

## Chapter 2

# Proposed Solution

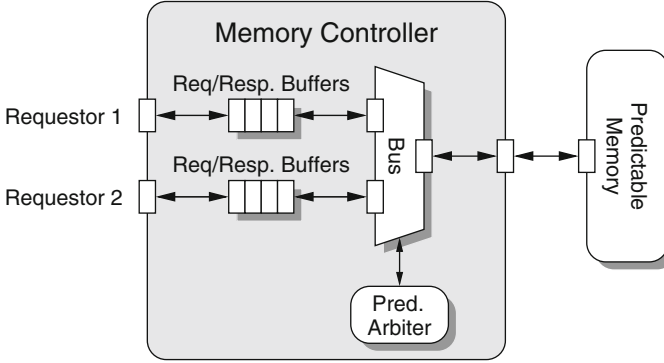
The previous chapter identified problems related to mapping applications with real-time requirements to a heterogeneous multi-processor platform with SDRAM memory and verifying that all requirements are satisfied. We then committed to designing a memory controller with requirements on predictability, 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



**Fig. 2.1** Overview of predictable memory controller

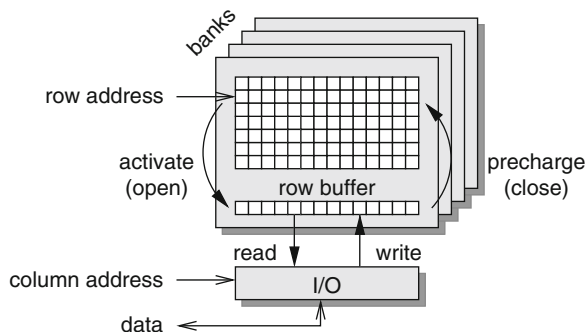
to a memory satisfying this requirement as a *predictable memory*. We also require a *predictable arbiter*, where the number of interfering requests that can be scheduled before a particular request is bounded. Combining a predictable memory and a predictable arbiter allows the maximum time to schedule 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.

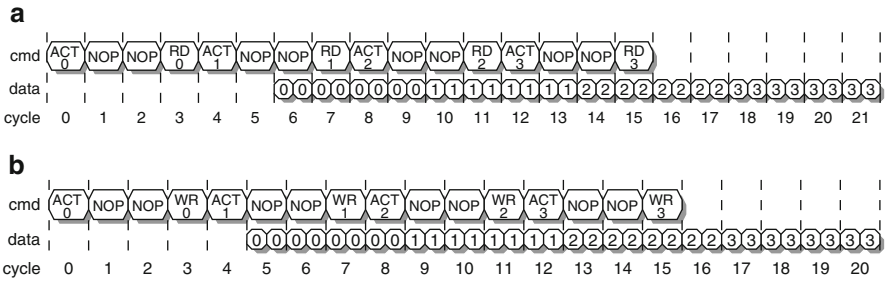
We propose a hybrid approach to SDRAM command scheduling that combines elements of statically and dynamically scheduled SDRAM controllers in an attempt to get the best of both worlds. Our approach is based on *predictable memory patterns*, which are precomputed sequences (sub-schedules) of SDRAM commands 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.

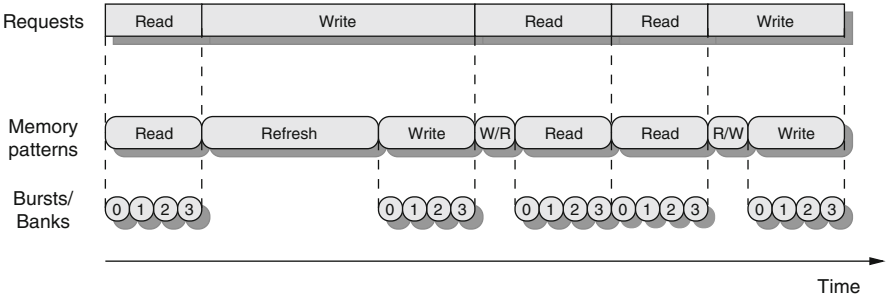
The read and write patterns consist of a fixed number of SDRAM bursts, all targeting the same row in a bank. The bursts are issued to the different banks in sequence, since the data bus is shared between all banks to reduce the number of pins on the SDRAM interface. The fixed number of bursts is hence 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

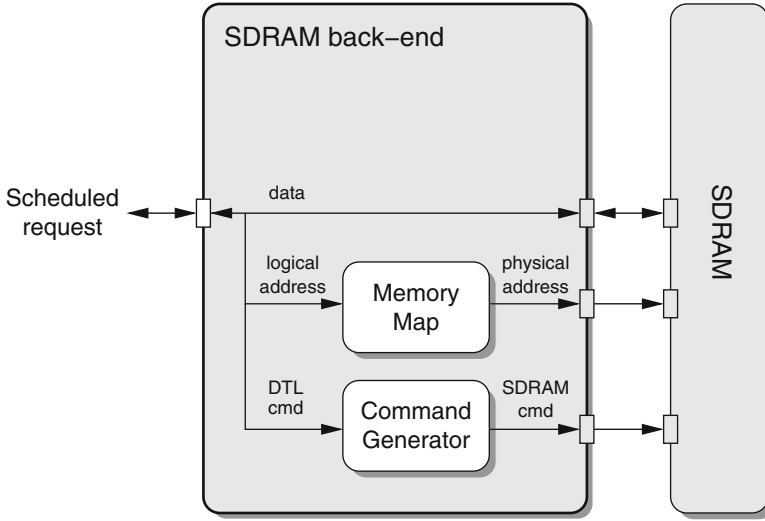


**Fig. 2.4** Mapping from requests to patterns to SDRAM bursts

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.

