\frac{\bar{\lambda}}{\rho_r'}$.

6.3 Architecture

In this section, we introduce the architecture of our proposed resource front-end that implements the concepts from Sect. 6.1 using the model from Sect. 6.2. We start by presenting an overview of the architecture in Sect. 6.3.1, followed by brief descriptions of the functional blocks in Sects. 6.3.2–6.3.4. The design and implementation of all blocks in the implementation are described in full detail in [122].

6.3.1 Architecture Overview

The proposed resource front-end is located in front of a predictable resource, as shown in Fig. 6.3. The architecture is comprised of three main simple and reusable blocks: an Atomizer, a Delay Block, and a Data Bus with an arbiter. Additionally, there is a Configuration Bus that allows registers inside the different blocks to be programmed via memory mapped I/O during use-case transitions [46]. The blocks communicate using a Device Transaction Level (DTL) protocol [101], which is a standardized communication protocol similar to Advanced eXtensible Interface (AXI) [14]. All ports shown in Fig. 6.3 are DTL ports.

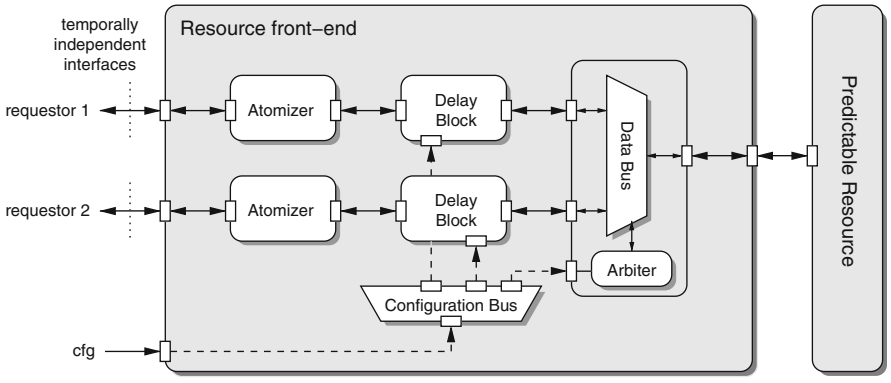

Fig. 6.3 An instance of the proposed architecture supporting two requestors

The architecture achieves composability by combining two approaches to composable system design at the block level. The Atomizer and Delay Blocks are composable because they are not shared with other requestors, corresponding to the first approach presented earlier. The Data Bus shares the predictable resource using an arbiter in the class of $\mathcal{LR}$ servers. The Delay Block hides the interference caused by scheduling and accessing the resource by emulating worst-case interference from other requestors, according to our proposed fourth approach. This creates an interface per requestor that is temporally independent of the behavior of other requestors, as shown in Fig. 6.3. Note that this architecture is similar to the conceptual image in Fig. 6.1, since the Request Buffer and Response Buffer are located inside the Delay Block.

6.3.2 Atomizer

The Atomizer is responsible for splitting requests into atoms with a fixed programmable size. This ensures that requests have a known size that can be served in a bounded time by the resource. The design is hence predictable without relying on a characterization of the maximum request size or requiring explicit support for preemption in the arbiter. Fixed-sized requests furthermore simplify other blocks in the architecture. The size of an atom corresponds to the service unit of the resource, as mentioned in Sect. 5.2. For a typical SRAM, the natural service unit is a single word, but for our predictable SDRAM back-end it is equal to the granularity of the access patterns, g, previously defined in Definition 4.2. The original sizes of the requests are stored in the Atomizer to allow it to merge arriving responses back into the size expected by the requestor.

6.3.3 Delay Block

The most complex block in the architecture is the Delay Block, shown in Fig. 6.4, and we hence explain this block in greater detail than the rest. The purpose of the Delay Block is to emulate maximum interference from other requestors created either in the resource or arbiter to provide a composable interface towards the Atomizer. This makes the interface of the entire front-end composable, since the Atomizer is not shared. The Delay Block is composable if all arrows on the interface in Fig. 6.4 pointing left towards the Atomizer exhibit composable behavior, which implies that both response data and flow control signals must emulate maximum interference. We proceed by discussing how the Delay Block accomplishes this, based on the results from Sect. 6.2. After this, we discuss how to configure the Delay Block.

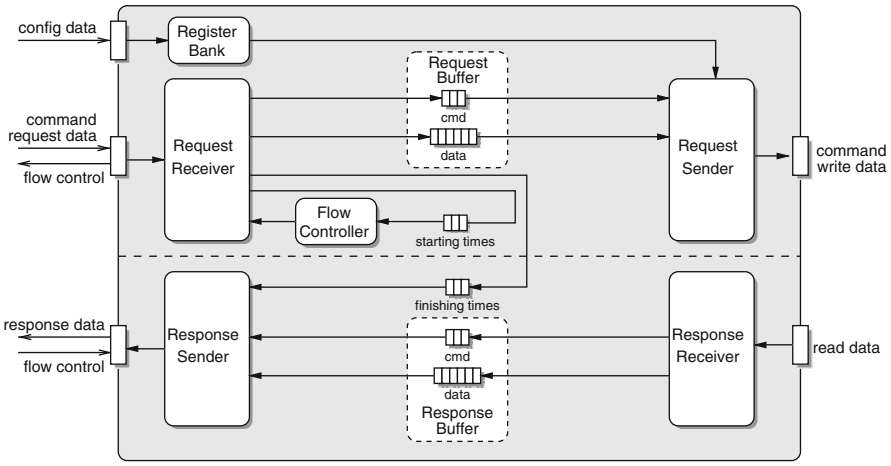

Fig. 6.4 Delay Block architecture

6.3.3.1 Composable Responses

Requests are received by the Request Receiver according to the DTL protocol. Incoming requests are split into a command (read/write information and request size) and data (for write requests), and are stored in the Request Buffer. The Request Receiver then waits until the request has completely arrived in the Request Buffer and there is enough space to store its response in the Response Buffer, implementing the definition of arrival in Definition 5.6. At this time, it computes the worst-case starting time and the worst-case finishing time, according to (5.2) and (5.3), and stores the results in two respective FIFO buffers.

The Request Sender pops the request at the head of the Request Buffer and presents it to the Data Bus, such that it can be scheduled for resource access by the arbiter. This is further discussed in Sect. 6.3.4.

Responses are received by a Response Receiver and are stored in the Response Buffer. The Response Sender pops the worst-case finishing time from the head of the FIFO buffer and waits until the appropriate clock cycle to release the response, thus emulating maximum interference according to the $\mathcal{LR}$ server model. This ensures that the finishing times of the requestor are unaffected by the interference from others, which is one of the two requirements to be composable according to Definition 6.1.

The notion of time in the Delay Block is implemented using a locally running wrapping cycle counter. The counter has to be wide enough to represent the maximum number of clock cycles between the arrival time and finishing time of a requestor. Refer to [122] for more details on the implementation of this time base.

6.3.3.2 Composable Flow Control

Having taken care of composable responses, we proceed by discussing the issue of composable flow control, which is required to make the arrival times of a requestor independent of others. The arrival time of a request is assigned when it has completely arrived in the Request Buffer and there is enough space to store a response in the Response Buffer, and it is hence determined by the state of both of these buffers. However, the time at which a request leaves the Request Buffer and enters the Response Buffer depends on the starting time and the completion latency, which may be affected by other requestors. We must hence make sure that space in these buffers is claimed and released independently of others. This is done by a Flow Controller block. For the Request Buffer, we base the flow control on the worst-case buffer filling. The Flow Controller has a counter that is initialized to the size of the Request Buffer. This counter is decremented whenever a request enters the Request Buffer and incremented at the computed worst-case starting times, removing the dependence on the actual starting times. For the Response Buffer, the Flow Controller reserves space at the arrival time of a request, since this is required to ensure that the bounds on starting times and finishing times are valid. However, this also removes the dependency on the starting time and the state of the resource, thus effectively serving a double purpose. Together, these buffer management strategies ensure that the arrival times of a requestor are unaffected by interference from others, which is the remaining requirement to provide composable service according to Definition 6.1.

6.3.3.3 Discrete Approximation Mechanism

A problem arises if the completion latency, l^{cc}, is not an integer multiple of clock cycles, which it typically is not. Rounding off the value causes the enforced worst-case finishing times to diverge from the exact values over time for a busy requestor. Similarly to what we discussed in Sect. 6.2, rounding the value downwards makes the finishing times too optimistic, leading to non-composable behavior. On the other hand, rounding upwards makes the finishing time too pessimistic and causes the actual provided bandwidth to be less than the allocated bandwidth, $\rho' \cdot b^{net}$. As we will see in Sect. 6.4, this divergence is significant for requestors with high allocated rates for resources with small service units, where completion latencies are in the order of a few clock cycles. The maximum theoretical impact of rounding up is that almost 50% of the resource capacity is wasted for a requestor that is allocated slightly less than 50% of the resource, resulting in a completion time of just above one clock cycle that is rounded off to two. The problem with rounding up and down is illustrated in Fig. 6.5. Note that the requestor in the figure is busy throughout the entire shown interval, although the busy line has been omitted for clarity.

Our solution to this problem is to implement a mechanism that changes between using the rounded up and rounded down completion latencies in a weighted fashion to conservatively approximate the actual value, as shown in Fig. 6.5. The fraction of

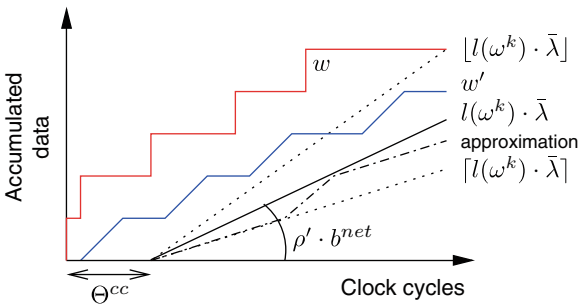

Fig. 6.5 Diverging finishing times prevented by discrete approximation of the completion latency

the service units for which the rounded down value should be used is expressed as $\eta = \lceil \bar{\lambda} \cdot l(\omega^k) \rceil - \bar{\lambda} \cdot l(\omega^k)$. Since $\eta \in \mathbb{R}$ and $0 \leq \eta < 1$, our mechanism requires a discrete approximation based on integer arithmetic that has a fast and simple hardware implementation. For this purpose, we reuse the service representation introduced for the CCSP rate regulator in Sect. 5.6. We hence represent η as a fraction of integers according to $\eta = n^*/d^*$, where $n^*, d^* \in \mathbb{N}^+$ and $n^* \leq d^*$. The values of n^* and d^* are chosen to be the (n^*, d^*) pair that provides the closest approximation of η. The accuracy of this approximation is only limited by the number of bits used to represent n^* and d^*. The n^* and d^* values are computed for all requestors at design time and are programmed at run time.

The behavior of the mechanism is such that the approximated completion latency is $\lceil l^{cc} \rceil - n^*/d^* \approx \lceil l^{cc} \rceil - \eta$. The implementation is based on a credit counter, c^*, as described by the pseudo code in Algorithm 6.1. The credit counter is set to zero at the start of a busy period, which is detected by checking if the first parameter of the max expression in (5.3) is larger than the second. The mechanism then alternates between the rounded up and the rounded down completion latencies based on the value of the counter. The approximation done by the mechanism is conservative and guarantees that the maximum difference between the approximated and actual completion latency is less than one clock cycle at any time.

Algorithm 6.1 Mechanism for discrete approximation of completion latency.

for all $\omega_r^k \in \Omega_r$ **do**
  if $t_a(\omega_r^k) + \Theta_r^{cc} \geq \hat{t}_f(\omega_r^{k-1})$ **then** // Start of busy period
    $c_r^* \leftarrow 0$
  end if

  if $c_r^* < d_r^* - n_r^*$ **then** // Rounding up
    $c_r^* \leftarrow c_r^* + n_r^*$
    $\hat{t}_f(\omega_r^k) \leftarrow \max(t_a(\omega_r^k) + \Theta_r^{cc}, \hat{t}_f(\omega_r^{k-1})) + \lceil l_r^{cc} \rceil$
  else // Rounding down
    $c_r^* \leftarrow c_r^* + n_r^* - d_r^*$
    $\hat{t}_f(\omega_r^k) \leftarrow \max(t_a(\omega_r^k) + \Theta_r, \hat{t}_f(\omega_r^{k-1})) + \lfloor l_r^{cc} \rfloor$
  end if
end for

6.3.3.4 Configuring the Delay Block

The Delay Block is programmed with the service latency and completion latency of its requestor to facilitate run-time computation of the worst-case starting times and finishing times. Note that the Atomizer ensures that all requests have the same size and that we only have to program one completion latency per requestor. The presence of an Atomizer thus reduces the amount of computation required to dynamically determine the completion latency of a particular request, or the space required to store precomputed values.

The programmed service latencies and completion latencies are computed according to Definitions 6.4 and 6.5, respectively. The completion latency is rounded upwards to the closest integer before programming, although the discrete approximation mechanism asserts that this does not negatively impact throughput. The rounded down completion latency, required by the mechanism, is easily obtained by subtracting one from the programmed value. Every block in our implementation is output registered, resulting in a total of four pipeline stages between the Request Buffer and Response Buffer. Four clock cycles are hence added to the service latency to account for the pipelining in the implementation, as stated by Definition 6.4.

Composable service is dynamically disabled by programming both the service latency and completion latency to zero clock cycles. This feature has two advantages. First, it allows requestors that do not require composable service to use slack generated by others, as mentioned in Sect. 6.1. The second advantage is that it enables requestors that require composable service to share hardware with requestors that do not by enabling or disabling composable service on use-case transitions.

6.3.4 Data Bus

The Data Bus is a regular DTL bus that schedules requests according to the policy of an attached arbiter that belongs to the class of $\mathcal{LR}$ servers, such as TDM, FBSP, or CCSP, previously presented in Chap. 5. The Data Bus is a very general building block with multiple DTL input ports and a single DTL output port. A challenge with using such a general block is that it is not aware of the type of resource it is providing access to. This makes it difficult to know when to trigger a new scheduling decision, since it is not known in advance when the previous request is finished. We proceed by discussing five options:

1. Schedule a request when the resource raises the accept signal on the DTL interface. This results in that the resource is idle during one clock cycle when the arbiter schedules the next request. For small requests, such as word-sized requests for an SRAM, this may reduce throughput up to 50%.
2. Make a new scheduling decision periodically, where the period is set to the worst-case service cycle length. This approach works well for resources with a constant

access time, such as an SRAM. In this case, the best-case completion latency equals the worst case, resulting in a simple periodic counter. However, for our SDRAM back-end, this would assume a switching pattern and a refresh pattern for every access, reducing the provided net bandwidth as discussed previously.

3. Schedule the next request immediately after the previous has been accepted. This way, a request is always scheduled when the resource is ready to accept. However, arbitration may be done on old state, missing later arrivals from latency-sensitive requestors. This approach increases the average latency of sensitive requestors unnecessarily.
4. Reevaluate the arbitration decision every clock cycle to ensure that it is always up to date, improving the average case latency of sensitive requestors. A drawback with this approach is that it complicates the interaction with the accounting mechanism in the arbiter. This is because the accounting should be updated exactly once every service cycle to preserve the net bandwidth guarantee of the requestors.
5. Schedule a new requestor based on the minimum service cycle length. With this approach, a new scheduling signal is generated based on a programmed minimum service cycle length, $\check{\lambda}$. For our SDRAM back-end, this corresponds to the time required to execute the shorter of a read and a write pattern, as expressed in (6.1). This ensures that a scheduling decision has been made when the resource is ready to accept without committing to a decision unnecessarily early. The resource may not accept the scheduled request immediately, for instance if there is a refresh or a read/write switch. The timer generating the scheduling signal is hence not reset until the resource accepts the request, causing the service cycle to dynamically stretch with the behavior of the resource. This approach, employed in our implementation, is hence a compromise between scheduling only once and using the most up-to-date information possible. It furthermore makes exactly one scheduling decision per service unit, simplifying the interaction with the accounting mechanism in the arbiter.

$$\check{\lambda} = \min(t_{read}, t_{write}) \tag{6.1}$$

When the arbiter schedules a request, the Data Bus stores an identifier to the scheduled port, so that responses are demultiplexed correctly to their respective Delay Blocks. These identifiers are stored in separate FIFO buffers for read and write requests, since the DTL protocol does not enforce ordering between reads and writes.

6.3.5 Synthesis Results

The proposed front-end has been implemented in VHDL [122] and synthesized in a 90 nm CMOS process using Cadence RTL Compiler. Synthesis is done using a 50% clock duty cycle, and with 20% of the cycle time as input and output delay

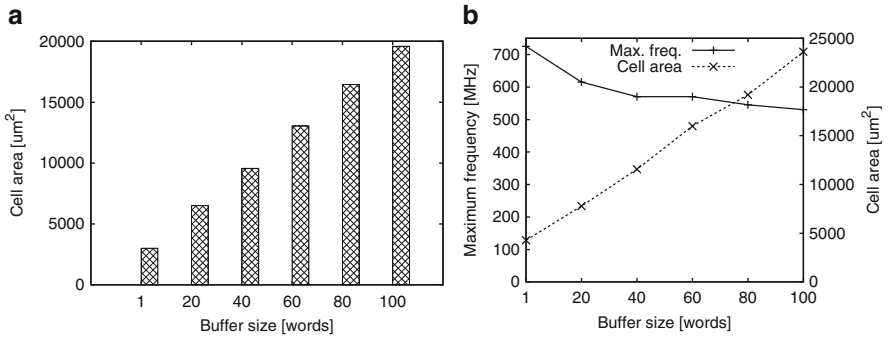

Fig. 6.6 Synthesis results for the Atomizer. (**a**) Cell area for different buffer sizes. (**b**) Maximum frequency and corresponding cell area for different buffer sizes

with 10% clock skew. Both clock-gate insertion and scan insertion are disabled, and we synthesize under worst-case commercial conditions. We proceed by walking through the synthesis results for each of the blocks in the front-end, starting with the Atomizer. Figure 6.6 shows the cell area of the Atomizer with a speed target of 200 MHz, suitable for our example DDR2-400 memory, as the size of the buffer storing the original sizes of requests is varied in the range [1, 100] words. We conclude from the figure that the Atomizer is a small and simple block with a cell area of less than 5000 μm^2 occupied by logic, while the rest is buffering. The maximum frequency and the corresponding cell area of the Atomizer are shown in Fig. 6.6b. We observe that the Atomizer synthesizes above 700 MHz with a buffer size of one word. The area of the implementation grows linearly as the buffer size is increased, while the maximum frequency reduces, ending at 530 MHz for the instance with a buffer size of 100 words.

