A Reconfigurable Real-Time SDRAM Controller for Mixed Time-Criticality Systems

Sven Goossens, Jasper Kuijsten, Benny Akesson, Kees Goossens
Eindhoven University of Technology
{s.l.m.goossens,k.b.akesson,k.g.w.goossens}@tue.nl

ABSTRACT

Verifying real-time requirements of applications is increasingly complex on modern Systems-on-Chips (SoCs). More applications are integrated into one system due to power, area and cost constraints. Resource sharing makes their timing behavior interdependent, and as a result the verification complexity increases exponentially with the number of applications. Predictable and composable virtual platforms solve this problem by enabling verification in isolation, but designing SoC resources suitable to host such platforms is challenging.

This paper focuses on a reconfigurable SDRAM controller for predictable and composable virtual platforms. The main contributions are: 1) A run-time reconfigurable SDRAM controller architecture, which allows trade-offs between guaranteed bandwidth, response time and power. 2) A methodology for offering composable service to memory clients, by means of composable memory patterns. 3) A reconfigurable Time-Division Multiplexing (TDM) arbiter and an associated reconfiguration protocol. The TDM slot allocations can be changed at run time, while the predictable and composable performance guarantees offered to active memory clients are unaffected by the reconfiguration. The SDRAM controller has been implemented as a TLM-level SystemC model, and in synthesizable VHDL for use on an FPGA.

1. INTRODUCTION

Modern Systems-on-Chips (SoCs) are growing in complexity. The number of applications mapped to a single system is increasing, and they have to share resources due to power, area and cost constraints [10]. Some applications may have real-time constraints, while others may not, creating systems of mixed time-criticality. Applications can run simultaneously in a large number of different combinations or *use-cases*, and can dynamically be started or stopped. The system has to be verified in each of these use-cases to guarantee that application requirements are always satisfied. The verification complexity grows exponentially with the number of applications if they can be combined arbitrarily.

When the active use-case changes, hardware components may have to be reconfigured to adapt to the new set of applications and requirements. These transitions between different use-cases add to the analysis complexity, since running applications must show correct behavior even during reconfiguration. Combining applications of mixed time-criticality and different models of computation makes finding a common analysis model very difficult. Even if such a model exists, a single change in an application requires all use-cases in which the application is active to be re-verified in general.

A solution for the verification problem is to give each application its own *virtual platform* [6, 12]. Literature distinguishes two types of virtual platforms. The first is a *predictable* virtual platform that guarantees a certain budget to an application on each resource that it uses, such that useful bounds on the worst-case response time (WCRT) can be derived. This limits its applicability to real-time applications that are formally analyzable.

The second type of virtual platform provides complete temporal isolation for an application, making its cycle-by-cycle execution independent of other applications. Such platforms are called *composable* and allow independent verification by execution in isolation, because the application's behavior will not change once it is integrated. This property is useful in mixed real-time systems, where execution of test vectors can be the only available verification method for some applications. Once an application has been verified on its own on a composable virtual platform, it is also guaranteed to work when it shares resources with other applications, as long as the virtual platform remains unchanged. Virtual platforms are mapped on predictable and composable resources, and the challenge lies in the design of such resources.

This paper focuses on creating a reconfigurable predictable and composable SDRAM controller. Existing solutions are static, configured once for a single use-case without considering use-case transitions. We improve on the state-of-the-art with the following contributions: 1) An SDRAM controller architecture with a run-time reconfigurable command scheduler. This configuration is a trade-off between worst-case bandwidth, WCRT or power consumption, as shown in [7]. Our reconfiguration infrastructure enables changing this trade-off per use-case at run time. 2) A methodology for offering predictable and composable service to memory clients using this memory controller. We introduce *composable memory patterns* as a way to eliminate interference between memory requests. 3) A reconfigurable TDM arbiter and reconfiguration protocol that allows run-time reallocation of the TDM slots in a predictable and composable way. Implementations of the memory controller and TDM arbiter in SystemC and on an FPGA are used to experimentally verify that it gives predictable and composable service to all clients even during use-case transitions. We show that the proposed controller is a suitable building block for virtual platforms.