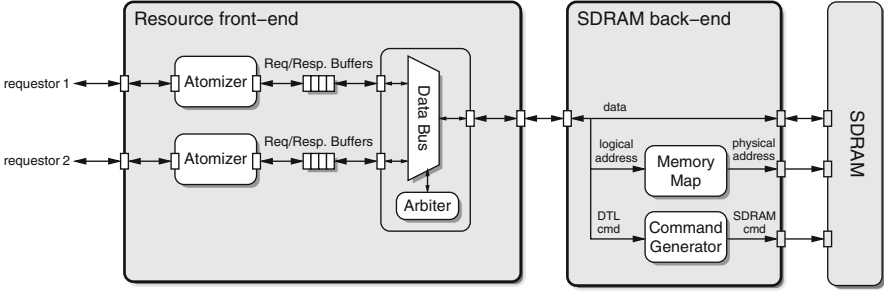
### 2.1.3 Predictable Arbitration

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 a 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.

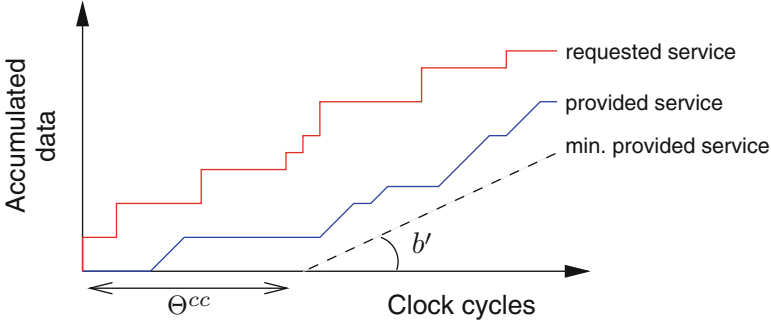
The benefit of using a predictable arbiter that combines rate regulation and preemption is that it makes it possible to bound the latency of a requestor without relying on the cooperation of others. Instead, bounds are based either on the allocated budgets (explicit rate regulation) or on analysis results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

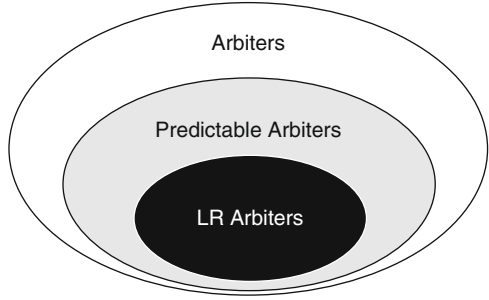
The memory controller is required to use a common abstraction that captures the temporal behavior of many different memory and arbiter types to mitigate the increasing system complexity. We have chosen Latency-Rate ( $\mathcal{LR}$ ) servers [118] as the shared resource abstraction in this work. In essence,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



$\Theta^{cc}$ , 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

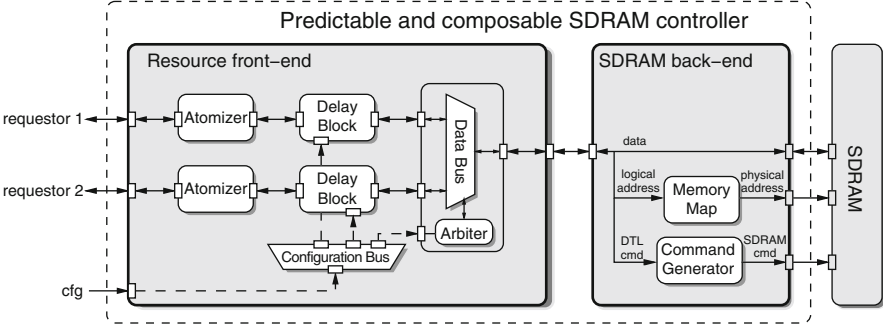
The memory controller is required to provide composable service to applications to enable them to be developed and verified independently, as explained in Sect. 1.3.3. Composability requires that applications are independent in both the value and time domains. The proposed memory controller only explicitly 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 it 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.

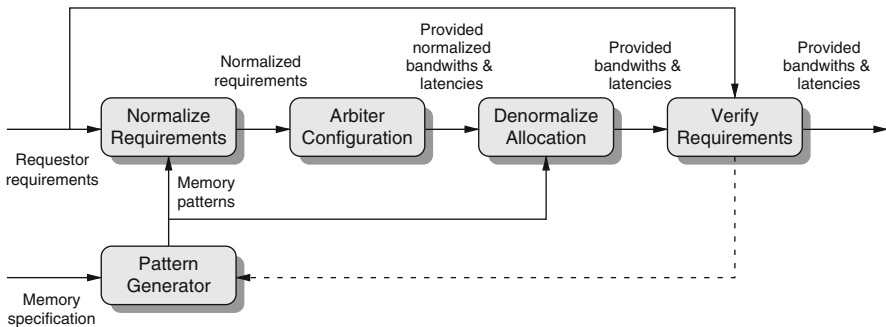
We implement this concept by adding an additional component, called a Delay Block, to the architecture in Fig. 2.6. This component embeds the Request and Response Buffers and contains the additional functionality to implement composable service according to our approach. The refined architecture, providing 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 it 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.

Memory Controllers for Real-Time Embedded Systems

Predictable and Composable Real-Time Systems

Akesson, B.; Goossens, K.

2012, XXII, 222 p., Hardcover

ISBN: 978-1-4419-8206-3