Next, we look at the Delay Block, which is a considerably more complex block. The size and maximum frequency of this block depends on the sizes of the many buffers, on the width of time stamps, and on the precision of discrete approximation mechanism. For the synthesized instance, we have used 15 bits for the time stamps, and varied the buffer sizes and precision. The considered buffers are the FIFOs with starting and finishing times, and the Request and Response Buffers, which both have separate queues for commands and data. For simplicity, we vary the sizes of all these buffers uniformly. This is reasonable assuming an atom size of a single word, suitable for an SRAM, since the buffers for data and commands should have equal sizes in this case. The cell area at 200 MHz for different buffer sizes and precisions are shown in Fig. 6.7a. We note that the Delay Block is more complex than the Atomizer, considering that it is three times larger with minimum buffering. We also see that the many buffers cause the area to increase quickly as the buffer depths are increased. The impact on area when changing the precision used in the discrete approximation mechanism are hardly noticeable in the Delay Block. Increasing precision from 4 to 10 bits adds just below 10% to the area for a Delay

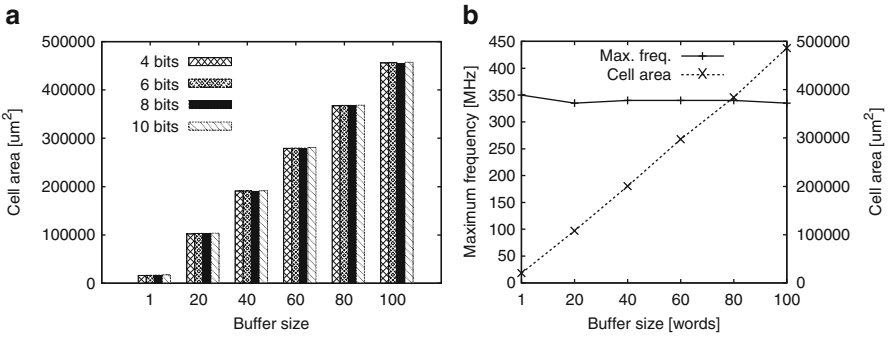

Fig. 6.7 Synthesis results for the Delay Block. (**a**) Cell area for different buffer sizes and precisions. (**b**) Maximum frequency and corresponding cell area for different buffer sizes with 10 bits of precision

Block with buffer sizes of a single element, and the effect is negligible for larger buffer sizes. Figure 6.7b shows that the maximum frequency of the Delay Block is relatively stable around 340 MHz as the buffer sizes change. The precision is 10 bits for all synthesized instances in this figure, although synthesis results omitted here indicate that reducing precision to 4 bits increases the maximum frequency by less than 5% for all buffer sizes.

The last block is the Data Bus, combined with a CCSP arbiter. The area and maximum frequency scales with the number of requestors for both of these components. The CCSP arbiter is additionally affected by the chosen precision. We see in Fig. 6.8a how the cell area changes with the both the number of requestors and precision. The total area is dominated by the arbiter, constituting some 60–70% of the area of the combination. Figure 6.8b shows that the maximum frequency of the combined Data Bus and arbiter is between 350 and 450 MHz, depending on the number of requestors. The maximum frequency of the combination is lower than that of the CCSP arbiter alone, which synthesizes at approximately 600 MHz. The reason is that a command is scheduled by the arbiter and moved from the input of the Data Bus to the output in a single clock cycle. The maximum frequency is fast enough to keep up with most DDR2 memories, but improvement is required to keep up with any memory in the DDR3 generation. It is possible that the maximum frequency can be improved by pipelining the arbitration, although no attempts have been carried out in this direction. In conclusion, it appears that the Delay Block is the bottleneck in the current implementation, limiting the maximum clock frequency to approximately 350 MHz. However, judging from the trend in Fig. 6.8b, it seems like the Data Bus may become the limiting factor if the number of requestors is scaled up further, beyond the needs of our memory controller.

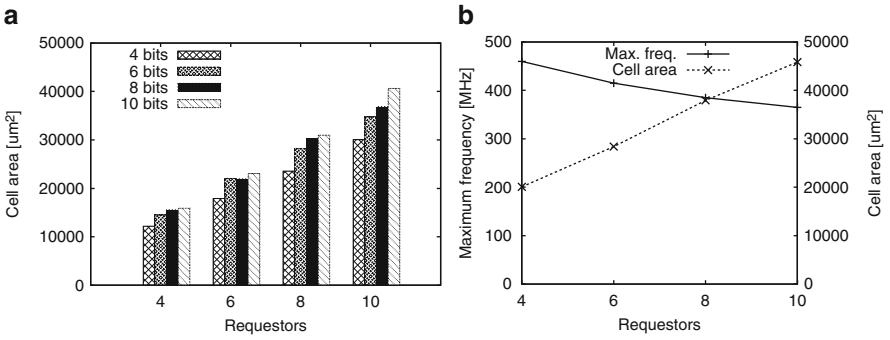

Fig. 6.8 Synthesis results for the Data Bus with a CCSP arbiter. (**a**) Cell area for different number of requestors and precisions. (**b**) Maximum frequency and corresponding cell area for different number of requestors with 10 bits of precision

6.4 Experiments

We proceed by experimentally evaluating our approach to composable resource sharing, using both a simple SRAM controller and our proposed SDRAM back-end. The behavior of the resource front-end together with these resources is studied to increase the understanding of our approach. We furthermore evaluate the tightness of the bound on finishing time, look at the added average latency and buffering requirements of our approach, and examine the benefits of distributing slack bandwidth to requestors that do not require composable service. Most importantly, we also demonstrate that the arrival times and finishing times of a requestor are independent of other requestors and hence that our design provides composable service according to Definition 6.1.

6.4.1 SRAM Experiments

For our first set of experiments, we use a simple SRAM controller with constant service cycle length. This is a simpler case than our SDRAM back-end with variable service cycle length, allowing us to build up the complexity of the experiments gradually. The SRAM controller is running at 200 MHz with a 32-bit data path, offering a gross bandwidth of 800 MB/s. The service unit size of this controller is a single word (4 bytes), and the length of a service cycle is one clock cycle. The proposed resource front-end is fitted with a CCSP arbiter and is located in front of the SRAM controller. The experimental setup with SystemC models from Sect. 5.7 is used as a starting point for the experiments in this chapter. Traffic generators generating requests according to a normal distribution are used to

Table 6.1 SRAM use-case specification and configuration

Requestor	Type	b_r (MB/s)	Size (B)	p_r	σ'_r (su)	ρ'_r (su/sc)	Θ_r^{cc} (cc)	l_r^{cc} (cc)
r_0	Read	210.0	32	0	1.0	0.263	5	3.80
r_1	Write	210.0	8	1	1.0	0.263	6	3.80
r_2	Read	210.0	4	2	1.0	0.263	9	3.80
r_3	Write	20.0	16	3	1.0	0.025	19	40.00

represent the CoMPSoC processor tiles [82] that are interconnected using a model of the Æthereal [38, 47] Network-on-Chip (NoC). For continuity, we reuse the use-case with four requestors from Sect. 5.7. However, we scale down the request sizes in proportion to the reduction in access granularity to keep the sizes in service units constant. The access granularity of the SDRAM back-end in Sect. 5.7 was 64 B, while it is 4 B for the SRAM controller in this section. The request sizes in the original use-case are hence divided by $64/4 = 16$. The data efficiency of the requestors is 100%, making the offered gross and net bandwidths equal. The revised use-case is presented in Table 6.1. Since the SRAM controller in this experiment provides higher net bandwidth than the DDR2-400 memory in the previous chapter, the allocated rates of the requestors are decreased to remain the fraction between their requested bandwidths and the total net bandwidth. With these changes, the total allocated bandwidth in this use-case is 81.4% of the provided net bandwidth, indicating a moderate load. Priority levels are assigned in ascending order and the service latencies in clock cycles (cc), computed according to Definition 6.4, are listed in the table. The service latency offset in this setup is zero clock cycles, since we are using an SRAM with constant service cycle time. The completion latencies of r_0, r_1, and r_2 are 3.80 clock cycles. As mentioned in Sect. 6.3.3.3, rounding this value downwards might lead to non-composable behavior, and rounding it upwards results in that the provided bandwidth is reduced from 210 to 200 MB/s (1 word / 4 clock cycles), failing to satisfy the bandwidth requirements of the requestors. This is prevented by our proposed approximation mechanism, which ensures that each requestor receives their allocated bandwidth in a composable manner.

6.4.1.1 General Observations

For our first experiment, we simulate the use-case in Table 6.1 during 100 ms to observe the behavior of the front-end and the SRAM controller. We size all buffers to 255 words to prevent overflow, thus enabling us to evaluate both the added latency and buffering that follows from delaying responses. Figure 6.9 plots the worst-case finishing times, the actual finishing times, and the actual starting times versus the arrival times of the first 200 requests from requestor r_2.

By studying the figure, three general observations can be made. First, that it is possible to see which requests that start a new busy period by looking at the bound on finishing time. The starting times of these requests are determined by

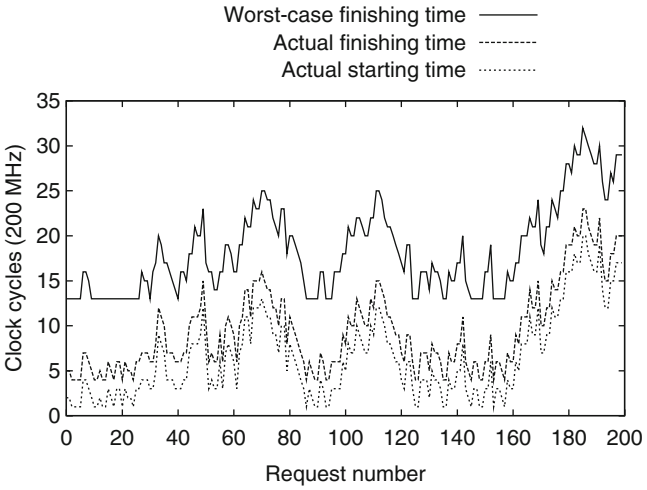

Fig. 6.9 The first 200 requests of r_2 in the SRAM use-case

the service latency in the first term in (5.2), as opposed to by the finishing time of the previous request. The finishing time in this case hence equals the sum of the service latency and the rounded up completion latency, which is 13 clock cycles in total for requestor r_2. This corresponds to the lowest bounds on finishing time in the figure, while self-interference during busy periods increases the bound. The second general observation is that the number of clock cycles between the starting times and the finishing times is constant for all requests and equal to three clock cycles. This is expected, since the SRAM controller has a constant service cycle time. One out of the three clock cycles is when the request is served, while the other two are due to the two pipeline stages between the SRAM controller and the Response Buffer. The third observation is that the number of clock cycles between the worst-case finishing times and the arrival times in Fig. 6.9 is not constant for all requests, as mentioned in Sect. 6.1, since the traffic generators are pipelined and support multiple outstanding requests. The drawback of enforcing a constant time between the arrival time and finishing time is that the constant would have to be at least equal to our worst case, which is 523 clock cycles in this simulation. However, analytically computing this value as the worst case assumes a perfect characterization of the requested service and its resulting self interference, which is very difficult to obtain. Actual analytical results are likely to add pessimism, further increasing this delay. *From this observation, we conclude that enforcing a constant time between the arrival time and finishing time of a request results in very pessimistic latencies.*

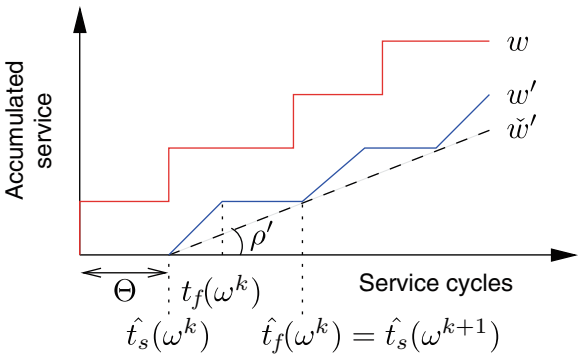

Fig. 6.10 Atoms finish before the computed bound, since they are served non-preemptively

6.4.1.2 Tightness of Bound on Finishing Time

We proceed by focusing our attention on the bound on finishing time. We see in Fig. 6.9 that the worst-case finishing times are larger than the actual finishing times, indicating that the bound is conservative in the shown interval. The minimum difference between the worst-case and actual finishing times during this simulation is 7 clock cycles. There are three reasons why the bound is not perfectly tight. The first reason is that the requestor does not experience the maximum interference predicted by the CCSP arbiter. The service latency bound of the requestor is 4 service cycles, while the arbiter measures a maximum interference of 2 service cycles. The reason the latency bound provided by CCSP is not tight in this case is because it assumes that service is provided in a continuous manner, as discussed in Sect. 5.7.4, while it is actually done discretely. The second reason the bound is not perfectly tight is also related to discrete versus continuous service. The finishing time of a request is computed based on the $\mathcal{LR}$ service guarantee provided by the arbiter. This bound assumes that a requestor receives its allocated bandwidth in a continuous fashion after the service latency, as shown in Fig. 6.10. The computed finishing time, $\hat{t}_f(\omega^k)$, is hence after the completion latency when the next request from the requestor is scheduled. However, atom-sized requests are served in a non-preemptive manner and either receives service at the full capacity of the memory, or not at all. They are hence guaranteed to finish one service cycle after their starting times, corresponding to $t_f(\omega^k)$ in Fig. 6.10. The next request from the requestor is then scheduled at the originally computed finishing time when the server becomes available again to the requestor. The impact of this effect is that the computed finishing time over-estimated by $\lceil l(\omega_r^k) \rceil - 1 = \lceil 1/\rho' \rceil - 1$ service cycles for atom-sized requests. For requestor r_2 in the use-case, this corresponds to 3 service cycles, which is equal to 3 clock cycles with our SRAM memory. This problem can be addressed by programming a second completion latency of one service cycle that is used when computing finishing times, while keeping the regular one for starting times. However, we did not implement this optimization. The third reason the bound is not tight is related to blocking. An extra service unit is added to the service latency

in (4.8) to account for that a request may arrive just after a scheduling decision is taken. This actually over-estimates the blocking with one clock cycle, considering that a request must arrive at least one cycle after a scheduling decision is taken to be blocked. Since a service cycle is a single clock cycle for the SRAM controller, blocking actually cannot occur, although it is still included in the bound. Together, these three reasons explain why the service latency bound is not perfectly tight in this simulation. *We conclude that the bound on finishing time is conservative, but not tight.*

6.4.1.3 Added Latency and Buffering

We now examine the cost of composable service for our observed requestor in terms of added latency and buffering. The average actual finishing time and the average worst-case finishing time for r_2 during the simulation are 59.5 and 68.6 clock cycles after the corresponding arrivals, respectively. This corresponds to an increase of 15.2%, supporting the intuition that delaying responses makes it more difficult to satisfy requirements on average-case latency. Delaying responses furthermore implies that more data has to be stored in the Response Buffer to prevent reducing throughput. The amount of extra data to buffer is related to the tightness of the bound on finishing time, since this determines the extra time an atom spends in the Response Buffer before being released. Without delaying responses, the read requestors have a maximum Response Buffer filling of one command and one data word each, since responses are immediately passed on to the Atomizer. When enabling delays, the maximum buffer filling increases with one command and one data word for r_0 and two commands and two data words for r_2. These results are not unexpected, since the requests of r_2 are buffered an extra 9 clock cycles on average, roughly corresponding to slightly more than two completion latencies. *We conclude that enabling composable service according to our approach increases the finishing times of the requestors, thus requiring larger buffers to sustain throughput.*

6.4.1.4 Composable SRAM Controller

For our second experiment, we experimentally demonstrate that the resource front-end makes the service provided by the SRAM controller composable. In this experiment, we illustrate the consequences of small changes in application software by simulating the use-case twice (case 1 and case 2) with different variances in the request generation for r_0. We additionally increase the allocated burstiness of r_0 in Table 6.1 to $\sigma'_{r_0} = 8$. This creates larger service variations for lower priority requestors, allowing us to visualize our point more clearly. The results for requestor r_2 are shown in Fig. 6.11a. We see that changing the variance causes the actual finishing times of the requests to change, making the system non-composable. However, the requests are held in the Delay Block until their worst-case finishing times, which are completely overlapping for the two cases, indicating that requests

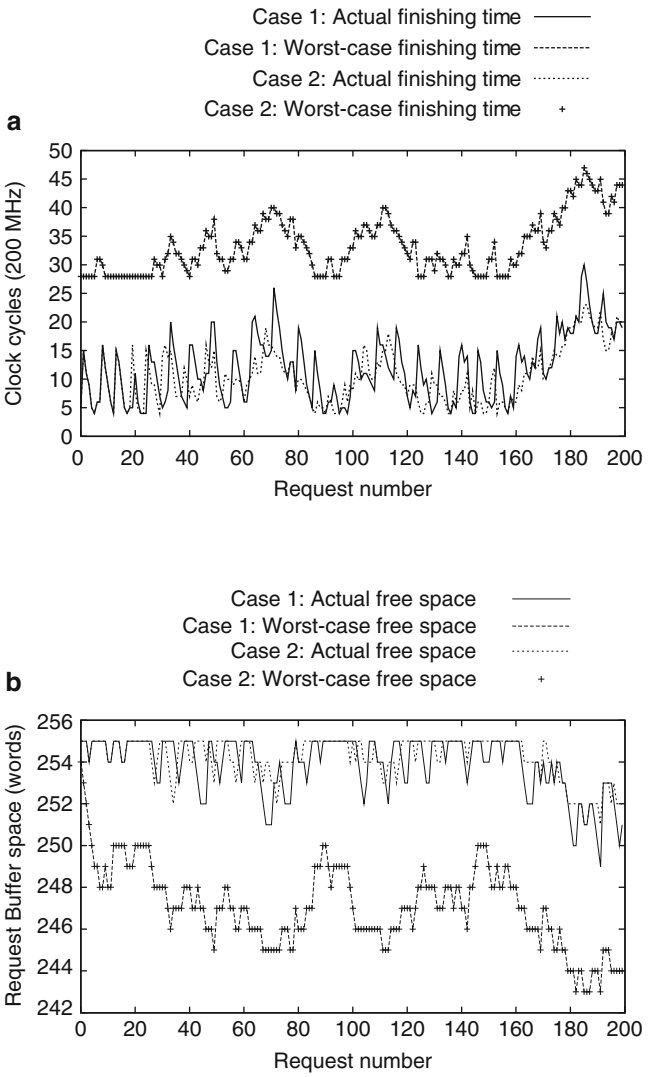

Fig. 6.11 SRAM controller behaving in a composable manner. (**a**) Request releases are unaffected by other requestors. (**b**) Worst-case Response Buffer space is unaffected by other requestors

are released from the Delay Block at the same time regardless of these changes. Making the finishing times of requestors independent of each other in this way delivers on one of the two requirements in Definition 6.1 for the service provided by the front-end to be considered composable. The remaining requirement is that also the arrival times of the requestors should be independent. As previously explained in Sect. 6.3.3.2, this is accomplished by basing the flow-control on the worst-case

Request Buffer filling, rather than the actual case. The worst-case buffer filling is stored in a counter in the Delay Block and is computed based on the worst-case starting time of a requestor, making it independent of actual interference from other requestors. Figure 6.11b shows that the worst-case Request Buffer filling of r_2 is unaffected when the behavior of r_0 changes, although the actual filling changes. This implies that the arrival times of the requestor are also unaffected. Similar experiments are performed with the RTL implementation in [122]. The front-end is shown to provide composable service with an SRAM controller, both in behavioral simulation and on FPGA. *We conclude that the service provided by the resource front-end combined with an SRAM controller is composable.*

6.4.1.5 Distributing Slack Bandwidth

The third experiment shows how to increase the performance of requestors that do not require composable service. We now consider r_2 as a non-real-time requestor and program its service latency and completion latency to zero clock cycles to disable the emulation of worst-case interference. This causes requests to be released at the actual finishing time, as opposed to the worst-case finishing time. This reduces the release time of the requests from r_2 by 13.2%, as we have seen in our first experiment. However, the performance of the requestor may be further improved by distributing the slack bandwidth in the use-case, corresponding to the 18.6% of unallocated bandwidth and any allocated bandwidth that is not used by its requestor. To demonstrate the benefit of slack distribution, we compare a work-conserving instance of CCSP to our non-work-conserving one. As previously discussed in Sect. 5.2.2, a work-conserving arbiter always schedules a request when there is a backlogged requestor. The highest priority backlogged requestor is hence scheduled if there are no eligible requestors. The use-case in Table 6.1 is simulated twice for 100 ms, the first time with a work-conserving arbiter, and the second time with a non-work-conserving instance. Figure 6.12 illustrates the results for the first 500 requests from r_2. It is clear that disabling the emulation of worst-case interference causes requests to finish earlier. However, we note that the impact of distributing the unallocated net bandwidth is more significant in this use-case. In fact, the average finishing time of request from r_2 is reduced from 59.5 clock cycles after the arrival to 5.5 clock cycles, corresponding to a reduction of 90.7%. This large difference is due to that bandwidth is allocated very closely to the average requested bandwidth, thus causing self interference to increase quickly if the requested service is bursty. *From this experiment, we conclude that disabling emulation of worst-case interference reduces the finishing times of requestors that do not require composable service. The finishing times may further reduce significantly by using a work-conserving arbiter to distribute slack bandwidth.*