The rest of this paper is organized as follows: In Section 2,

this paper is positioned with respect to related work. Section 3 provides background information on SDRAMs and related concepts. The contributions start in Section 4, where the architecture of the controller is presented. Section 5 shows how the controller is used to create a predictable and composable SDRAM resource, followed by a discussion of the reconfigurable arbiter and reconfiguration protocol in Section 6. Finally, experiments are shown in Section 7, and we end with conclusions in Section 8.

2. RELATED WORK

Several real-time memory controllers have been proposed in related work. In [3], a completely static memory command schedule is derived at design time. The controllers from [2, 15, 16] are more flexible and all provide a degree of virtualization. The *PRET DRAM controller* [16] partitions the SDRAM into multiple resources. Access to the individual resources is interleaved using TDM, which makes them timing independent. The number of partitions that can be created is limited by number of banks and ranks on the SDRAM module, and a second level of arbitration is required if there are more clients than partitions. The *Analyzable memory controller* [15] dynamically schedules memory commands according to a set of rules from which an upper bound on the execution time of a request is determined. It does not aim to provide composable virtualization. Instead, when an application is verified, the controller is temporarily set to a mode where it emulates worst-case interference. Such a verification is only valid under the assumption of performance monotonicity, where faster service always implies increased performance. This only holds for limited models of computation, and is not necessarily true for complex architectures and adaptive applications. In [2], a controller is shown that dynamically schedules precomputed sequences of SDRAM commands according to a fixed set of scheduling rules. Through a design-time analysis, a *latency-rate* bound [18] on the performance provided to each application is determined, creating predictable virtual resources. A hardware delay-block can use this worst-case bound to delay the response for each request until its WCRT , turning this predictable SDRAM controller into a composable resource.

All mentioned controllers calculate their configuration and allocation settings per use-case, and transition behavior between use-cases is not considered.

The *PARDIS* programmable memory controller [4] is reconfigurable in several respects. Two small processors with custom instruction set architectures take the role of memory controller, their firmware determining the command scheduling policy, address mapping, refresh scheduling and power management. However, no bounds on performance are given, so applying it in a real-time system is not straightforward. This holds for most non-real-time memory controllers.

An OS-based bandwidth reservation system suitable for mixed real-time Commercial off-the-shelf systems is shown in [23], but it does not offer composable service.

Two strategies for dealing with reconfiguration of resources with real-time service guarantees can be distinguished. The first strategy requires assumptions on the frequency of reconfiguration events to analytically bound their interference [5, 14]. The second strategy constrains the reconfiguration process such that the guaranteed performance during reconfiguration is not worse than during regular operation [19]. In [14], task-level WCRT analysis for multi-mode applications that share resources in multi-core systems is discussed. Mode changes are defined as changes in the set of active tasks or applications, which we refer to as use-cases switches in this paper. The resource arbitration mechanism that is used involves software-based critical sections combined with priorities. Interference due to reconfiguration is bounded by limiting the number of active mode changes to one. Contrary to our approach, intimate knowledge of the task-scheduling policy and critical sections within a task is required to ensure safe reconfiguration. [5] presents a reconfiguration method for soft real-time applications. The hardware offers no service during reconfiguration, but the reconfiguration time is bounded at design time. This approach is not composable since reconfiguration influences all running applications.

The work presented in [19] describes reconfiguration algorithms for TDM-based servers while guaranteeing schedulability of the client applications. The algorithms assume server time can be continuously allocated, and by carefully choosing the location of the unallocated server time and the length of transition periods, predictable performance bounds are given. The algorithms are not applicable to composable resources, since scheduling times may vary as a result of reconfiguration. The reconfigurable TDM-based network-on-chip proposed in [8] provides composable service to selected clients during reconfiguration, but other clients can experience reduced performance. Our work provides an additional predictable service level with bounded worst-case performance even during reconfiguration.

In contrast to related work, this paper present a reconfigurable SDRAM controller suitable for mixed real-time systems. Both predictable and composable service can be offered, allowing the corresponding types of virtual platforms to use the SDRAM resource. Access to the resource is regulated by a TDM arbiter, which can be reconfigured while the service level for running applications remains constant. The resource is modeled as a latency-rate server, and we formally prove that behavior during reconfiguration is not worse than during regular operation.

3. BACKGROUND

This section provides background information on SDRAM in general, an approach that turns it into a predictable resource, latency-rate servers, and our definition of use-case requirements.

3.1 SDRAM

SDRAM is a type of memory that is widely used as the first level of off-chip storage for SoCs . An SDRAM device is hierarchically split into a number of *banks* (typically four or eight). Banks share a common command, data and address bus, but can further operate independently. An SDRAM request starts with an activate command, which opens a *row* from the memory such that it can be read from or written to. Each read or write command results in a *burst* of data from a number of *columns* in the open row. Rows are closed with an explicit precharge command or by attaching an *auto-precharge* flag to a read or write command. Only one row in each bank can be open at any time. An SDRAM is volatile, which means data has to be refreshed regularly using a *refresh* command at an interval of *tREFI* cycles, typically every 7.8 μs. A large set of standardized *timing constraints* exists that dictate the minimum distance between the different commands, based on the state of the SDRAM [9]. It is the task of the memory controller to make sure these constraints are all satisfied.

3.2 Predictable SDRAM resource

To create a *predictable* SDRAM resource, useful bounds on the execution time of memory requests have to be given. For this we choose to use the approach from [2], where the controller translates each request into a design-time constructed series of SDRAM commands with a known execution time, called a *pattern*. Six different basic patterns exist: *read*, *write*, *read-to-write switch*, *write-to-read switch*, *refresh* and *idle* patterns. The read and write patterns are *access patterns* that transport data from and to the SDRAM. They are constructed such that they can be repeated after themselves without violating SDRAM timing constraints. A *close-page policy* is used, meaning all banks are precharged at the end of a pattern. Switching patterns consist of only NOPs. They are inserted between read and write patterns to resolve timing constraints that span across them. A refresh pattern consists of a single refresh command preceded and succeeded by enough NOPs such that it can be scheduled after an access pattern without violating timing constraints. Based on the pattern lengths and their scheduling rules, the worst-case execution time (WCET) of a request can be determined. One of the contributions of this paper is to extend this approach such that the SDRAM resource also becomes composable, without using hardware delay-blocks to eliminate interference [2, 15].

A pattern set has a *worst-case efficiency* [1], which is the fraction of the theoretical peak bandwidth that it guarantees in the worst case. The *gross bandwidth* is the product of the efficiency and the peak bandwidth. It describes the total bandwidth that is distributable amongst the memory clients.

A memory request has a type (read or write), an address and a size. The smallest memory request size in most systems is larger than the size of one read or write burst. This property is used to create more efficient patterns that exploit bank-parallelism by interleaving read and write commands over multiple banks. The number of *banks a request is interleaved* across is denoted by BI. Orthogonally, executing multiple read or write bursts to the same bank within a pattern can also increase the efficiency. The *burst count* (BC) parameter denotes how many bursts are performed to each bank per access pattern. Each BI and BC combination leads to a different pattern set with its own real-time properties and power consumption [7]. The product of BI and BC selects the total number of bursts in a pattern and thus the *access granularity* (AG) of the controller, which is the minimum number of bytes that can be efficiently read from or written to the memory.

3.3 Latency-rate servers

The SDRAM resource is shared using a predictable arbiter. To analyze the arbiter's behavior, we use the latency-rate ($\mathcal{LR}$) server abstraction [18]. A $\mathcal{LR}$ server guarantees a client a minimum allocated rate (bandwidth), ρ, after a maximum service latency (interference) Θ, as shown in Fig. 1. This linear service guarantee bounds the amount of data that can be transferred during any interval, independently of the behavior of other clients. The value of Θ and ρ depend on the arbiter and its configuration.

The $\mathcal{LR}$ guarantee is conditional only applies if the client produces enough requests to keep the server busy. This is captured by the concept of *busy periods*, which are periods where a client requests at least as much service as it has been allocated on average (ρ). In Fig. 1, the client is busy as long as the requested service curve stays above the busy

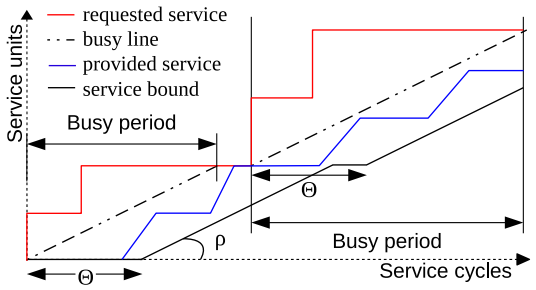

Figure 1: A $\mathcal{LR}$ server and its associated concepts.

line. Note that the service bound is maximal if the client continuously remains busy.