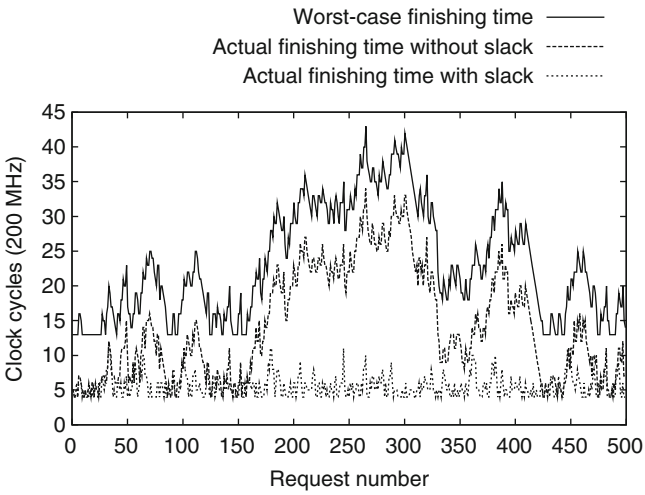

Fig. 6.12 Using a work-conserving arbiter to distribute unallocated bandwidth may significantly reduce finishing times

6.4.2 SDRAM Experiments

For our second set of experiments, we evaluate our approach to composable service when pairing our resource front-end with the predictable SDRAM back-end proposed in Chap. 4. The setup used in this experiment hence corresponds to the illustration previously shown in Fig. 2.9. Just like in our experiments in Sect. 5.7, the SDRAM back-end is connected to our example 16-bit DDR2-400 memory, using the pattern set generated by the bank scheduling algorithm with $BL = 8$ and $BC = 1$ from Table 4.3. The access granularity of the memory is hence 64 B and the SDRAM back-end guarantees a minimum gross bandwidth of 660 MB/s. We keep the same use-case as in the earlier experiments, but we scale up the request sizes to fit with the larger access granularity of the SDRAM back-end. All request sizes remain an integer multiple of the access granularity of the memory, resulting in a data efficiency of 100%, making gross and net bandwidth the same. We furthermore increase the allocated rates in response to the reduced gross bandwidth provided by the memory to satisfy the bandwidth requirements of the requestors. The total allocated net bandwidth equals 98.8% of what is provided by the SDRAM memory, indicating a high load. The use-case in this experiment, shown in Table 6.2, is hence identical to what we previously used in Sect. 5.7. The service latencies, Θ_r^{cc}, and completion latencies, l_r^{cc}, expressed in clock cycles, are computed according to Definitions 6.4 and 6.5, respectively. The completion latencies are determined based on the average service cycle length using the considered memory pattern, corresponding to $\bar{\lambda} = 16/0.825 = 19.4$ clock cycles. The intuition behind this value is that our pattern set is mix read dominant, causing the worst-case bandwidth

Table 6.2 SDRAM use-case specification and configuration

Requestor	Type	b_r (MB/s)	Size (B)	p_r	σ''_r (su)	ρ''_r (su/sc)	Θ^{cc}_r (cc)	l^{cc}_r (cc)
r_0	Read	210.0	512	0	1.0	0.319	88	60.8
r_1	Write	210.0	128	1	1.0	0.319	106	60.8
r_2	Read	210.0	64	2	1.0	0.319	182	60.8
r_3	Write	20.0	256	3	1.0	0.031	1418	621

and latencies to be provided with alternating read and write requests. For our pattern set, $t_{wtr} + t_{read} = 18$ clock cycles and $t_{rtw} + t_{write} = 20$ clock cycles, resulting in an average of 19 clock cycles. The remaining 0.4 clock cycles in $\bar{\lambda}$ accounts for refresh by ensuring that 32 clock cycles can be lost once every 1560 when the memory needs to refresh. Note that the completion latencies are much longer for SDRAM memories with large access granularities, making the discrete approximation mechanism less significant. Rounding the completion latencies of r_0, r_1, and r_2 upwards, reduces their provided bandwidths by approximately 1 MB/s, and the provided bandwidth of r_3 by just a couple of KB/s. The service latencies include a latency offset of 32 clock cycles to compensate for the use of the average service cycle time when computing the completion latency.

6.4.2.1 General Observations

For our first experiment with SDRAM, we simulate the use-case during 100 ms to make some general observations about the behavior of the front-end and the back-end. The results of this simulation for the first 200 requests of r_2 are shown in Fig. 6.13. There are two interesting differences compared to the results for the SRAM controller, previously shown in Fig. 6.9. The first difference is that the bound on finishing time is flatter, indicating less self interference. This is explained by that requests are generated with the same variance in both cases, although the average time between generated requests increases with the request size. Requests in this use-case are hence generated in a less bursty fashion. The second difference is that the number of clock cycles between the starting times and finishing times is no longer constant, but varies between 20 and 54 clock cycles with an average of approximately 24. This variation is explained by the introduction of read/write switches and refreshes. The effects of read/write switches are difficult to see, since they are in the range of a few clock cycles. The 54 cycle difference due to refresh is somewhat more noticeable, although it only happens approximately once per 100 requests. In Fig. 6.13, there is interference from refresh for request 50 and request 178.

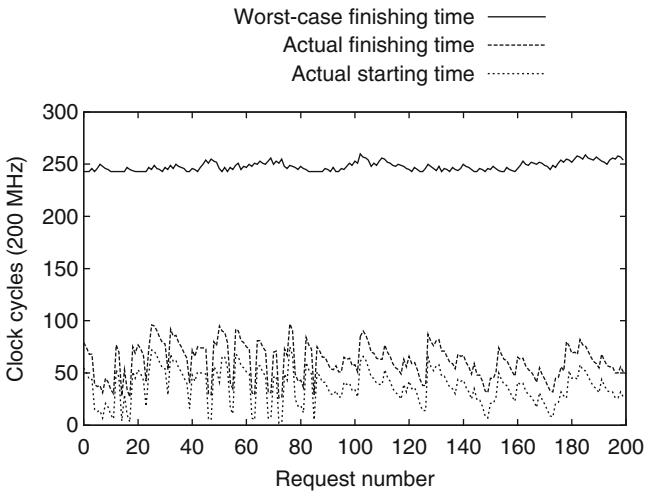

Fig. 6.13 The first 200 requests of r_2 in the SDRAM use-case

6.4.2.2 Tightness of Bound on Finishing Time

When looking at the bounds on finishing times in Fig. 6.13, we note that they seem less tight than for the SRAM in Fig. 6.9. In fact, the minimum difference between the actual finishing times and the corresponding bounds is 134 clock cycles. The same reasons for the bound not being tight in the case of the SRAM still applies to the case of SDRAM. Increasing the allocated rates of the requestors furthermore increases the possible interference from other requestors, according to (5.14). The bound on interference from other requestors is 150 clock cycles, although the arbiter maximally measures 70 clock cycles, excluding blocking, leaving 60–80 out of the 150 clock cycles of interference unaccounted for. Blocking is over-estimated by one clock cycle, although this is negligible in the case of SDRAM, where latencies are much longer due to the larger access granularity. The impact of computing the finishing time to be one completion latency after the starting time, as opposed to one average service cycle, is quite significant also in this case. The completion latency is 61 clock cycles, whereas the average service cycle length rounds up to 20 clock cycles, accounting for another 40 cycles of missing interference. For the case of SDRAM, there are also two new reasons why the bound is not tight. The first reason is that the bound assumes the maximum number of interfering refreshes and read/write switches, where the actual case may contain less. The second reason is the added latency offset, which is very pessimistic unless the actual service latency is very close to a multiple of $tREFI$. Together, all these factors contribute to the bound not being tight. *We conclude that the bound on finishing time for SDRAM is conservative, but less tight than the bound for SRAM, due to the extra uncertainties introduced by the variable service cycle length.*

6.4.2.3 Added Latency and Buffering

The fact that the bounds on finishing time are less tight for the SDRAM than the SRAM implies that the added latency and buffering by delaying requests increase. The average actual finishing time and average worst-case finishing time of r_2 in this use-case are 49 and 254 clock cycles after the corresponding arrival times, respectively. Delaying responses hence increases the average latency of r_2 by a factor 4.2 in this use-case. However, this added latency can be reduced by at least 25% by not using the completion latency to compute the bound. It is also important to consider that generating the requests in a burstier manner may increase the average actual completion latency, while the bound remains unaffected, hence reducing the cost in terms of added average latency. The Response Buffer of the read requestors has a maximum filling of one command and 16 data words, corresponding to a single atom, when responses are not delayed. Enabling emulation of worst-case interference increases the Response Buffer filling to four atoms for r_0 and slightly less than six atoms for r_2. *We conclude that the larger access granularity and looser latency bound when using SDRAM increases the relative cost in terms of average latency and required buffer capacity compared to SRAM when enabling composable service.*

6.4.2.4 Composable SDRAM Controller

For our second experiment with SDRAM, we demonstrate that the time requests are released from the Delay Block and the worst-case Response Buffer space are independent of other requestors. We follow the same procedure as with the SRAM controller and simulate the use-case twice, changing the variance in the request generation of r_0 between the runs. The impact of this change on r_2 is shown in Fig. 6.14. We note that the actual finishing times and Response Buffer filling change, while the worst-case values emulated by the Delay Block are unaffected. *This leads us to conclude that the service provided by the resource front-end combined with our SDRAM back-end is composable.*

6.5 Summary

A predictable memory controller enables formal verification of latency and throughput requirements of applications. However, this requires a performance model of the application, which is not always available. A complementary verification approach based on simulation of *composable systems* is proposed in this chapter. Applications in a composable system cannot affect each other's temporal behavior by even a single clock cycle. *This enables independent verification of applications, reducing the verification effort.* Existing approaches to composable system design are either restricted to applications that can be statically scheduled, or share

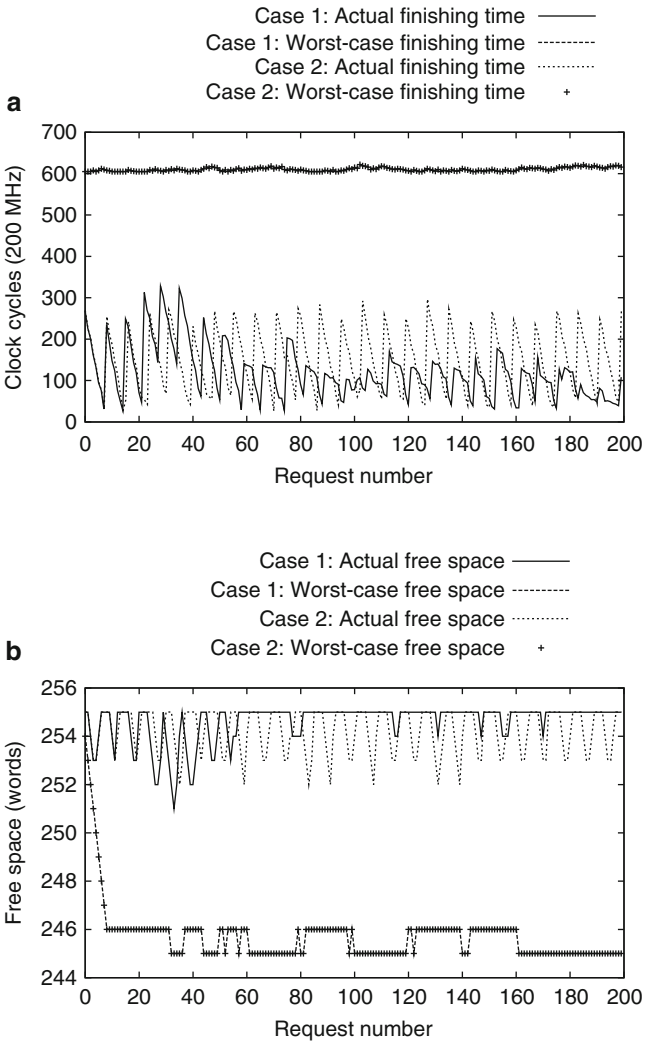

Fig. 6.14 SDRAM controller behaving in a composable manner. (**a**) Request releases are unaffected by other requestors. (**b**) Worst-case Response Buffer space is unaffected by other requestors

inherently composable resources using Time-Division Multiplexing (TDM), which cannot efficiently satisfy the requirements of latency-sensitive requestors. Neither of these approaches apply to an arbitrary application in a platform with our proposed SDRAM back-end or priority-based arbitration.

This chapter proposes a new approach to composable resource sharing that applies to *any combination* of predictable resource and Latency-Rate ($\mathcal{LR}$) arbiter without any restrictions on the application. The key idea is to delay all signals sent

from the resource to a requestor by *emulating worst-case interference from other requestors*. This makes the system composable at the level of requestors, which is a sufficient condition for it to be composable at the level of applications. Our approach supports providing composable service to a subset of the requestors by dynamically enabling or disabling emulation of worst-case interference. *This enables slack bandwidth to be used to improve the performance of requestors that do not require composable service.* Providing composable service in our approach requires the starting times and finishing times of requests to be dynamically computed. We showed how to compute these in a manner that is efficient also for resources with variable service cycle length, such as our SDRAM back-end.

The ideas presented in this chapter are implemented as a *composable resource front-end* that is placed in front of the predictable resource. The architecture of the front-end has three main building blocks: (1) an *Atomizer*, (2) a *Delay Block*, and (3) a *Data Bus with an arbiter in the class of $\mathcal{LR}$ servers*. The Atomizer chops requests into smaller pieces fitting with the access granularity of the resource and merges responses into the expected size. This prevents malfunctioning requestors from violating latency guarantees of others by sending large requests, and simplifies the implementation of the rest of the architecture. The Delay Block makes the front-end composable by delaying signals to emulate worst-case interference from other requestors. The Data Bus schedules requests for resource access according the policy of its attached $\mathcal{LR}$ arbiter. It was experimentally demonstrated that the resource front-end fitted with a Credit-Controlled Static-Priority (CCSP) arbiter provides composable service when combined with both an SRAM controller and our SDRAM back-end. Based on our experiments, we concluded that the bounds on finishing times are conservative, but not tight. We also demonstrated the benefits of distributing slack bandwidth to requestors that do not require composable service by disabling the delays and using a work-conserving arbiter.

Chapter 7
Configuration

Our journey towards a predictable and composable memory controller is approaching its end. A predictable SDRAM back-end has been presented that enables net bandwidth and latency to be bounded. The memory controller architecture was completed by a front-end that enables the SDRAM back-end, or any other predictable resource, to be shared among multiple requestors in a predictable and composable manner using an arbiter in the class of Latency-Rate ($\mathcal{LR}$) servers. While presenting this architecture, a number of instantiation parameters and configuration settings were mentioned. The remaining problem is to automatically derive these parameters and settings, such that the requestor requirements are satisfied, thus delivering on our automation requirement.

This chapter introduces a configuration flow that automatically derives architecture parameters and configuration settings, given requestor requirements and a specification of the memory and arbiter. The discussion is structured around the different steps in the configuration flow, illustrated in Fig. 7.1, and relies heavily on results from earlier chapters. This chapter hence acts as a summary that brings the pieces together to satisfy requestor requirements. First in Sect. 7.1, we formalize requestor requirements and define a metric that is used to evaluate the quality of a given configuration. We then proceed by walking through each of the steps in the configuration flow in Sects. 7.2–7.6. A running example is used throughout these sections to clearly illustrate what happens in the different steps of the flow. The flow is experimentally evaluated with a large number of use-cases in Sect. 7.7, before the chapter is concluded with a summary in Sect. 7.8.

7.1 Formal Model

The discussion in this chapter is focused around satisfying requestor requirements, making it prudent to include these in our formal model. Note that a complete list of the symbols in the formal model is found in Appendix B.2 along with a brief descriptions and page references to the definitions.

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_7, © Springer Science+Business Media, LLC 2011

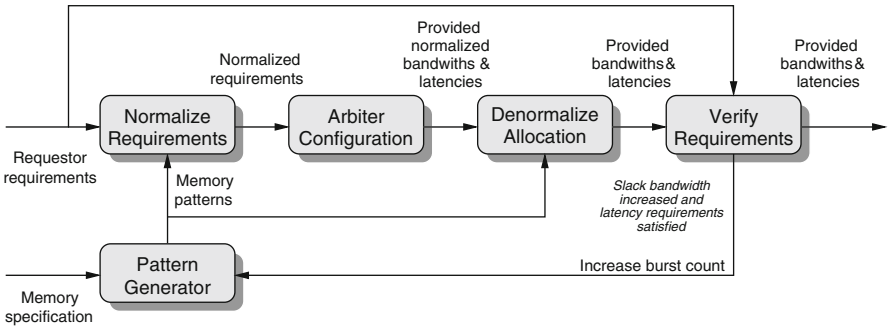

Fig. 7.1 Overview of the automated configuration flow

Definition 7.1 states that a requestor requires a maximum service latency, $\hat{\Theta}^{cc}$, measured in clock cycles, and a required minimum net bandwidth, b. These requirements are provided by the first part of the mapping process, discussed in Sect. 1.1.4, which is considered outside the scope of this book. The requirements of a requestor are assumed to be derived in different ways depending on its real-time classification. The requirements of hard and firm real-time requestors are assumed to be derived based on a conservative model of the application that guarantees that the application requirements are satisfied if the requestor requirements are met. Soft real-time applications are often more complex than their hard and firm real-time counterparts and a conservative application model may hence not exist. Soft real-time requirements may hence be derived based on estimates or simulation that suggests that the application meets its real-time requirements often enough to be considered useful. The requirements of non-real-time requestors, on the other hand, just have to be derived in a way that makes the application seem responsive to the user. A non-real-time requestor may hence have an infinite latency requirement and use only slack bandwidth. It may not be possible to derive suitable requirements if there is no model of the application. For this reason, it is possible to side-step parts of the configuration flow and manually configure a requestor. This can be used as a fall-back mechanism together with simulation-based verification to ensure that all applications meet their requirements.

Definition 7.1 (Requestor requirements). The requirements of a requestor $r \in R$ are denoted by $(\hat{\Theta}_r^{cc}, b_r)$, where $\hat{\Theta}_r^{cc}$ is an upper bound on service latency in clock cycles and b_r the requested net bandwidth in MB/s.

We proceed in Definition 7.2 by defining a use-case as valid if three requirements are satisfied: (1) The allocated bandwidths for all requestors, b'_r, must be at least as large as the requested bandwidths, b_r. (2) The provided service latencies, Θ_r^{cc}, cannot be larger than the corresponding bounds, $\hat{\Theta}_r^{cc}$. (3) The requestors cannot be allocated more bandwidth than what is provided by the memory. The goal of the flow is to derive instantiation parameters and settings for the memory controller, such that all use-cases are valid. However, the current implementation of the configuration

Table 7.1 Use-case specification

Requestor	Type	b_r (MB/s)	Size (B)	$\hat{\Theta}_r^{cc}$ (cc)
r_0	Read	210.0	512	300
r_1	Write	210.0	128	110
r_2	Read	210.0	64	90
r_3	Write	20.0	256	200

flow is limited to a single use-case. Generalizing the flow to remove this limitation is considered important future work. There may be many configurations that result in a valid use-case. In this case, the configuration with maximum slack bandwidth, defined in Definition 7.3, is preferred. The rationale behind this decision is that slack bandwidth can be used to improve the performance of requestors that do not require composable service, as previously shown in Sect. 6.4. Note that slack bandwidth is computed based on gross bandwidth, since the net usage depends on the request size of the requestor the slack is allocated to. Definition 7.3 hence converts the allocated net bandwidths to gross allocations using the data efficiencies of the requestors.

Definition 7.2 (Valid use-case). A use-case is defined as valid iff $\forall r \in R : \hat{\Theta}_r^{cc} \geq \Theta_r^{cc} \wedge b'_r \geq b_r \wedge \sum_{\forall r \in R} b'_r \leq b^{net}$.

Definition 7.3 (Slack bandwidth). The slack bandwidth in a use-case is defined as $b^{slack} = b^{gross} - \sum_{\forall r \in R} b'_r / e_r^{data}$.

The configuration flow will be demonstrated step by step using an example use-case. For this purpose, we revisit the use-case with four requestors previously used in Sects. 5.7 and 6.4. Service latency requirements are added to the use-case, as shown in Table 7.1, to provide a starting point for the configuration flow. The considered memory controller is using our resource front-end fitted with a Credit-Controlled Static-Priority (CCSP) arbiter and an SDRAM back-end interfacing to our example 16-bit DDR2-400 memory. Both the CCSP arbiter and the approximation mechanism in the Delay Block use six bits of precision to represent the allocated service and the fractional part of the completion latency, respectively. The proposed configuration flow runs at design time and only configures the memory controller. The configuration of the network-on-chip is covered in [47].

7.2 Memory Pattern Generation

The first step of the configuration flow is to generate a set of memory patterns. For an SDRAM controlled by our proposed back-end, any of the algorithms presented in Sect. 4.5 can be used. However, we have chosen to integrate the bank scheduling algorithm into our tool flow, since it provides a favorable trade-off between run-time and memory efficiency, as experimentally shown in Sect. 4.7.

The memory architecture and timings, previously defined in Definitions 3.1 and 3.2 are supplied as inputs to the memory pattern generation algorithm. These are provided as parts of a system architecture specification file, shown in Appendix A.

Table 7.2 Output from pattern generation stage

t_{read} (cc)	t_{write} (cc)	t_{rtw} (cc)	t_{wtr} (cc)	t_{ref} (cc)	$\check{\lambda}$ (cc)	g (B)
16	16	2	4	32	16	64

The final input to the pattern generation is the burst count, although this is supplied automatically by the configuration flow, since the optimal burst count is not known up front. Larger burst count results in more gross bandwidth, as previously shown in Sect. 4.7. Increasing burst count thus provides an opportunity to create more slack bandwidth, and hence a better configuration. However, it was also shown that increasing the burst count increases access granularity, potentially reducing net bandwidth if request sizes are not sufficiently large. Increasing burst count furthermore increases the length of the access patterns, making it more difficult to satisfy latency requirements. In our configuration flow, this is addressed by starting to generate patterns with $BC = 1$ and later visit other options by iteration in the flow. This is further explained in Sect. 7.6.

No memory patterns are needed if the memory is an SRAM, controlled by an off-the-shelf SRAM controller. However, a pattern specification is generated that describes the characteristics of accesses to the memory. For example, for a Zero-bus-turnaround (ZBT) SRAM, we set $t_{read} = 1$, $t_{write} = 1$, $t_{wtr} = 0$, $t_{rtw} = 0$, and $t_{ref} = 0$, reflecting that this memory reads or writes a burst of one word in a single clock cycle. It furthermore does not require any time to switch from reads to writes, and a refresh is performed in zero time. Similar specifications are straight-forwardly derived for SRAMs with larger burst lengths or that requires a few clock cycles to change direction of the data bus. The advantage with this type of specification is that it abstracts from the detailed timing behavior of the memory, allowing the same configuration flow to be used with several different memory types. This approach fits well with our abstraction requirement. Currently, we use the configuration flow with both SRAM and SDRAM, although we believe that the pattern specification is general enough to also cover other types of memories, such as flash.

The memory pattern generation step determines the set of patterns that should be implemented in the Command Generator of the SDRAM back-end. The pattern specification determines two instantiation parameters used in the resource front-end: (1) The minimum service cycle time, $\check{\lambda}$, used by the Data Bus to determine when the next scheduling decision should be made. This parameter is computed according to (6.1). (2) The access granularity of an access pattern, g, which is the atom size used by the Atomizer, is calculated according to Definition 4.2.

Applying the memory pattern generation step to our example use-case and system results in the output shown in Table 7.2. The generated pattern set with $BC = 1$ is the same as we previously generated for this memory with the bank scheduling algorithm in Sect. 4.7. This pattern set has an access granularity of 64 B and provides a gross bandwidth of 660 MB/s. The minimum service cycle length is 16 clock cycles, resulting from either a read pattern or a write pattern, since they are equally long.

7.3 Normalization of Requirements

The second step in the configuration flow is to normalize the requestor requirements, thereby making them independent of the target memory. The advantage of this abstraction is that it makes the choice of memory *completely transparent* to the arbiter configuration step, allowing *any* supported arbiter to be configured for *any* supported memory in a streamlined fashion. The requirements are normalized by converting the requirements to service units according to Definition 7.4. The requestor requirements are provided by the user as an input to this stage using a use-case specification file, shown in Appendix A. The second input is the description of the generated memory pattern set.

Definition 7.4 (Normalized requestor requirements). The normalized requirements of a requestor $r \in R$ are defined as $(\hat{\Theta}_r, \rho_r)$, where $\hat{\Theta}_r$ is an upper bound on service latency in service cycles and ρ_r the requested service rate.

Definition 6.4 states how to convert a service latency expressed in service units to clock cycles. Since normalizing the service latency requirement is the inverse of this operation, we proceed by inverting the expression and solving for the service latency in service cycles. Equation (7.1) starts the inversion by stating that $\hat{\Theta}^{cc} \geq \Theta^{cc}$ and solving for the pattern dominant expression $t_{aux}(\hat{\Theta}+1)$. The inversion is conservative, but somewhat pessimistic, since it adds up to an additional refresh, t_{ref}, to the worst-case latency when removing a ceiling.

$$\begin{aligned}
\hat{\Theta}^{cc} \geq \Theta^{cc} = t_{tot}(\hat{\Theta}) + \Delta + n_{pipe} &= \left\lceil \frac{t_{aux}(\hat{\Theta}+1)}{tREFI - t_{ref} - t_{block}} \right\rceil \cdot t_{ref} + t_{aux}(\hat{\Theta}+1) \\
+\Delta + n_{pipe} &\geq \left(\frac{t_{aux}(\hat{\Theta}+1)}{tREFI - t_{ref} - t_{block}} + 1 \right) \cdot t_{ref} + t_{aux}(\hat{\Theta}+1) \\
+\Delta + n_{pipe} &= \frac{t_{aux}(\hat{\Theta}+1) \cdot t_{ref}}{tREFI - t_{ref} - t_{block}} + t_{ref} + t_{aux}(\hat{\Theta}+1) \\
+\Delta + n_{pipe} &= t_{aux}(\hat{\Theta}+1) \cdot \left(1 + \frac{t_{ref}}{tREFI - t_{ref} - t_{block}} \right) + t_{ref} \\
+\Delta + n_{pipe} &\Rightarrow \frac{\hat{\Theta}^{cc} - t_{ref} - \Delta - n_{pipe}}{1 + \frac{t_{ref}}{tREFI - t_{ref} - t_{block}}} \geq t_{aux}(\hat{\Theta}+1)
\end{aligned} \tag{7.1}$$

We proceed by solving for $\hat{\Theta}$ for the different dominance classes according to the different cases in (4.7). Equations (7.2) and (7.3) derive upper bounds on $\hat{\Theta}$ for read-dominant patterns and mix-read-dominant patterns, respectively. The cases of write-dominant and mix-write-dominant patterns are derived in the same manner, but with t_{read} switched for t_{write} and t_{wtr} switched for t_{rtw}. Equation (7.3) introduces an over-estimation of the worst-case latency when removing the ceiling and floor

operations. The pessimism introduced by the inversion is an added cost in terms of latency attributed to our choice to use abstraction to decouple the configuration of the memory and the arbiter.

$$\begin{aligned}
&\frac{\hat{\Theta}^{cc} - t_{ref} - n_{pipe}}{1 + \frac{t_{ref}}{tREFI - t_{ref} - \Delta - t_{block}}} \geq t_{aux}(\hat{\Theta}+1) = (\hat{\Theta}+1) \cdot t_{read} + t_{wtr} \\
&\Rightarrow \frac{\hat{\Theta}^{cc} - t_{ref} - \Delta - n_{pipe}}{1 + \frac{t_{ref}}{tREFI - t_{ref} - t_{block}}} - t_{read} - t_{wtr} \geq \hat{\Theta} \cdot t_{read} \\
&\Rightarrow \left(\frac{\hat{\Theta}^{cc} - t_{ref} - \Delta - n_{pipe}}{1 + \frac{t_{ref}}{tREFI - t_{ref} - t_{block}}} - t_{read} - t_{wtr} \right) \cdot \frac{1}{t_{read}} \geq \hat{\Theta}
\end{aligned} \tag{7.2}$$

$$\begin{aligned}
&\frac{\hat{\Theta}^{cc} - t_{ref} - \Delta - n_{pipe}}{1 + \frac{t_{ref}}{tREFI - t_{ref} - t_{block}}} \geq t_{aux}(\hat{\Theta}+1) = \left\lceil \frac{\hat{\Theta}+1}{2} \right\rceil \cdot (t_{wtr} + t_{read}) + \left\lfloor \frac{\hat{\Theta}+1}{2} \right\rfloor \\
&\quad \cdot (t_{rtw} + t_{write}) \geq \left(\frac{\hat{\Theta}+1}{2} + 1 \right) \cdot (t_{wtr} + t_{read}) + \left(\frac{\hat{\Theta}+1}{2} \right) \cdot (t_{rtw} + t_{write}) \\
&= \frac{\hat{\Theta}+3}{2} \cdot (t_{wtr} + t_{read}) + \frac{\hat{\Theta}+1}{2} \cdot (t_{rtw} + t_{write}) \Rightarrow \frac{\hat{\Theta}^{cc} - t_{ref} - \Delta - n_{pipe}}{1 + \frac{t_{ref}}{tREFI - t_{ref} - t_{block}}} - \frac{3}{2} \\
&\quad \cdot (t_{wtr} + t_{read}) - \frac{1}{2} \cdot (t_{rtw} + t_{write}) \geq \hat{\Theta} \cdot \frac{t_{wtr} + t_{read} + t_{rtw} + t_{write}}{2} \\
&\Rightarrow \hat{\Theta} \leq \frac{\frac{\hat{\Theta}^{cc} - t_{ref} - \Delta - n_{pipe}}{1 + \frac{t_{ref}}{tREFI - t_{ref} - t_{block}}} - \frac{3}{2} \cdot (t_{wtr} + t_{read}) - \frac{1}{2} \cdot (t_{rtw} + t_{write})}{\frac{t_{wtr} + t_{read} + t_{rtw} + t_{write}}{2}}
\end{aligned} \tag{7.3}$$

The pattern specification for a ZBT SRAM is technically mix-read-dominant according to Definition 4.8, and is hence normalized using (7.3). This specification is $t_{read} = 1$, $t_{write} = 1$, $t_{wtr} = 0$, $t_{rtw} = 0$, and $t_{ref} = 0$, reducing the equation to $\hat{\Theta} \leq \hat{\Theta}^{cc} - n_{pipe} - 2$. This is an over-estimation of two clock cycles, which is quite an acceptable loss for a streamlined configuration flow. However, it can easily be eliminated by treating SRAM as a special case.

The normalized bandwidth requirement of a requestor, ρ, is a service rate that represents the required fraction of the total available service units provided by the memory. However, the size of a service unit equals the access granularity of the resource, which may be larger than the request size of the requestor. This problem of data efficiency must hence be addressed in the normalization to ensure that the net bandwidth requirement of the requestor is satisfied. This is done by converting

Table 7.3 Output from normalization stage

Requestor	$\hat{\Theta}_r$ (sc)	ρ_r (su/sc)
r_0	10	0.318
r_1	1	0.318
r_2	0	0.318
r_3	5	0.030

the net bandwidth requirement of a requestor into a gross requirement by scaling it with the requestors data efficiency, previously computed in (4.6). The intuition behind this is that a requestor that cannot use half of the data in a service unit hence requires twice as many service units to satisfy its requirements. This implies that the normalized bandwidth requirement may increase with burst count, since a larger access granularity results in lower data efficiency, as previously shown in (4.6). The normalized bandwidth requirement is computed according to (7.4). This equation reduces to $\rho_r = \frac{b_r}{b^{peak}}$ for the ZBT SRAM pattern specification, since all categories of memory efficiency are 100%.

$$\rho_r = \frac{b_r}{e_r^{data} \cdot b^{gross}} = \frac{b_r}{e_r^{data} \cdot e^{gross} \cdot b^{peak}} \tag{7.4}$$

The results of normalizing the requirements in our example use-case are shown in Table 7.3. Note that the service latency requirement of r_2 is zero service cycles. This suggests that it is not possible to satisfy much lower latency requirements than its 90 clock cycles with our example memory due to three factors: (1) unavoidable interference from refresh of t_{ref} clock cycles for every started $tREFI$ clock cycles, (2) the service latency offset, Δ, and (3) the overhead introduced by the pessimistic inversion of the requirement. However, the requirements in service cycles scale better with the requirements in clock cycles after the first service cycle, as only effects of refresh recur as requirements in clock cycles increase. This is seen as the latency requirement of 100 clock cycles turns into a requirement of 1 service cycle for r_1, while 200 clock cycles results in a requirement of 5 service cycles for r_3. Computing the normalized bandwidth requirement is quite straight forward, since the access granularity of the memory pattern is 64 B. All request sizes are hence integer multiples of the access granularity for this burst count, making gross and net bandwidth equal. Normalizing the results shows that 98.5 % of the available gross bandwidth is required by the requestors.

7.4 Arbiter Configuration

The arbiter configuration is computed after the requestor requirements have been normalized. Due to the normalization, the arbiter configuration is completely independent of the memory. The implementation of this step depends on the particular arbiter, which is specified in the system architecture specification provided by the user. It is possible for the user to side-step the arbiter configuration for a subset of the

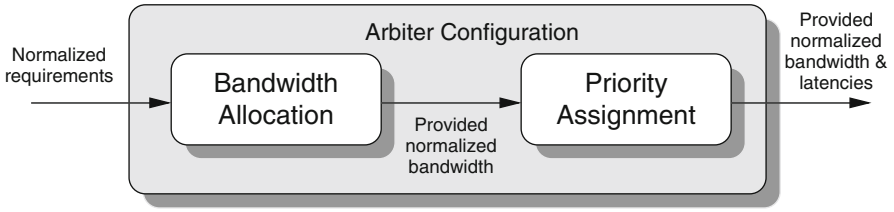

Fig. 7.2 Configuration of CCSP and FBSP consists of a bandwidth allocation step and a priority assignment step

requestors by manually entering configuration settings in the use-case specification file. As mentioned in Sect. 7.1, this enables the user to manually search for suitable settings if requestor requirements cannot be derived for an application. We proceed by showing how the configuration is done for the CCSP and Frame-Based Static-Priority (FBSP) arbiters. We split the configuration of these arbiters into two steps, bandwidth allocation and priority assignment, as shown in Fig. 7.2 and solve the problem according to a waterfall approach. Decomposing the problem in this manner has the advantage of making it easier to solve at the expense of possibly not finding a valid configuration even if one exists. The two steps are discussed in more detail in Sects. 7.4.1 and 7.4.2, respectively.

7.4.1 Bandwidth Allocation

The input to this step is the normalized bandwidth requirements of the requestors, ρ_r. The bandwidth allocation step needs to perform two tasks. The first task is to determine the allocated normalized bandwidths (allocated service rates), $\rho'_r \geq \rho_r$, for the requestors. Secondly, it has to find arbiter-specific settings to allocate these bandwidths. We discuss each of these tasks in turn.

The first task is addressed in a very simple way by assigning $\rho'_r = \rho_r$ for all requestors. This ensures that each requestor has their bandwidth requirement satisfied, assuming they do not request more bandwidth than is offered by the resource in total. This is checked in the verification step at the end of the flow. A limitation with this approach is that it does not consider the fact that bandwidth allocation may affect the ability to satisfy latency requirements. Reserving additional bandwidth reduces the latency of a requestor if bandwidth and latency are coupled, such as in the case of Time-Division Multiplexing (TDM). However, allocating additional bandwidth to a requestor *increases* the latency of lower priority requestors in priority-based schemes like CCSP and FBSP. For simplicity, we choose to keep these steps decoupled, although we consider improvements in arbiter configuration an important part of future work.

The second task is to find arbiter settings that provide the allocated bandwidth. For frame-based arbiters, such as TDM or FBSP, this involves finding the number of

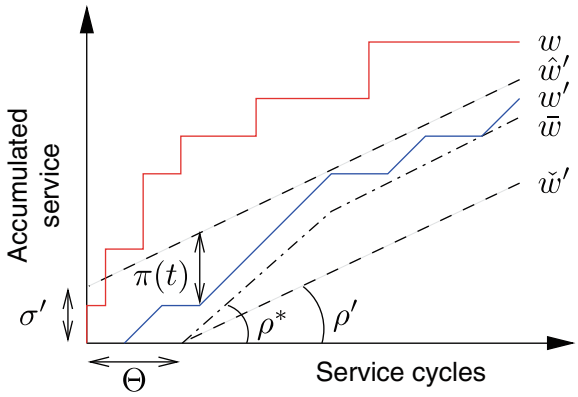

Fig. 7.3 $\mathcal{LR}$ servers cannot capture service provided with multiple rates to a requestor

slots, ϕ_r, guaranteed to a requestor in a frame of size f. The frame size is manually chosen to balance the conflicting requirements of providing low latency and over-allocating, discussed in Chap. 5. The frame size is included as an input to this step through the system specification, listed in Appendix A, if FBSP is used. Given a frame size, bandwidth is allocated with FBSP by letting $\phi_r = \lceil \rho'_r \cdot f \rceil$, as discussed in Sect. 5.5.

Unlike FBSP, bandwidth allocation with CCSP considers two parameters per requestor, (σ'_r, ρ'_r), as previously explained in Sect. 5.6. These parameters do not only reserve a particular bandwidth, but also explicitly define the maximum deviation from this value through the allocated burstiness. In contrast, this value is implicitly allocated for a requestor with FBSP when the slots are reserved. CCSP requires that $\sigma' \geq 1$ for a requestor to act as a $\mathcal{LR}$ server. The benefit of assigning $\sigma' > 1$ is that a requestor r_i is temporarily served with a higher rate $\rho^*_{r_i} = 1 - \left(\sum_{\forall r_j \in R^+_{r_i}} \rho'_{r_j}\right) \geq \rho'_{r_i}$, while $\pi(t)_{r_i} \geq 1 - \rho'_{r_i}$, as shown in Fig. 7.3. In essence, this means that a lower priority requestor does not receive service while a higher priority requestor is eligible. Assigning $\sigma' > 1$ increases the time eligible high-priority requestors enjoys service at the higher rate at the expense of increased service latency of the lower priority requestors. The increase in service latency is visible in (5.14) through the dependence on the allocated burstiness of higher priority requestors. The temporarily higher service rate, ρ^*, can be used to reduce the worst-case finishing time of a request, since it results in an improved lower bound on provided service in an interval, $\bar{w}$, as shown in Fig. 7.3. However, this is not captured by the $\mathcal{LR}$ server model, which assumes a constant service rate ρ' after a service latency Θ. This issue is addressed in [112], where the allocated rate and the temporary higher service rate are used to derive an improved service guarantee for the CCSP arbiter. This bi-rate service model enables throughput requirements of applications to be satisfied with less allocated bandwidth. However, we do not consider this model further in this book, since the $\mathcal{LR}$ server model is chosen as the shared resource abstraction. We configure $\sigma' = 1$ for all requestors, since limitations of the chosen service model implies that assigning $\sigma' > 1$ to a requestor increases

Table 7.4 Results from the bandwidth allocation stage

Requestor	σ_r'' (su)	ρ_r''	n_r (su/sc)	d_r	$c_r(0)$
r_0	1.0	0.319	15	47	47
r_1	1.0	0.319	15	47	47
r_2	1.0	0.319	15	47	47
r_3	1.0	0.0312	1	32	32

the service latency of lower priority requestors without reducing worst-case latency or worst-case finishing times of the requestor itself.

Once the allocation parameters have been determined, they are discretized to fit with the allocation granularity of the arbiter. For CCSP, this involves computing the three parameters n_r, d_r, and $c_r(0)$ that approximate the service allocation of a requestor given a particular precision, as previously discussed in Sect. 5.6. Based on these parameters, the discrete allocated burstinesses, σ_r'', and the discrete allocated rates, ρ_r'', of the requestors are determined. This enables the over-allocated rates and over-allocated burstinesses to be computed by Definitions 5.10 and 5.25, respectively.

The results of allocating bandwidth in our use-case for a CCSP arbiter are shown in Table 7.4. Choosing σ_r' to be integers implies that there is no discretization of the allocated burstinesses and hence that $\sigma_r'' = \sigma_r' = 1.0$. From this, it furthermore follows from Definition 5.26 that $c_r(0) = d_r$. Allocating the service rates with a precision of six bits results in an over-allocated rate of 0.4%. The total allocated service of all requestors, including over-allocation, is hence 98.9% of the available service units.

7.4.2 Priority Assignment

Priorities are assigned using the optimal priority assignment algorithm proposed in [15]. This algorithm is reproduced in Algorithm 7.1, based on the implementation in [123]. Note that $|R|$ represents the number of elements in R. The algorithm first finds a requestor that meets its service latency requirement with the lowest priority. If such a requestor is found, it is assigned the lowest priority. If multiple requestors are found, a choice between them can be made arbitrarily. This procedure is then repeated for the next higher priority. The algorithm terminates either if all priorities are assigned, indicating that a valid priority assignment has been found, or if none of the remaining requestors can meet their service latency requirement at a particular level, indicating failure. It is shown in [15] that this algorithm has a quadratic time complexity and is optimal in the sense that it is guaranteed to find a successful priority assignment if one exists. For the algorithm to be correct, it is required that the service latency is monotonically non-increasing with decreasing priority level, meaning that giving a requestor higher priority may not result in increased service latency. This assumption holds for both the service latency equations of FBSP and CCSP, previously shown in (5.9) and (5.14), respectively. Priority assignment

Table 7.5 Results from priority assignment stage

Requestor	p_r	Θ_r (sc)	$\hat{\Theta}_r$ (sc)
r_0	3	9	10
r_1	1	1	1
r_2	0	0	0
r_3	2	5	5

concludes the arbiter configuration for both CCSP and FBSP, since all configuration settings have been derived. The configuration settings are stored pending final approval in the last step of the flow. The discrete allocated service rates, ρ_r'', and the service latencies, Θ_r, are output from the arbiter configuration.

Algorithm 7.1 Optimal priority assignment algorithm.

```
prio ← |R| − 1
repeat
    finished ← false
    failed ← true
    j ← 0
    repeat
        assign priority prio to r_j
        if Θ_{r_j} ≤ Θ̂_{r_j} then
            prio ← prio - 1
            failed ← false
            finished ← true
        else
            restore old priority of r_j
        end if
        j ← j + 1
    until finished or j = prio - 1
until prio = 0 or failed
```

The priority assignment for our use-case is shown in Table 7.5. Priorities happen to be assigned according to the tightness of the latency requirements, which seems intuitive, but does not always result in the best solution. For example, a requestor with high allocated rate and burstiness may significantly increase the latency of a requestor with a low service allocation if given high priority. However, the requestor with the low service allocation cannot significantly interfere with others if given high priority, making this an interesting option, even if the latency requirement of the higher priority requestor is tighter.

7.5 Denormalization of Allocation

This step receives the discrete allocated rates and service latencies along with the generated memory patterns and transforms it from the normalized domain with service units and service cycles to the domain of bytes and clock cycles. The

Table 7.6 Output from denormalization stage

Requestor	b'_r (MB/s)	Θ_r^{cc} (cc)	l_r^{cc} (cc)	n_r^*	d_r^*
r_0	210.5	258	60.8	11	50
r_1	210.5	106	60.8	11	50
r_2	210.5	88	60.8	11	50
r_3	20.6	182	621	15	56

service latencies are converted to clock cycles using Definition 6.4, and the discrete allocated rates to net bandwidths in MB/s by (7.5), which is the inverse of (7.4) used during the normalization process.

$$b'_r = \rho''_r \cdot e_r^{data} \cdot b^{gross} \tag{7.5}$$

The service latency in clock cycles is one of the four configuration settings for the Delay Blocks. The other three settings are related to the completion latency. First, there is the integer part, l_r^{cc}, which is computed according to Definition 6.5 and is rounded up when programmed. This is followed by the two numbers, n_r^* and d_r^*, that are used to approximate the fractional part. These two parameters are chosen to get tightest possible approximation of the exact completion latency. The denormalized allocated bandwidths and service latencies, (Θ_r^{cc}, b'_r), are forwarded to the next step in the flow.

Denormalizing the arbiter configuration of our running example using six bits to represent the fractional parts of the completion latency, gives us the results in Table 7.6. The denormalized bandwidths show the actual meaning of the over-allocation resulting from discretization in the CCSP arbiter. The over-allocated bandwidths are quite modest, indicating that the chosen precision is suitable for this use-case. We observe that the completion latency of r_3 is 621 clock cycles, which seems to be a rather long time. This completion latency follows naturally from the low bandwidth allocation of the requestor. Service latency is decoupled from rate using priorities, but completion latency corresponds to the time it takes to serve an atom given a particular bandwidth allocation. The only way to reduce this number is hence to increase the allocated bandwidth.

7.6 Requirement Verification

The requirement verification step asserts that the use-case is valid according to Definition 7.2, meaning that all bandwidth and latency requirements are satisfied without allocating more bandwidth than provided by the memory controller. If the use-case is valid, the computed configuration is stored as a candidate, along with its associated slack bandwidth, determined according to Definition 7.3. It is possible that there exists a configuration with a larger burst count that provides more slack bandwidth. This is investigated by increasing the burst count to the next power of two and iterate in the flow, as shown in Fig. 7.1. For each iteration, the configuration

Table 7.7 Allocated bandwidths and service latencies together with their corresponding bounds

Requestor	b_r (MB/s)	b'_r (MB/s)	Θ_r^{cc} (cc)	$\hat{\Theta}_r^{cc}$ (cc)
r_0	210.0	210.5	258	300
r_1	210.0	210.5	106	110
r_2	210.0	210.5	88	90
r_3	20.0	20.6	182	200

with the most slack bandwidth is stored. The loop terminates in either of two cases: (1) The latency requirements of a requestor could not be satisfied. Increasing the burst count results in larger access granularity and thus longer latencies. The configuration flow will hence not be able to satisfy the failing latency constraint for any larger burst count. (2) The amount of slack bandwidth is less than or equal to the slack bandwidth of the previous iteration. If the amount of slack bandwidth does not increase with the burst count, the access granularity of the memory is already too large considering the request sizes of the requestors. Any gains in bank efficiency or read/write switching efficiency are hence cancelled out by losses in data efficiency. The iteration is guaranteed to terminate with these conditions, since both latency requirements and request sizes are finite. For all current practical applications, burst size is unlikely to go beyond four, since this already implies a large access granularity and potentially long latencies.

The inputs for the requirement verification of our use-case are shown in Table 7.7. The configuration with $BC = 1$ is valid, since all bandwidth and latency requirements are satisfied and the total allocated bandwidth is approximately 652 MB/s, resulting in a slack bandwidth of 8 MB/s. This configuration is stored as a potential candidate, while we iterate in the flow to generate a configuration with $BC = 2$. Only the most interesting parameters and configuration settings are shown during the iteration to keep the discussion focused.

Increasing the burst count to two results in the pattern set generated by the bank scheduling algorithm that was previously shown in Table 4.3. This pattern set offers a gross bandwidth of 716 MB/s, which is an increase of 8.5% over the pattern with $BC = 1$. Normalizing the requirements with respect to the new pattern set results in the output in Table 7.8. Observe that the service latency requirements, expressed in service cycles, are reduced compared to the previous iteration, due to the longer service cycles resulting from the new patterns. The latency requirement of r_2 is negative, meaning that it cannot be satisfied for any priority assignment. We also note that the required service rate of r_2 is doubled compared to before. This is because its request size is 64 B and the access granularity of the pattern set increased from 64 B to 128 B, resulting in a data efficiency of 50%. The configuration flow bravely continues, although the arbiter configuration fails to assign priorities. The requirement verification steps notes that the configuration is not valid, since all latency requirements could not be satisfied and approximately 20% more gross bandwidth than available has been allocated. Neither of these problems can be resolved by increasing the burst count. Further iteration is hence not required and the configuration with $BC = 1$ is chosen for the use-case, concluding our running example.

Table 7.8 Output from normalization stage with $BC = 2$

Requestor	$\hat{\Theta}_r$ (sc)	ρ_r (su/sc)
r_0	5	0.294
r_1	0	0.294
r_2	-1	0.587
r_3	2	0.0279

7.7 Experimental Results

The running example in this chapter demonstrated how the configuration flow works with a single use-case. Now, we experimentally show how the flow performs with a large number of use-cases with requestors accessing a 16-bit DDR2-400 memory, connected to the SDRAM back-end proposed in Chap. 4. The timings of this memory device were previously listed in Table 3.1. The memory is shared using a CCSP arbiter with six bits of precision in the service allocation mechanism.

The experiment in this section evaluates the configuration flow and show the benefit of iterating over different burst counts. For this purpose, we generate 5000 synthetic use-cases, each with six requestors. The requestors issue requests with sizes $64 \cdot i$ bytes, where i is a uniformly varying integer in the range 1–8. Together, the requestors require 660 MB/s in all use-cases. This corresponds to 82.6% of the peak bandwidth offered by the memory, which is very close to 100% of the gross bandwidth provided by the memory with $BC = 1$. The generated service latency requirements are randomized according to $27 \cdot j$ clock cycles at 200 MHz, where j is a uniformly varying integer in the range 1–100. Some latency requirements may hence be quite tight, while others may be quite relaxed and up to 50% longer than the refresh interval of the memory.

To illustrate the benefits of iterating over burst counts, we let the flow configure the use-cases in four different ways. First, using only memory patterns with $BC = 1$, then using only patterns with $BC = 2$, followed by only using $BC = 4$. Lastly, we use the iterating scheme presented in this chapter that tries all of these and chooses the best result. All generated patterns use a burst length (BL) of eight words. Just like in Sect. 5.7, we look at the three metrics: (1) the percentage of use-cases where bandwidth requirements are satisfied for all requestors, (2) the percentage where latency requirements are satisfied for all requestors, and (3) the percentage where both bandwidth and latency requirements are satisfied for all requestors. The results of this experiment are shown in Fig. 7.4.

Bandwidth requirements are only satisfied in 21% of the use-cases with $BC = 1$, due to the high load required by the requestors. The success rate increases to 54% with $BC = 2$, because of the extra 55 MB/s provided by the longer patterns. At this point, some requests may be larger than access granularity of the memory, being 128 B, although the reducing data efficiency does not eliminate the benefits of the increased gross bandwidth. However, further increasing the burst count to $BC = 4$

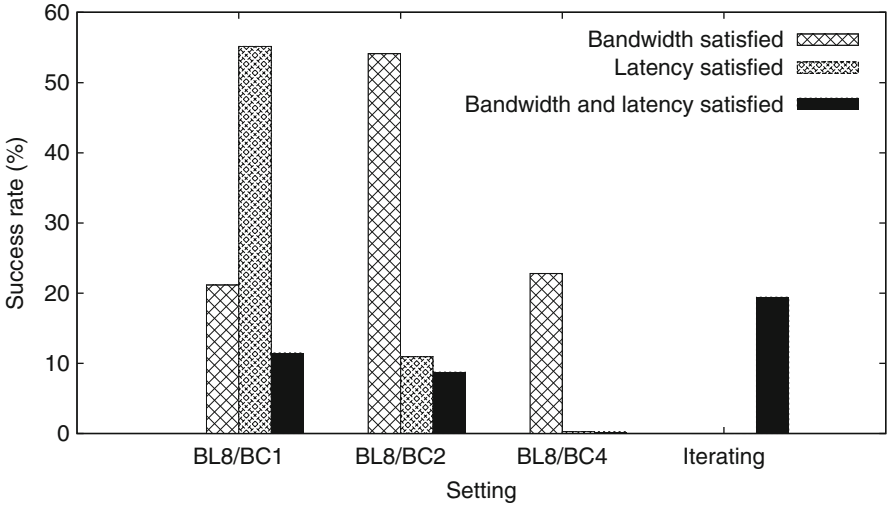

Fig. 7.4 The percentage of use-cases with bandwidth and latency requirements satisfied using pattern generators with fixed and iterating burst counts

reduces the percentage of satisfied bandwidth requirements to 23%, since the access granularity of 256 B is now too large compared to the sizes of the requests. This results in that more bandwidth is wasted than is added by the longer access patterns. While the percentage of use-cases with satisfied bandwidth requirements initially increases with burst count, the percentage of latency requirements monotonically decrease, starting at 55% with $BC = 1$, 11% for $BC = 2$, and ending at 0% with $BC = 4$. Looking at the percentage of use-cases with both bandwidth and latency requirements satisfied, we conclude that it is kept at approximately 10% for both $BC = 1$ and $BC = 2$, in the first case because of unsatisfied bandwidth requirements, and in the second case because of failing latency requirements. The total success rate with $BC = 4$ is 0%, as it is totally killed by the latency requirements.

We conclude the experiment by looking at the results with the iterative approach that is normally used in the flow. We ignore the separate results for bandwidth and latency requirements, since these depend on which pattern set is chosen for a use-case if either set of requirements fail. Instead, we focus on the percentage of use-cases where all requirements are satisfied. The iterative approach satisfies the requirements of almost twice as many use-cases as any of the fixed burst counts. This result is not surprising, since a larger solution space is considered. The only drawback of this approach is increased run time of the configuration flow. However, this is negligible, since the time to configure a use-case is in the order of a few seconds.

7.8 Summary

This chapter presented a configuration flow that automatically computes instantiation parameters and configuration settings for the proposed resource front-end and SDRAM back-end with the goal of satisfying *net bandwidth and service latency requirements* of the requestors. If there are multiple configurations that satisfy the requirements, the one with the *most slack bandwidth* is preferred to improve the performance of requestors that do not require composable service.

The configuration flow consists of five main steps: (1) *memory pattern generation*, (2) *normalization of requirements*, (3) *arbiter configuration*, (4) *denormalization of allocation*, and (5) *requirement verification*. The pattern generation step first generates a pattern with burst count one for the specified memory device. Other burst counts are considered later by iteration in the flow. No memory patterns are needed if the memory is an SRAM. Instead, a pattern specification is generated that describes the characteristics of accesses to the memory. This enables the same configuration flow to be used both for SRAM and SDRAM. The specification of the generated memory pattern determines the instantiation parameters for the Atomizer and Data Bus in the resource front-end. The requirements of the requestors are then normalized to make them independent of the target memory. The normalization is done by converting the requirements into the abstract domain of service units and service cycles. *This use of abstraction enables the same arbiter configuration to be used for all supported memories at the expense of making tight latency requirements somewhat more difficult to satisfy due to pessimism in the conversion.* The size of a service unit equals the access granularity of the generated pattern set. A requestor may have a data efficiency of less than 100%, making it unable to use all data in a service unit. This is addressed by dividing the net bandwidth requirement with the data efficiency, turning it into a gross bandwidth requirement before normalization. *This technique enables net bandwidth requirements to be satisfied for requestors with arbitrary data efficiency.* The arbiter configuration tries to find settings that satisfy the normalized requirements. For the Frame-Based Static-Priority (FBSP) and Credit-Controlled Static-Priority (CCSP) arbiters, this is done by first allocating bandwidth, and then assigning priorities according to a waterfall approach. The service allocations of the requestors are then denormalized back into bandwidths in MB/s and service latencies in clock cycles. The service latencies and completion latencies required to configure the Delay Blocks are derived in this step. The final step verifies if all requirements are satisfied and computes the slack bandwidth. Patterns with higher burst count are evaluated by iteration in the flow if it can result in a valid configuration with more slack bandwidth.

Chapter 8
Related Work

This book proposes a predictable and composable memory controller design and a supporting configuration flow to satisfy real-time requirements of applications in embedded systems. In this chapter, we position the proposed solution with respect to the related work. This is done in three parts. First in Sect. 8.1, we relate the presented Time-Division Multiplexing (TDM), Frame-Based Static-Priority (FBSP), and Credit-Controlled Static-Priority (CCSP) arbiters to the existing body of resource arbiters. We then compare our predictable SDRAM controller to the state of the art in memory controller design in Sect. 8.2. Lastly, we position our way of achieving composable service to earlier approaches in Sect. 8.3.

8.1 Resource Arbitration

Resource arbitration has been extensively researched in different contexts during the past half century. This discussion focuses on arbiters suitable for scheduling of transaction-based System-on-Chip (SoC) resources, such as memories, peripherals, and interconnect. Several such arbiters are based on the Round-Robin algorithm [89], because it is simple and starvation free. Weighted Round-Robin [66] and Deficit Round-Robin [111] are extensions of this algorithm that guarantee each requestor a minimum service, proportional to an allocated rate (bandwidth), in a common periodically repeating frame of fixed size. This type of *frame-based arbitration* is easy to implement, but suffers from an *inherent coupling between allocation granularity and latency*, where allocation granularity is inversely proportional to the frame size [140]. Increasing the frame size results in finer allocation granularity, reducing over-allocation. However, this comes at the cost of increased latencies for all requestors, as demonstrated in Sect. 5.7. TDM belongs to this class of frame-based arbiters, although it suffers from the additional disadvantage that it requires a schedule to be stored for each configuration, which may be very costly if the frame size or the number of use-cases are large.

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_8, © Springer Science+Business Media, LLC 2011

The coupling between allocation granularity and latency is addressed in [64, 65, 107] with hierarchical framing strategies that accomplish exact allocation over multiple frames. However, these algorithms, just as the family of Fair Queuing algorithms [140], are unable to distinguish different latency requirements, as the rate is the only parameter affecting scheduling. This results in an *unwanted coupling between latency and rate*, where latency is inversely proportional to the allocated rate. Requestors with low rate requirements hence suffer from long latencies unless their rates are increased, resulting in over-allocation. Our requirement that we must be able to distinguish latency-sensitive and latency-tolerant requestors implies that latency and rate must be decoupled, speaking in favor of priority-based solutions, such as FBSP and CCSP.

Four approaches using static-priority scheduling are presented in [20, 50, 52, 57]. Static-priority schedulers have the benefit of decoupling latency and rate and being cheap to implement in hardware. However, the arbiters in [50, 52, 57], as well as the FBSP arbiter, are frame based and couple allocation granularity and latency. In [20], service is allocated in discrete chunks, the size of which depends on the priority of the requestor and the total number of requestors sharing the resource. This couples allocation granularity and latency. Moreover, at most 84% of the resource capacity can be allocated to the requestors as guaranteed service. A priority-based arbiter is presented in [98] for resource scheduling in SoCs. The rate regulator uses an accounting mechanism based on integers that is easily implemented in hardware, and inspired the implementation of the CCSP rate regulator. However, it is not clear if the proposed arbiter meet our requirements, as no results are presented on provided bandwidth, latency, over-allocation, or area and speed of the implementation.

The CCSP arbiter resembles an arbiter with a rate regulator that enforces a (σ, ρ) constraint [28] on requested service together with a static-priority scheduler, a combination we will refer to as Sigma-Rho Static-Priority (SRSP) arbitration in this work. Similarly to SRSP, the CCSP rate regulator replenishes the service available to a requestor continuously, instead of basing it on frames, decoupling allocation granularity and latency. This allows over-allocation to become negligible, which is essential for scarce SoC resources with very high loads, such as memories. Both arbiters furthermore use priorities to decouple latency and rate. However, instead of enforcing a burstiness constraint on *requested service*, as done by SRSP, both CCSP and FBSP enforce it on *provided service*. We proceed by discussing this difference in more detail. Figure 8.1a shows an arbiter that enforces an upper bound on requested service, such as [28, 102, 141]. The rate regulator is positioned before the Request Buffers, allowing it to regulate the arriving requests by holding them until a particular burstiness constraint, such as a minimum inter-arrival time, is satisfied. Note that there is no communication between the scheduler and the rate regulator. A rate regulator that enforces an upper bound on provided service, such as those in [26, 34, 57, 66, 111] and the CCSP and FBSP rate regulators, is shown in Fig. 8.1b. As seen in the figure, the rate regulator is positioned after the Request Buffers. It is hence only aware of requests at the heads of the buffers, and cannot constrain arrivals of requests in any way. The scheduler communicates the identifier

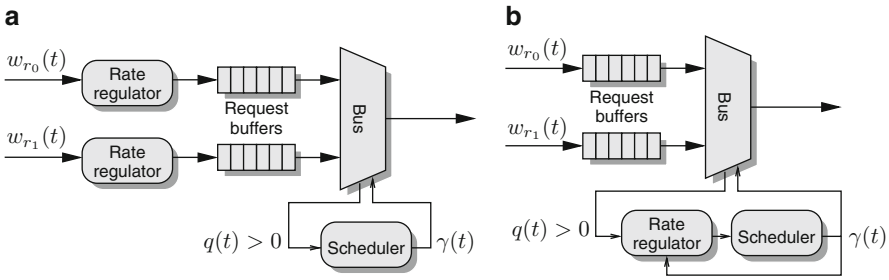

Fig. 8.1 Two arbiters regulating requested service and provided service, respectively. (**a**) Requested service regulation. (**b**) Provided service regulation

of the scheduled requestor, $\gamma(t)$, back to the rate regulator every cycle to update the accounting mechanism. Enforcing an upper bound on provided service has the benefit that the amount of service required by a particular request does not have to be known in advance. For example, it is typically not possible for a processor scheduler to know the number of cycles required to decode a video frame, since this is highly data dependent. A similar problem occurs in SDRAM controllers if the arbiter is scheduling memory cycles, as opposed to a fixed amount of transferred data, since the time to serve a request is not known in advance for most SDRAM controllers. These situations cannot be efficiently handled if requested service is regulated, since the rate regulator determines if the request is eligible based on the effort involved in serving it. It is possible to use worst-case assumptions to estimate the amount of required service, although this is very inefficient if the variance is large. This is efficiently handled when combining provided service regulation with preemptive arbitration, since the accounting is updated for every service unit, causing a requestor to be preempted when it runs out of budget. Unlike SRSP, CCSP enjoys this benefit without any performance penalty. In fact, we conjectured in [12], based on experimental results, that CCSP and SRSP provide identical latencies for all requests.

8.2 SDRAM Controllers

Existing SDRAM controller designs are either statically or dynamically scheduled, depending on which kind of systems they target. Statically scheduled memory controllers [16, 105, 115] combine static front-end arbitration with static scheduling of SDRAM commands in the back-end. The command generator executes a static schedule of SDRAM commands that has been computed at design time. The read and write bursts in the schedule are statically mapped to the requestors according to their requirements. These controllers are predictable, since the latency of a request and the offered net bandwidth can be bounded at design time by analyzing the schedule and the mapping of bursts to requestors. It is hence possible to formally verify that all requestor requirements are satisfied at design time. For this reason,

statically scheduled memory controllers are most frequently used in embedded systems with firm or hard real-time requirements, such as TV picture improvement ICs [116]. The predictability of statically scheduled memory controllers comes at the expense of flexibility. The precomputed schedule in the back-end makes these designs unable to adapt to changes in the behaviors of the requestors. This limits their applicability to applications whose requestors have regular access patterns and where the request sizes and read/write ratio do not change during a use-case. Static front-end arbitration furthermore couples latency and allocated bandwidth, as previously discussed in Sect. 8.1. This makes statically scheduled memory controllers unable to satisfy the requirements of latency-sensitive requestors with low bandwidth requirements without wasting bandwidth. Finally, many schedules are required, as the number of use-cases grows exponentially with the number of applications. These schedules may take a long time to compute and require significant storage space, since each schedule may contain thousands of commands for newer memories. *These properties prevent statically scheduled controllers from scaling to larger systems with more use-cases and more dynamic applications.*

Dynamically scheduled memory controllers, on the other hand, combine dynamic front-end arbitration with dynamic back-end scheduling. These controllers target high efficiency and flexibility to fit in high-performance systems with dynamic applications whose behaviors may not be known up front. Priorities are used in the front-end arbitration in several dynamic memory controllers [52, 73, 87, 130, 132] to cater to the needs of latency-sensitive requestors, often corresponding to processors that stall while waiting for cache lines. Some controllers provide additional mechanisms to further reduce latency. The design in [73] lets high priority requestors preempt lower priority requestors that are receiving service in the back-end, which reduces latency at the expense of memory efficiency. Another technique is to prefer reads over writes [110], which is beneficial if reads are blocking while writes are posted.

A number of dynamic memory controllers use information about memory state when scheduling to improve memory efficiency. This consideration is typically done in the back-end, but some designs communicate memory state to the front-end arbiter, blurring the distinction between the two. Typically, requests are preferred if they target an open row in a bank [87, 104, 110], if they fit with the current direction of the data bus [23, 52, 132], or a combination of the two [73, 75, 130]. The idea behind the scheduler in [87] is to exploit thread-level parallelism by scheduling bursts belonging to the same requestor simultaneously in all banks. It is shown that this approach reduces the average latency of the requestors, although it probably reduces memory efficiency. A disadvantage of all the mentioned scheduling algorithms is that they are not capable of long-term planning. Instead, they make short-term scheduling decisions to transfer data on the bus as fast as possible, such as preferring read or write commands over activates and precharges. These decisions are clever on short term, but may result in sub-optimal performance in the long run. This issue is addressed by a self-optimizing memory controller in [58]. The proposed memory scheduler uses theory from reinforcement learning to recognize which scheduling decisions that result in high long-term memory efficiency.

Many dynamic designs [23,52,73,75,87,130,132] use rate regulators in the front-end to protect requestors from each other. This is especially important in controllers with priority-based arbiters, since these are often prone to starvation. The designs in [52, 75, 132] regulate the amount of requested service, while [23, 73, 87, 130] regulate provided service. The rate regulators in [23, 52, 73, 87, 130, 132] are all frame-based and hence couple allocation granularity, latency, and rate. An interesting difference between these controllers is that [52, 132] only consider rate regulation of high-priority requestors, corresponding to processors, while low-priority hardware accelerators are assumed to be well-behaved.

The problem with dynamically scheduled memory controllers is that the interaction between the front-end and back-end scheduler is complex, especially in the presence of reordering mechanisms. For this reason, neither of the mentioned memory controllers provides bounds on either latency or provided gross/net bandwidth. *Dynamically scheduled memory controllers are hence typically unsuitable for applications with firm or hard requirements on worst-case latency and bandwidth.*

A related problem with dynamically scheduled controllers is that there is often not a clear relation between configuration parameters and the provided bandwidth and latency. This prevents automatic generation of configuration parameters that satisfy requestor requirements. Instead, successful deployment of these controllers has to rely on extensive simulation to measure the provided bandwidth and latency with different configuration parameters. This results in a significant verification effort as it has to be done for all use-cases and must be repeated every time a requestor is added, removed, or changes behavior.

Despite the mentioned predictability challenges, a dynamically scheduled controller providing bounded latency is presented in [100]. However, this controller is implemented in the DRAMSim [129] memory simulator and does not have a real hardware implementation. The memory controller supports any JEDEC compliant SDRAM device, but is limited to a single front-end arbiter, namely Round-Robin. This arbiter fails to satisfy our requirements as it cannot deliver on diverse bandwidth requirements or distinguish latency-sensitive and latency-tolerant requestors.

Our memory controller combines elements of statically and dynamically scheduled memory controllers. The front-end uses predictable dynamic arbiters in the class of Latency-Rate ($\mathcal{LR}$) servers, which enables us to satisfy diverse latency requirements. The command generator uses a hybrid approach based on memory patterns that is a mix between static and dynamic command scheduling. Memory patterns are precomputed sub-schedules that are dynamically combined at run time, enabling the controller to accommodate traffic that is not fully known at design time in a predictable fashion. Our memory controller offers bounds on both net bandwidth and the latency of requestors at design time, which enables configuration settings to be automatically synthesized for a given set of requirements. *The proposed memory controller significantly increases flexibility over existing predictable memory controllers and is suitable for systems with firm and hard real-time requirements on worst-case bandwidth and latency.*

8.3 Composable Service

A number of works in the field of high-performance computing discuss performance isolation of applications in predictable systems by providing lower bounds on performance. Fair Queuing Memory Systems [93] and Virtual Private Caches [92] are both part of the Virtual Private Machine framework [94] for multi-core resource management. The authors show that the service provided by a Virtual Private Machine running at an allocated fraction of the original capacity is at least as good as a real machine with the same resources. This allows real-time requirements to be verified by simulation in isolation, assuming that the applications executing on the system are *performance monotonic* [72], which means that having additional resources cannot result in worse performance.

Two simulation-based approaches to verification of real-time requirements in predictable systems are presented in [72, 99]. The idea in these papers is to simulate the execution of an application and verify that real-time requirements are satisfied when emulating maximum interference from other applications by delaying responses until their worst-case latency. This is similar to our approach to composability, although with some important differences. In contrast to our work, the authors propose to disable emulation of worst-case interference for all requestors when deploying the system to benefit from slack and increase performance. This breaks the isolation between applications, limiting the approach to applications and systems that either have performance monotonic execution, or can be captured in a performance monotonic model, such as deterministic data-flow graphs [18]. Furthermore, no hardware architecture is presented for the approach in [72], although our proposed resource front-end can be used to implement the methodology.

The drawback of relying on performance monotonicity is that it severely restricts both the supported platform and application software. The platform has to be free from timing anomalies, which can appear in shared caches or in dynamically scheduled processors, such as PowerPCs [74]. Another example is that increasing the memory bandwidth allocated to an application may lead to a net performance loss due to cache pollution, caused by an increased number of prefetches [92]. Timing anomalies also appear in some multi-processor systems [40], making verification results of distributed applications unreliable. Applications can furthermore not have timing dependent behavior, such as adapting the quality level of a video decoder based on the decoding time of previous frames.

Verification of composable systems, on the other hand, does not rely on performance monotonicity, since applications are completely independent of each other in both the value and time domains. There are currently three approaches to composable system design. The first involves not sharing any resources, which is used by federated architectures in the automotive and aerospace industries [70]. This method is trivially composable, but prohibitively expensive for systems that do not have safety-critical applications. The second option is the time-triggered approach [69] that uses component interfaces where the time instances for communication are

specified at design time. This approach requires a global notion of time and is limited to applications that can be statically scheduled at design time. The third approach is to dynamically schedule resource access at run time using TDM, as proposed in [17,48]. Using run-time scheduling has the benefit of supporting event-triggered systems, although a limitation of TDM is that it couples the worst-case latency and the allocated bandwidth of an application. Another drawback of this approach is that it only applies to inherently composable resources, such as Zero-bus-turnaround (ZBT) SRAM. An SDRAM memory is an example of a resource that is not inherently composable, since requestors can affect each other's temporal behavior by changing the memory state, such as switching direction from read to write. We addressed this in [120], where a composable SDRAM controller based on memory patterns is proposed. The idea is to enforce a read/write switch between all requests, thus removing the possibility for requestors to interfere with each other by changing the state of the resource. A disadvantage of this approach is that no slack is created by the resource itself. Only slack generated by the arbiter can hence be used to improve performance of requestors that do not require composable service.

This work adds a fourth approach to composability, based on delaying signals sent from predictable shared resources to the requestors to emulate worst-case interference. *The approach applies to any combination of predictable resource and arbiter in the class of $\mathcal{LR}$ servers, thereby widely extending the types of platforms that can provide composable service.* Generalizing composability beyond arbiters and resources that are inherently composable affects the conditions for which the resource can continue to provide composable service as more applications are integrated. This property is known as stability of prior services [70]. When resources are shared using TDM [17,48], stability of prior services is guaranteed as long as the slots reserved by an application remains unchanged. Our approach has a more general requirement to address the diversity of the supported arbiters. Stability of prior services is guaranteed as long as the starting times and finishing times of a requestor are unchanged as more applications are integrated. If we use TDM, which is a predictable arbiter in the class of $\mathcal{LR}$ servers, to share a predictable resource, the requirement is satisfied by not changing the slots reserved by an application. On the other hand, if we use FBSP or CCSP arbitration, stability is only guaranteed if lower priority requestors are added, since additional higher priority requestors increases the worst-case interference that must be emulated.

Our approach to composable resource sharing makes the resource composable at the level of requestors, which is a sufficient requirement to be composable at the level of applications. However, this is also a stricter requirement, since requestors belonging to the same application are allowed to interfere with each other. The CoMPSoC platform capitalizes on this by having two levels of arbitration in its predictable and composable processor tiles [9]. The first level is a composable application scheduler, and the second a predictable task scheduler that does not have to be composable. This type of arbitration enables tasks from the same application to use slack created in the task scheduler to boost performance without violating composability at the application level. A novel aspect of our approach is that composable service can be enabled or disabled per requestor at run time by turning

the emulation of worst-case interference on or off. This introduces the notion of *partially composable systems*, where some applications are free from interference and others are not. The benefit of this distinction is that it *allows requestors that do not have real-time requirements to use slack to improve performance.*

Chapter 9
Conclusions and Future Work

There is a growing mapping and verification problem in System-on-Chips (SoCs), as an increasing number of applications with real-time requirements are mapped on heterogeneous multi-processor platforms with distributed memory hierarchies. To reduce cost, resources in the platform, such as SRAM and SDRAM memories, are shared between applications using a variety of arbiters. The mapping process is challenging as it involves both binding application tasks and data structures to processing elements and memories in the platform, and determining configuration settings such that all real-time requirements are satisfied. Once a candidate mapping has been determined, system-level simulation is often used to verify the real-time requirements. However, resource sharing introduces interference between applications, causing their temporal behaviors to become inter-dependent. As a result, all combinations of concurrently executing applications have to be verified together, resulting in a verification complexity that grows exponentially with the number of applications. To manage this increasing complexity, industry often restricts verification to a subset of use-cases with the most sensitive requirements, resulting in poor coverage. Formal verification offers significantly better coverage, but is typically not an alternative, since many resources, such as memory controllers, are not designed with formal analysis in mind. This problem is addressed in this book by designing a memory controller with requirements on *predictability*, *abstraction*, *composability*, and *automation*. We conclude this work by explaining how the proposed solution delivers on these requirements in Sect. 9.1, followed by a discussion on future work in Sect. 9.2.

9.1 Conclusions

Each of the four requirements predictability, abstraction, composability, and automation are discussed in turn, as we highlight the strengths and limitations of the proposed memory controller and its configuration flow.

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0_9, © Springer Science+Business Media, LLC 2011

9.1.1 Predictability

This work presents a predictable memory controller, consisting of a *back-end* and a *front-end*. The back-end makes a DDR2/DDR3 SDRAM behave in a predictable manner, enabling us to derive a tight bound on the provided gross bandwidth. The provided net bandwidth depends on the relation between the sizes and alignments of the requests from the requestors and the access granularity of the memory. The back-end accesses the memory by interleaving SDRAM bursts over all banks, which provides a high bound on gross bandwidth at the expense of a large access granularity. To be efficient, requests should have sizes in words that are integer multiples of the product between the burst length and the number of banks. Other requests may significantly reduce the provided bandwidth. The front-end contains a predictable arbiter that allows the SDRAM back-end to be shared among multiple requestors. Three predictable arbiters are presented, Time-Division Multiplexing (TDM), Frame-Based Static-Priority (FBSP), and Credit-Controlled Static-Priority (CCSP), and their respective bounds on service latency and over-allocation are derived and evaluated. Based on this evaluation, we conclude that CCSP enables fine-grained resource allocation that reduces over-allocation without negatively impacting service latency. This makes CCSP a suitable arbiter for highly loaded resources with diverse bandwidth and latency requirements, such as SDRAM controllers. We show the merit of our predictable memory controller by experimentally demonstrating that the combination of back-end and front-end provides conservative guarantees on bandwidth and service latency, even in the presence of misbehaving requestors.

The key benefit of the proposed predictable SDRAM controller is that it provides *increased flexibility* compared to current predictable controllers. These controllers are either completely statically scheduled, or dynamically scheduled but limited to a single front-end arbiter. In contrast, our controller combines dynamic front-end scheduling using any predictable arbiter with a combination of static and dynamic scheduling in the back-end.

9.1.2 Abstraction

The combination of front-end and predictable back-end behaves like a Latency-Rate ($\mathcal{LR}$) server, which means that a minimum bandwidth and a maximum latency are guaranteed to a requestor. The memory controller and the associated analysis methods and tooling are designed to use the $\mathcal{LR}$ server model as a *shared resource abstraction*, which makes our solution very general. It is possible to use *any arbiter in the class of $\mathcal{LR}$ servers*, which enables the controller to cater to diverse sets of requirements. There are many well-known arbiters belonging to the class, such as Weighted Round-Robin, Deficit Round-Robin, several varieties of Fair Queuing, as well as the presented TDM, FBSP, and CCSP arbiters. It is also possible to remove the SDRAM back-end and use the front-end with *any other predictable memory*, such as an SRAM.

The $\mathcal{LR}$ server model enables verification with several commonly used formal analysis frameworks, such as network calculus and data-flow analysis. Our memory controller hence allows *any combination* of predictable memory and $\mathcal{LR}$ arbiter to be used transparently for formal verification of applications with any of these frameworks. However, we show that the benefits of abstraction come at the cost of increased latency. The $\mathcal{LR}$ server model assumes that a request is served in a continuous manner according to the allocated rated rate of its requestor, while it is actually either served at the full capacity of the memory, or not served at all. This causes the $\mathcal{LR}$ server model to over-estimate the time when a request is served by an amount that is inversely proportional to the allocated bandwidth. The model is furthermore unable to capture the bursty service behavior of many priority-based arbiters, such as FBSP and CCSP, which may further reduce accuracy.

9.1.3 Composability

The proposed front-end is made composable by adding a Delay Block that delays all signals sent from a resource to a requestor to emulate worst-case interference from other requestors. Achieving composability in this way *removes restrictions* imposed by earlier approaches that are limited to applications and resources that can be statically scheduled, or sharing inherently composable resources at run time using TDM. In contrast, our approach applies to *any combination* of predictable resource and arbiter in the class of $\mathcal{LR}$ servers *without any assumptions* on the application.

Delaying signals to emulate worst-case interference makes the average latency equal to the computed worst case, which may significantly increase latency if the two are far apart. This furthermore increases the required buffering to sustain the allocated bandwidth. Currently, our approach uses the $\mathcal{LR}$ server model to compute the worst-case release time of delayed signals, which introduces some pessimism adding to this cost. However, a strength of our approach is that composable service can be dynamically activated and deactivated, and hence limited to requestors with real-time requirements. This removes the added cost for requestors that do not require composable service, and furthermore allows them to benefit from slack bandwidth to improve performance.

9.1.4 Automation

The proposed memory controller is supported by a configuration flow that automatically computes appropriate configuration settings for the front-end and back-end, given bandwidth and latency requirements of the requestors. The flow uses abstraction to make the memory and arbiter configuration independent of each other. This allows all supported arbiters to be configured for all supported

memories in a streamlined fashion without a special case for every combination. The configuration tool explores different configuration options for the back-end, but uses a simple bandwidth allocation algorithm when configuring the arbiter. The flow may hence be unable to find a configuration that satisfies a given set of requirements even if one exists. This is left as an open issue.

9.2 Future Work

For every door your close in research, two new doors are opened. This section discusses interesting future work and open issues in the context of this work.

9.2.1 Reducing Power Consumption

The proposed memory controller accesses the memory in an interleaving manner. This enables us to guarantee high gross bandwidth, but the frequent activates and precharges consume a lot of power, as explained in Sect. 3.4.3.2. We believe it is important to address this issue, since power consumption is of utmost importance for many embedded systems. A simple technique that fits within the current architecture is to let the back-end benefit from locality to reduce power. The idea is to gate out any activate or (auto)-precharge commands if the right row is already open. This is a low-complexity extension of the back-end that reduces power by adding a degree of dynamism to the execution of a memory pattern. Another direction is to exploit low-power features of the memories themselves and incorporate the predictable use of power-down modes [61, 62]. This option is primarily interesting in systems where the memory is not constantly utilized. Power is saved by letting the SDRAM enter a low-power state when idle. However, powering up the memory incurs a latency penalty on the requestors, thus making it more difficult to satisfy latency requirements. This presents an interesting trade-off between power and latency that deserves further exploration.

9.2.2 Opportunities with 3D Integration

3D integration enables stacking SDRAM on top of one or more logic layers and connecting them with vertical wires called through-silicon-vias (TSVs) [36], thus removing the need to go off-chip to access the memory. Since TSVs require less area and consume less power than off-chip pins, the number of connections to the SDRAM can significantly increase [131]. Removing the pin constraint has many benefits for memory efficiency, since sharing wires between memory banks can be reduced or removed. Three possible scenarios are: (1) Every bank gets

its own command bus, removing losses due to command conflicts. (2) The data path is split into a separate read and write channel, removing lost cycles due to read/write switches. (3) Each bank gets its own data path, removing all conflicts on the data path. These changes incrementally bring each bank closer to being separate memories. To what extent the sharing between banks is reduced depends on the cost and availability of TSVs, which is not yet fully known. Interesting future work, while this is being determined, involves investigating the benefits of the three scenarios in proportion to the increase of signals on the memory interface.

The impact of 3D integration may go well beyond the memory devices themselves and change the architecture of contemporary systems. Increasing the number of connections to memory enables wider memory interfaces and higher peak bandwidths. However, wider interfaces increase the access granularity of the memory, reducing data efficiency and net bandwidth [24]. An alternative to wider interfaces is to use multiple memory channels, each with their own memory controller. Although recent publications [4, 24, 96] propose using multiple memory channels, no one has considered how to do this in a predictable or composable way. Multi-channel solutions enable more net bandwidth and a reduction of memory contention. We hence believe that extending our approach to cover multiple channels is important future work to meet future real-time requirements.

9.2.3 Improved Arbiter Configuration

The arbiter configuration attempts to automatically derive arbiter settings, such that all bandwidth and service latency requirements are satisfied. The current configuration approach for the FBSP and CCSP arbiters is rather limited. First, we assign $\rho' = \rho$, for all requestors, even though a higher allocated rate could help satisfying potential throughput requirements. The reason is that allocating a higher rate impacts the latency of lower priority requestors, and we currently cannot oversee how much extra bandwidth that can be allocated before the requirements of all requestors cannot be satisfied anymore.

A benefit of the CCSP arbiter is that it can allocate a particular burstiness allowance, σ', per requestor, independently from the allocated rate. However, we always assign $\sigma' = 1$ for all requestors, effectively throwing away the flexibility provided by a second allocation parameter. The reason for this assignment is that the $\mathcal{LR}$ server model does not capture the benefits of a higher allocated burstiness, as previously discussed in Sect. 7.4.1. We believe it is important to extend the $\mathcal{LR}$ server model to cover the effects of multiple service rates. Initial steps have already been taken for Priority Budget Scheduling [114] and the CCSP arbiter [112], but these models are specific to the particular arbiters and do not cover the general case. We believe that a more general model can be derived that captures the behavior of a whole class of schedulers with bursty behaviors.

Using the improved service model in [112] involves deviating from our abstraction requirement, although it enables us to set the two allocation parameters (σ', ρ')

freely. Exploiting this requires a more refined allocation scheme that considers the allocation parameters and requirements of all requestors simultaneously, since the service allocation of one requestor impacts the service provided to another.

9.2.4 Reconfiguration

The proposed predictable and composable memory controller currently only offers limited support for multiple use-cases. Only a subset of the hardware blocks, namely the Delay Block and CCSP arbiter, have programmable configurations, while other important parameters are fixed at design time. The most prominent example of this is the SDRAM back-end, which generates the memory patterns with a hard-coded finite-state machine in the Command Generator. Making this block programmable has two important advantages: (1) It increases the re-usability of the component, since it can be used with different memory devices without modification. (2) It allows the memory patterns to be changed between use-cases, increasing the diversity of the use-case requirements that can be accommodated by the platform. A consequence of changing the memory patterns between use-cases is that the service unit size used by the Atomizer and the best-case service cycle length used by the Data Bus also changes. These blocks hence have to be connected to the configuration infrastructure and the parameters made run-time programmable.

The current implementation of the configuration flow only supports a single use-case. Generalizing it to support multiple use-cases should not pose any conceptual difficulties as it involves iterating through the flow for every use-case. If the back-end is made programmable, different memory patterns can be used for every use-case. Otherwise, a single pattern has to be chosen for all use-cases. We also need to change verification step to consider multiple use-cases and modify the quality metric to consider the total slack in all use-cases or something more refined.

9.2.5 Data-Flow Model of Memory Controller

The memory controller acts like a $\mathcal{LR}$ server, which enables formal verification using well-known performance analysis frameworks, such as network calculus and data-flow analysis. Data-flow analysis is suitable for the SoC context, since it supports cyclic dependencies between nodes in the graph. This is an essential feature that allows communication between tasks using finite buffers to be included in the model, which is necessary to capture the behavior of partitioned applications. It furthermore enables modeling flow-control mechanisms that are common in communication protocols used in contemporary SoCs. A data-flow model of the proposed memory controller would bridge the gap between the application and the memory controller by enabling them to be represented in the same framework. This would allow throughput requirements of applications accessing the memory

controller to be verified using traditional data-flow techniques. There are also benefits related to buffer sizing. Currently, buffers are sized by trial-and-error to be large enough to prevent overflow. A data-flow model of the architecture enables us to extend the configuration flow with a buffer sizing step that uses an existing tool [133] to find sufficient sizes for all buffers in the controller, given throughput requirements of the applications. This would further automate our approach and reduce area by removing unnecessary buffer space.

Appendix A
System XML Specification

This chapter shows the XML specifications that are used as input to the configuration flow, presented in Chap. 7. First, we look at the architecture specification in Appendix A.1, followed by the use-case specification in Appendix A.2.

A.1 Architecture Specification

The architecture specification lists a number of Intellectual Property (IP) components, each with a number of ports. For each port, type, protocol and other relevant architecture parameters are specified. The architecture does not specify IP components that are automatically synthesized, such as the Network-on-Chip (NoC), and the resource front-end. However, some parameters are listed as directions for this synthesis. A number of such parameters required to synthesize the NoC [47] have been removed for clarity. The most interesting IP component in the context of this book is the memory controller, which is our proposed SDRAM back-end. The memory controller has a single port to which a synthesized resource front-end is connected. Port parameters determine the arbiter that is used in the front-end, as well as if a work conserving instance is desired. They also specify the timings of the SDRAM, used to generate the memory patterns.

```
<architecture  id="book">

  <!-- Default clock used by IPs and instantiated modules. -->
  <clk id="global_500MHz" period="2.0" />

  <ip id="ip_0" type="IP">
     <port id="p1" type="Initiator" protocol="MMIO_DTL">
        <parameter id="width" type="int" value="32" />
        <parameter id="blocksize" type="int" value="32" />
        <parameter id="speed_var" type="double" value="10.2" />
     </port>
  </ip>
```

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0, © Springer Science+Business Media, LLC 2011

```
<ip id="ip_1" type="IP">
   <port id="p1" type="Initiator" protocol="MMIO_DTL">
      <parameter id="width" type="int" value="32" />
      <parameter id="blocksize" type="int" value="32" />
      <parameter id="speed_var" type="double" value="10.2" />
   </port>
   <port id="p2" type="Initiator" protocol="MMIO_DTL">
      <parameter id="width" type="int" value="32" />
      <parameter id="blocksize" type="int" value="32" />
      <parameter id="speed_var" type="double" value="10.2" />
   </port>
</ip>
<ip id="ip_2" type="IP">
   <port id="p1" type="Initiator" protocol="MMIO_DTL">
      <parameter id="width" type="int" value="32" />
      <parameter id="blocksize" type="int" value="32" />
      <parameter id="speed_var" type="double" value="10.2" />
   </port>
</ip>

<!--  Memory clock -->
<clk id="memory_200MHz" period="5.0" />

<ip id="sdram_backend" type="MemoryController">
   <port id="p1" type="Target" protocol="MMIO_DTL">

    <parameter id="delay" type="bool" value="1"/>

     <!-- Arbiter Specification -->
    <parameter id="arbiter" type="string" value="CCSP"/>
    <parameter id="workConserving" type="bool" value="0"/>

    <!-- Memory Specification -->
    <parameter id="memoryId" type="string" value="DDR2-400"/>
    <parameter id="capacity" type="uint" value="65536" />
    <parameter id="nbrOfBanks" type="uint" value="4" />
    <parameter id="clk" type="string" value="memory_200MHz"/>
    <parameter id="dataRate" type="uint" value="2" />
    <parameter id="tREFI" type="double" value="7800" />
    <parameter id="burstSize" type="uint" value="8" />
    <parameter id="width" type="int" value="16" />
    <parameter id="RC" type="uint" value="11" />
    <parameter id="RCD" type="uint" value="3" />
    <parameter id="CL" type="uint" value="3" />
    <parameter id="WL" type="uint" value="2" />
    <parameter id="AL" type="uint" value="0" />
    <parameter id="RP" type="uint" value="3" />
    <parameter id="RFC" type="uint" value="21" />
    <parameter id="RAS" type="uint" value="8" />
    <parameter id="RTP" type="uint" value="2" />
    <parameter id="WR" type="uint" value="3" />
    <parameter id="FAW" type="uint" value="10" />
    <parameter id="RRD" type="uint" value="2" />
```

```
      <parameter id="CCD" type="uint" value="2" />
      <parameter id="WTR" type="uint" value="2" />
    </port>
  </ip>
</architecture>
```

A.2 Use-Case Specification

The use-case specification specifies the applications and their connections. Each connection corresponds to a requestor. For each requestor, the type (read, write, or both) is specified, along with burst sizes (B), required bandwidth (MB/s), and latency requirements (ns). Each requestor furthermore has a parameter that determines if responses and flow-control signals should be delayed to emulate maximum interference from other requestors. This parameter hence determines if the resource front-end should be programmed to provide composable service to the requestor. The use-case below corresponds to the running example in Chap. 7. This simple use-case only has one requestor per application and all applications execute concurrently.

```
<communication>
  <application id="Application_0">
    <connection qos="GT" id="0">
      <initiator ip="ip_0" port="p1"/>
      <target ip="sdram_backend" port="p1"/>
      <read latency="0" bw="210" burstsize="512"/>
      <parameter id="maxLatency" type="double" value="1500"/>
      <parameter id="delay" type="bool" value="1"/>
    </connection>
  </application>
  <application id="Application_1">
    <connection qos="GT" id="1">
      <initiator ip="ip_1" port="p1"/>
      <target ip="sdram_backend" port="p1"/>
      <write latency="0" bw="210" burstsize="128"/>
      <parameter id="maxLatency" type="double" value="550"/>
      <parameter id="delay" type="bool" value="1"/>
     </connection>
  </application>
  <application id="Application_2">
    <connection qos="GT" id="2">
      <initiator ip="ip_1" port="p2"/>
      <target ip="sdram_backend" port="p1"/>
      <read latency="0" bw="210" burstsize="64"/>
      <parameter id="maxLatency" type="double" value="450"/>
      <parameter id="delay" type="bool" value="1"/>
    </connection>
  </application>
  <application id="Application_3">
    <connection qos="GT" id="3">
```

```
        <initiator ip="ip_2" port="p1"/>
        <target ip="sdram_backend" port="p1"/>
        <write latency="0" bw="20" burstsize="256"/>
        <parameter id="maxLatency" type="double" value="1000"/>
        <parameter id="delay" type="bool" value="1"/>
      </connection>
    </application>
  </communication>
```

Appendix B
Glossary

This chapter provides a guide to the language used in this book. Appendix B.1 contains a list of abbreviations and Appendix B.2 a list of symbols.

B.1 List of Abbreviations

This list of abbreviations explains the most commonly used abbreviations in this book.

AXI	Advanced eXtensible Interface
CCSP	Credit-Controlled Static-Priority
CC	Clock Cycle
DDR	Double-Data-Rate
DRAM	Dynamic RAM
DSP	Digital Signal Processor
DTL	Device Transaction Level
FAW	Four-activate window
FBSP	Frame-Based Static-Priority
IP	Intellectual Property
$\mathcal{LR}$	Latency-Rate
NoC	Network-on-Chip
PE	Processing Element
RAM	Random Access Memory
RR	Round-Robin
SC	Service Cycle
SDRAM	Synchronous Dynamic RAM
SoC	System-on-Chip
SRAM	Static RAM

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0, © Springer Science+Business Media, LLC 2011

SRSP Sigma-Rho Static-Priority
SU Service Unit
TDM Time-Division Multiplexing
ZBT Zero-bus-turnaround

B.2 List of Symbols

The list of symbols explains most of the symbols that constitute the formal framework in this book. The symbols are sorted in alphabetical order with the Greek alphabet preceding the Latin.

Table B.1 List of symbols

Symbol	Description	Page
$\alpha(\omega_r^k)$	Address of request ω_r^k (bytes)	50
β	Precision used by CCSP to approximate the allocated rates and burstinesses (bits)	124
$\gamma(t)$	Scheduled requestor at a time t	110
Δ	Latency offset (clock cycles)	148
δ_{read}	Minimum clock cycles between a read and a write command	89
δ_{write}	Minimum clock cycles between a write and a read command	89
η	Fraction of service cycles with rounded-down completion latency	152
Θ_r	Service latency of requestor r (service cycles)	113
$\hat{\Theta}_r$	Service latency requirement of requestor r (service cycles)	175
Θ_r^{cc}	Service latency of requestor r (clock cycles)	148
$\hat{\Theta}_r^{cc}$	Service latency requirement of requestor r (clock cycles)	172
$\lambda(\omega_r^k, t)$	Service cycle length when serving request ω_r^k starting at time t (clock cycles)	107
$\check{\lambda}$	Minimum service cycle length (clock cycles)	155
$\bar{\lambda}$	Average service cycle length during worst-case conditions (clock cycles)	148
$\pi_r(t)$	Potential of requestor r at time t (service units)	124
ρ_r	Requested service rate of requestor r (service units/service cycle)	108
ρ_r'	Allocated rate of requestor r (service units/service cycle)	109
ρ_r''	Discrete allocated rate of requestor r (service units/service cycle)	109
σ_r'	Allocated burstiness of requestor r (service units)	122
σ_r''	Discrete allocated burstiness of requestor r (service units)	125
ϕ_r	Allocated slots of requestor r in a frame-based rate regulator	115
ω_r^k	K^{th} request from requestor r	50
Ω_r	Set of requests from requestor r	50
$a(\omega_r^k)$	Alignment of request ω_r^k (bytes)	50
BC	Burst count. Number of read/write commands per bank per access pattern	67
BL	Burst length (words)	50

(continued)

Table B.1 (continued)

Symbol	Description	Page
b_r	Requested bandwidth of requestor r (MB/s)	54
b'_r	Allocated bandwidth of requestor r (MB/s)	54
b^{gross}	Gross memory bandwidth (MB/s)	53
b^{net}	Net memory bandwidth (MB/s)	53
b^{peak}	Peak memory bandwidth (MB/s)	51
b^{slack}	Slack memory bandwidth (MB/s)	173
$c_r(t)$	Credits of requestor r in a CCSP arbiter at time t	125
c^*	Credits used to approximate completion latency	152
d	Denominator used to approximate the discrete allocated rate in a CCSP arbiter	124
d^*	Denominator used to approximate completion latency	152
dr	Data rate (words/clock cycle)	50
e^{bank}	Bank efficiency	52
e^{cmd}	Command efficiency	52
e^{data}	Data efficiency	52
e^{gross}	Gross memory efficiency	53
e^{net}	Net memory efficiency	53
e^{ref}	Refresh efficiency	51
e^{rw}	Read/write efficiency	51
f	The frame size of a frame-based rate regulator	115
f_{mem}	Clock frequency (MHz)	50
g	Access granularity of a memory pattern (bytes)	67
$l(\omega_r^k)$	Completion latency of request ω_r^k (service cycles)	113
l_r^{cc}	Completion latency of requestor r (clock cycles)	149
n	Numerator used to approximate the discrete allocated rate in a CCSP arbiter	124
n^*	Numerator used to approximate completion latency	152
n_{acc}	Remaining number of read or write commands to schedule	83
n_{act}	Remaining number of activate commands to schedule	83
n_{banks}	Number of banks in the SDRAM	50
n_{pipe}	Number of pipeline stages between the Request Buffer and Response Buffer	148
$o_\rho(\rho''_r, \rho'_r)$	The over-allocated rate of requestor r (service units/service cycle)	109
$o_\sigma(\sigma''_r, \sigma'_r)$	Over-allocated burstiness of requestor r (service units)	125
p_r	Priority level of requestor r	128
$q_r(t)$	Backlog of requestor r at time t	111
R	Set of requestors sharing the memory	50
R_r^+	Set of requestors with higher priority than r	119
R_t^a	Set of active requestors at time t	122
$s(\omega_r^k)$	Size of request ω_r^k (service units)	107
$s^{bytes}(\omega_r^k)$	Size of request ω_r^k (bytes)	50
$t_a(\omega_r^k)$	Arrival time of request ω_r^k	108
$t_{aux}(x)$	Maximum time to serve x service units, excluding refreshes (clock cycles)	76
t_{block}	Maximum blocking time (clock cycles)	71

(continued)

Table B.1 (continued)

Symbol	Description	Page
$tCCD$	Minimum time between two read commands or two write commands (clock cycles)	50
tCL	Time after read command until first data is available on the bus (clock cycles)	50
$t_f(\omega_r^k)$	Finishing time of request ω_r^k	111
$tFAW$	Window in which maximally four banks may be activated (clock cycles)	50
t_{read}^{first}	Cycle with first read command in a read pattern	89
t_{write}^{first}	Cycle with first write command in a write pattern	89
t_{read}^{last}	Cycle with last read command in a read pattern	82
t_{write}^{last}	Cycle with last write command in a write pattern	82
t_{read}^{pre}	Cycle when last bank is precharged after a read pattern	82
t_{write}^{pre}	Cycle when last bank is precharged after a write pattern	82
tRC	Minimum time between successive activate commands to the same bank (clock cycles)	50
$tRCD$	Minimum time between activate and read/write commands to the same bank (clock cycles)	50
$tRFC$	Minimum time between a refresh command and a successive refresh or activate command (clock cycles)	50
$tRAS$	Minimum time after an activate command to a bank until that bank is allowed to be precharged (clock cycles)	50
t_{read}	Length of a read pattern (clock cycles)	68
t_{ref}	Length of a refresh pattern (clock cycles)	68
$tREFI$	Average refresh interval (clock cycles)	50
tRP	Minimum time between a precharge command on a bank and a successive activate command (clock cycles)	50
$tRRD$	Minimum time between activates to different banks (clock cycles)	50
$tRTP$	Minimum time between a read and precharge command (clock cycles)	50
t_{rtw}	Length of a read/write switching pattern (clock cycles)	68
$t_s(\omega_r^k)$	Starting time of request ω_r^k	111
$t_{shortest}$	Length of the shortest access pattern found so far (clock cycles)	83
$t_{tot}(x)$	Total time to serve x service units (clock cycles)	77
$t_{transfer}$	Clock cycles with data transfer in an access pattern	73
tWL	Time after write command until first data is available on the bus (clock cycles)	50
tWR	Minimum time after the last data has been written to a bank until a precharge may be issued (clock cycles)	50
t_{write}	Length of a write pattern (clock cycles)	68
t_{wtr}	Length of a write/read switching pattern (clock cycles)	68
$tWTR$	Internal write to read command delay (clock cycles)	50
$w_r(t)$	Requested service curve of requestor r at time t	108
$w'_r(t)$	Provided service curve of requestor r at time t	111
$\check{w}'_r(t)$	Lower bound on provided service bound for requestor r at time t (service units)	113
$\hat{w}'_r(t)$	Upper bound on provided service bound for requestor r at time t (service units)	123
w_{mem}	Width of the data bus (bytes)	50

References

1. L. Abeni and G. Buttazzo. Resource Reservation in Dynamic Real-Time Systems. *Real-Time Systems*, 27(2):123–167, 2004.
2. S. Adee. 37 years of Moore's law. *IEEE Spectrum*, 45(5):56, 2008.
3. S. Adee. Thanks for the memories. *IEEE Spectrum*, 46(5), 2009.
4. E. Aho, J. Nikara, P. A. Tuominen, and K. Kuusilinna. A case for multi-channel memories in video recording. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 934–939, 2009.
5. B. Akesson. An analytical model for a memory controller offering hard-real-time guarantees. Master's thesis, Lund's Institute of Technology, May 2005.
6. B. Akesson. *Predictable and Composable System-on-Chip Memory Controllers*. PhD thesis, Eindhoven University of Technology, Feb. 2010. ISBN: 978-90-386-2169-2.
7. B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, 2007.
8. B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *Proc. Euromicro Conference on Digital System Design (DSD)*, pages 547 –555, Aug. 2009.
9. B. Akesson, A. Molnos, A. Hansson, J. Ambrose Angelo, and K. Goossens. Composability and predictability for independent application development, verification, and execution. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, chapter 2. Springer, Dec. 2010.
10. B. Akesson, L. Steffens, and K. Goossens. Real-Time Scheduling of Hybrid Systems using Credit-Controlled Static-Priority Arbitration. Technical report, NXP Semiconductors, 2007. http://www.es.ele.tue.nl/~kakesson/publications/pdf/NXP-TN-2007-00119.pdf.
11. B. Akesson, L. Steffens, and K. Goossens. Efficient Service Allocation in Hardware Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 59–68, 2009.
12. B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 3–14, 2008.
13. ARM Limited. http://www.arm.com, 2011.
14. ARM Limited. *AMBA AXI Protocol Specification*, 2003.
15. N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *Real-Time Systems*, 1991.

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0, © Springer Science+Business Media, LLC 2011

16. S. Bayliss and G. Constantinides. Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search. In *Field-ProgrammableTechnology, 2009. International Conference on*, pages 304–307, Dec. 2009.
17. M. Bekooij, A. Moonen, and J. van Meerbergen. Predictable and Composable Multiprocessor System Design: A Constructive Approach. In *Bits&Chips Symposium on Embedded Systems and Software*, 2007.
18. M. Bekooij, S. Parnar, and J. van Meerbergen. Performance guarantees by simulation of process networks. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 10–19, 2005.
19. A. Benveniste. Loosely time-triggered architectures for cyber-physical systems. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 3–8, Mar. 2010.
20. T. Bjerregaard and J. Sparsø. A scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip. In *Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 34–43, 2005.
21. S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual conference on Design automation*, pages 746–749, 2007.
22. J.-Y. L. Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the Internet*. Springer-Verlag New York, Inc., 2001.
23. A. Burchard, E. Hekstra-Nowacka, and A. Chauhan. A real-time streaming memory controller. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 20–25, 2005.
24. P. Casini. SoC Architecture to Multichannel Memory Management Using Sonics IMT. White paper, 2008. Sonics, inc.
25. K. Chandrasekar, B. Akesson, and K. Goossens. Improved Power Modeling of DDR SDRAMs. In *Proc. Euromicro Conference on Digital System Design (DSD)*, 2011.
26. H. Chao and J. Hong. Design of an ATM shaping multiplexer with guaranteed output burstiness. *Computer Systems Science and Engineering*, 12(2):131–141, 1997.
27. T. Claasen. The logarithmic law of usefulness. *Semiconductor international*, 21(8):175–184, 1998.
28. R. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.
29. W. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 684–689, 2001.
30. M. Di Natale and A. Sangiovanni-Vincentelli. Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools. *Proceedings of the IEEE*, 98(4):603 –620, Apr. 2010.
31. D. Dunning, R. Mooney, P. Stolt, and B. Casper. Tera-Scale Memory Challenges and Solutions. *Intel Technology Journal*, 13(4):80–101, Dec. 2009.
32. S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SOC for advanced set-top box and digital TV systems. *IEEE Design and Test of Computers*, pages 21–31, 2001.
33. S. Edwards and E. Lee. The Case for the Precision Timed (PRET) Machine. In *Design Automation Conference. DAC '07. 44th ACM/IEEE*, pages 264–265, June 2007.
34. A. Francini and F. Chiussi. Minimum-latency dual-leaky-bucket shapers for packet multiplexers: theory and implementation. *Quality of Service, 2000. IWQOS. 2000 Eighth International Workshop on*, pages 19–28, 2000.
35. S. Fuller and L. Millett. Computing performance: Game over or next level? *IEEE Transactions on Computers*, 44(1):31–38, Jan. 2011.
36. P. Garrou. *Handbook of 3D Integration: Technology and Applications of 3D Integrated Circuits*, volume 1. Wiley-VCH, 2008.
37. K. Goossens, O. P. Gangwal, J. Röver, and A. P. Niranjan. Interconnect and memory organization in SOCs for advanced set-top boxes and TV — Evolution, analysis, and trends. In *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 15, pages 399–423. Kluwer, 2004.

38. K. Goossens and A. Hansson. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 306–311, 2010.
39. K. Goossens, D. She, A. Milutinovic, and A. Molnos. Composable dynamic voltage and frequency scaling and power management for dataflow applications. In *Proc. Euromicro Conference on Digital System Design (DSD)*, pages 107–114, Sept. 2010.
40. R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, pages 416–429, 1969.
41. D. Grund, J. Reineke, and R. Wilhelm. A Template for Predictability Definitions with Supporting Evidence. *Predictability and Performance in Embedded Systems*, pages 22–31, 2011.
42. P. Gumming. The TI OMAP Platform Approach to SoC. *Winning the SoC revolution: experiences in real design*, page 97, 2003.
43. A. Hansson, B. Akesson, and J. van Meerbergen. Multi-processor programming in the embedded system curriculum. *SIGBED Rev.*, 6(1):1–9, 2009.
44. A. Hansson, M. Coenen, and K. Goossens. Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 954–959, 2007.
45. A. Hansson, M. Ekerhult, A. Molnos, A. Milutinovic, A. Nelsson, J. Ambrose, and K. Goossens. Design and Implementation of an Operating System for Composable Processor Sharing. *Microprocessors and Microsystems (MICPRO)*, 35(2):246–260, 2011.
46. A. Hansson and K. Goossens. Trade-offs in the configuration of a network on chip for multiple use-cases. In *The 1st ACM/IEEE International Symposium on Networks-on-Chip*, pages 233–242, 2007.
47. A. Hansson and K. Goossens. *On-Chip Interconnect with aelite: Composable and Predictable Systems*. Embedded Systems Series. Springer, Nov. 2010.
48. A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–24, 2009.
49. A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis. *IET Computers & Digital Techniques*, pages 398–412, 2009.
50. F. Harmsze, A. Timmer, and J. van Meerbergen. Memory arbitration and cache management in stream-based systems. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 257–262, 2000.
51. W. S. Hayes jr. Memory pattern generation based on specification and environment. Master's thesis, Eindhoven University of Technology, 2009.
52. S. Heithecker and R. Ernst. Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 575–578, 2005.
53. J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. *Research at Intel white paper*, 2006.
54. J. Henkel. Closing the SoC design gap. *IEEE Transactions on Computers*, pages 119–121, 2003.
55. T. Henriksson, P. Van Der Wolf, A. Jantsch, and A. Bruce. Network calculus applied to verification of memory access performance in SoCs. In *Embedded Systems for Real-Time Multimedia. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on*, pages 21–26, 2007.
56. M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Transactions on Computers*, 41(7), 2008.
57. S. Hosseini-Khayat and A. Bovopoulos. A simple and efficient bus management scheme that supports continuous streams. *ACM Transactions on Computer Systems (TOCS)*, 13(2):122–140, 1995.
58. E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture. ISCA '08. 35th International Symposium on*, pages 39–50, 2008.

59. International Technology Roadmap for Semiconductors (ITRS), 2009.
60. B. Jacob, S. Ng, and D. Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007.
61. JEDEC Solid State Technology Association. *DDR2 SDRAM Specification*, JESD79-2E edition, Apr. 2008.
62. JEDEC Solid State Technology Association. *DDR3 SDRAM Specification*, JESD79-3D edition, Sept. 2009.
63. J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589, 2005.
64. C. Kalmanek, H. Kanakia, and S. Keshav. Rate controlled servers for very high-speed networks. *Proceedings of GLOBECOM*, pages 12–20, 1990.
65. S. S. Kanhere and H. Sethu. Fair, efficient and low-latency packet scheduling using nested deficit round robin. *High Performance Switching and Routing, 2001 IEEE Workshop on*, pages 6–10, 2001.
66. M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8):1265–1279, Oct. 1991.
67. K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.
68. P. Kollig, C. Osborne, and T. Henriksson. Heterogeneous Multi-Core Platform for Consumer Multimedia Applications. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1254–1259, 2009.
69. H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
70. H. Kopetz, C. El Salloum, B. Huber, R. Obermaisser, and C. Paukovits. Composability in the time-triggered system-on-chip architecture. In *SOC Conference, IEEE International*, pages 87–90, 2008.
71. E. A. Lee. Absolutely positively on time: what would it take? *IEEE Transactions on Computers*, 38(7):85–87, 2005.
72. J. Lee and K. Asanovic. METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symp*, pages 135–147, 2006.
73. K. Lee, T. Lin, and C. Jen. An efficient quality-aware memory controller for multimedia platform SoC. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(5):620–633, 2005.
74. T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium*, pages 12–21, 1999.
75. C. Macian, S. Dharmapurikar, and J. Lockwood. Beyond performance: Secure and fair memory management for multiple systems on a chip. In *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 348–351, 2003.
76. S. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, pages 162–167, 2004.
77. T. F. Melham. *Formalising Abstraction Mechanisms for Hardware Verification in Higher Order Logic*. PhD thesis, University of Cambridge, 1990. Also available as Technical Report UCAM-CL-TR-201.
78. Calculating Memory System Power for DDR2. Technical report, Micron Technology Inc., 2005. TN-47-04.
79. Calculating Memory System Power for DDR3. Technical report, Micron Technology Inc., 2007. TN-41-01.
80. Micron Technology Inc. http://www.micron.com, 2011.
81. MIPS Technologies. http://www.mips.com, 2011.

82. A. Molnos, J. A. Ambrose, A. Nelson, R. Stefan, S. Cotofana, and K. Goossens. A Composable, Energy-Managed, Real-Time MPSOC Platform. In *Proc. Int'l Conference on Optimization of Electrical and Electronic Equipment (OPTIM)*, pages 870–876, May 2010.
83. A. Molnos and K. Goossens. Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors. In *Proc. Euromicro Conference on Digital System Design (DSD)*, pages 409–418, 2009.
84. G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38:114–117, 1965.
85. G. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, 1975.
86. O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 57–66, 2007.
87. O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers. *IEEE Micro*, 29(1):22–32, 2009.
88. J. Muttersbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, 2000.
89. J. B. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, COM-35(4):435–438, 1987.
90. A. Nelson. Conservative Application-Level Performance Analysis through Simulation of a Multiprocessor System on Chip. Master's thesis, Eindhoven University of Technology, 2009.
91. A. Nelson, A. Molnos, and K. Goossens. Composable power management with energy and power budgets per application. In *Proc. Int'l Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*, 2011.
92. K. Nesbit, J. Laudon, and J. Smith. Virtual private caches. In *Proceedings of the 34th annual international conference on Computer architecture*, pages 57–68, 2007.
93. K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, 2006.
94. K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, 2008.
95. A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.
96. J. Nikara, E. Aho, P. Tuominen, and K. Kuusilinna. Performance analysis of multi-channel memories in mobile devices. In *International Symposium on System-on-Chip 2009*, pages 128–131, Oct. 2009.
97. R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):956–965, July 2009.
98. C. Otero Pérez, M. Rutten, J. van Eijndhoven, L. Steffens, and P. Stravers. Resource reservations in shared-memory multiprocessor SOCs. In *Dynamic and Robust Streaming In And Between Connected Consumer-Electronics Devices*, chapter 5, pages 109–137. Springer, 2005.
99. M. Paolieri, E. Quiñones, F. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 57–68. ACM New York, NY, USA, 2009.
100. M. Paolieri, E. Quinones, F. Cazorla, and M. Valero. An Analyzable Memory Controller for Hard Real-Time CMPs. *Embedded Systems Letters, IEEE*, 1(4):86–90, Dec. 2009.
101. Philips Semiconductors. *Device Transaction Level (DTL) Protocol Specification. Version 2.2*, 2002.

102. J. Rexford, F. Bonomi, A. Greenberg, and A. Wong. Scalable architecture for fair leaky-bucket shaping. *Proc. IEEE INFOCOM*, 3:1054–1062, 1997.
103. M. Ringhofer. Design and implementation of a memory controller for real-time applications. Master's thesis, Graz University of Technology, 2006.
104. S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 128–138, 2000.
105. J. Roest. Spider project: Detailed design description of the DDR SDRAM controller. Technical Report 1.3, Philips Consumer Electronics, 2004. Philips confidential.
106. B. Rumpler. Complexity Management for Composable Real-Time Systems. In *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 365–373, 2006.
107. D. Saha, S. Mukherjee, and S. K. Tripathi. Carry-over round robin: a simple cell scheduling mechanism for ATM networks. *IEEE/ACM Transactions on Networking*, 6(6):779–796, 1998.
108. R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-chip: Reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, 2006.
109. R. Selvaggi and L. Pearlstein. Broadcom mediadsp: A platform for building programmable multicore video processors. *Micro, IEEE*, 29(2):30–45, 2009.
110. J. Shao and B. Davis. A burst scheduling access reordering mechanism. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 285–294, 2007.
111. M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. *ACM SIGCOMM Computer Communication Review*, 25(4):231–242, 1995.
112. F. Siyoum, B. Akesson, S. Stuijk, K. Goossens, and H. Corporaal. Resource-Efficient Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration. In *Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.
113. S. Sriram and S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC, 2000.
114. J. Staschulat and M. Bekooij. Dataflow models for shared memory access latency analysis. In *Proceedings of the seventh ACM international conference on Embedded software (EMSOFT)*, pages 275–284, 2009.
115. F. Steenhof. Columbus SDRAM interface. Technical Report 0.8, Philips Consumer Electronics, 2002. Philips confidential.
116. F. Steenhof, H. Duque, B. Nilsson, K. Goossens, and R. Peset Llopis. Networks on chips for high-end consumer-electronics TV system architectures. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 148–153, 2006.
117. L. Steffens, M. Agarwal, and P. van der Wolf. Real-Time Analysis for Memory Access in Media Processing SoCs: A Practical Approach. *ECRTS '08: Proceedings of the Euromicro Conference on Real-Time Systems*, pages 255–265, 2008.
118. D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, 1998.
119. STMicroelectronics and CEA. Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology, Nov. 2010. White paper.
120. E. Strooisma. A predictable and composable front-end for system on chip memory controllers. Master's thesis, Delft University of Technology, 2008.
121. S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Design Automation Conference. DAC '07. 44th ACM/IEEE*, pages 777–782, June 2007.
122. G. Teshome Woldegebreal. Front-end for composable resource sharing using latency-rate servers. Master's thesis, Delft University of Technology, 2009.
123. K. Tindell, A. Burns, and A. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

124. T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability. *IEEE Micro*, 30(5):66–75, Sept. 2010.
125. C. van Berkel. Multi-core for Mobile Phones. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1260–1265, 2009.
126. P. van der Wolf and J. Geuzebroek. SoC Infrastructures for Predictable System Integration. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, Mar. 2011.
127. B. Vermeulen and K. Goossens. *Multi-Core Embedded Systems*, chapter 5. CRC Press/Taylor & Francis Group, 2010.
128. J. Vink, K. van Berkel, and P. van der Wolf. Performance analysis of SoC architectures based on latency-rate servers. *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 200–205, 2008.
129. D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: a memory system simulator. *SIGARCH Comput. Archit. News*, 33:100–107, Nov. 2005.
130. W.-D. Weber. *Efficient Shared DRAM Subsystems for SOCs*. Sonics, Inc, 2001. White paper.
131. C. Weis, N. Wehn, L. Igor, and L. Benini. Design Space Exploration for 3D-stacked DRAMs. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, Mar. 2011.
132. S. Whitty and R. Ernst. A bandwidth optimized SDRAM controller for the MORPHEUS reconfigurable architecture. In *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS)*, 2008.
133. M. Wiggers, M. Bekooij, and G. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pages 658–663, 2007.
134. M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *SCOPES '07: Proceedings of the 10th international workshop on Software & compilers for embedded systems*, pages 11–22, 2007.
135. N. Wingen. What if you could design tomorrow's system today? In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 835–840, 2007.
136. L. Woltjer. Optimal DDR controller. Master's thesis, University of Twente, Jan. 2005.
137. D. Woo and H. Lee. Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era. *IEEE Transactions on Computers*, 41(12):24–31, 2008.
138. W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.
139. Xilinx. http://www.xilinx.com/, 2011.
140. H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–96, Oct. 1995.
141. H. Zhang and D. Ferrari. Rate-controlled service disciplines. *Journal of High-Speed Networks*, 3(4):389–412, 1994.

Index

B. Akesson and K. Goossens, *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*, Embedded Systems 2,
DOI 10.1007/978-1-4419-8207-0, © Springer Science+Business Media, LLC 2011