A TDM arbiter divides the resource time into *slots* that are distributed to multiple clients. Each slot represents a time slice in which one client can use the resource. A slot is non-preemptive, but its length is bounded by the WCET of a request. We assume allocation of slots to clients is done at design time, yielding a slot table that maps each slot to a certain client. The length of the slot table, or frame, (f) defines the period of the arbiter. Each slot corresponds to a fraction $1/f$ of the gross bandwidth.

Intuitively, the service latency expressed in slots (Θ_{slots}) is the worst-case number of slots a client has to wait until it reaches one of its slots. If a TDM arbiter uses continuous (greedy) allocation, this is equal to f times the rate not allocated to this client $(1-\rho)$, plus one, as shown in Eq. (1). The plus one accounts for the misalignment of the arrival of a request with the arbitration moments. In the worst case, a decision has been made one cycle before the arrival, and the client is too late to claim its slot.

$$\Theta_{\text{slots}} = f \cdot (1 - \rho) + 1 \tag{1}$$

The service latency only captures the maximum interference by other clients, but does not consider interference from refresh operations. To convert this number to an actual bound on the WCRT, the approach from [1] is used. In brief, it considers an interfering refresh at the start of the busy period, plus an interfering refresh for each refresh interval that fits within the client's service latency. The WCET of a request, which defines the worst-case slot length, is used to convert this latency from slots to actual time.

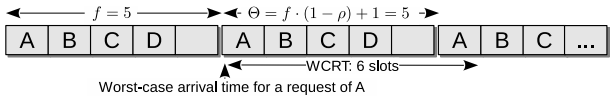

Figure 2: Three example TDM table iterations.

3.4 Use-case requirements

A use-case is defined as a set of concurrently running applications that share the memory resource. If arbitrary combinations of applications are allowed, then the number of use-cases is exponential in the number of applications. In the worst case, they are all active at the same time, and the system has to be dimensioned accordingly. However, in many practical systems this is not the case, as shown earlier by [8]. For example, applications that have similar functionality may never have to run simultaneously, and can use the same hardware resources. Furthermore, all use-cases that

are sub-sets of a larger use-case do not need separate configurations if they can partially re-use the configuration of their super-set use-cases. The largest super-sets of applications that must be able to run simultaneously are called a *maximum cliques*. This is the granularity at which configurations are generated in this paper. Applications can be distributed and map to one or more memory clients. The requirements of those clients have to be satisfied by finding a valid controller configuration in each maximum clique.

In this paper, we assume each application has known bandwidth and response times requirements. Finding these requirements for an application is non-trivial. For applications having a formal model, they can be derived from application-level throughput or response time requirements [13,20], while others might require simulation-based techniques. However, a detailed discussion is out of the scope of this paper.

4. CONTROLLER ARCHITECTURE

The SDRAM controller presented in this paper is based on the template proposed by [2]. All the components in the template are briefly discussed, and changes made as part of the contributions in this paper are mentioned explicitly.

A TLM-level SystemC version of the controller is implemented to allow rapid prototyping with high debuggability. Based on the SystemC prototype, a VHDL version targeting a Virtex 6 ML605 development board [22] from Xilinx was created.

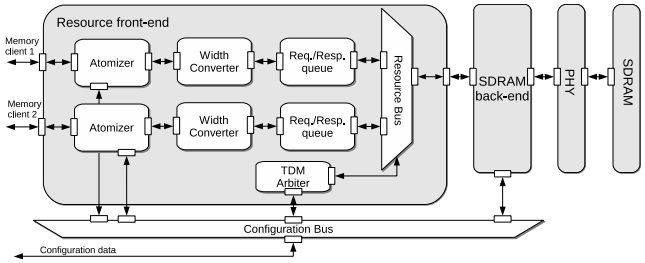

Figure 3: Example memory controller instance.

Fig. 3 shows the three main blocks that constitute the memory controller architecture. Working backwards from the SDRAM itself they are the *PHY*, *SDRAM back-end* and the *front-end*. Although the figure only shows two memory clients, up to 16 ports can be instantiated automatically by the associated design flow if required.

The PHY handles the physical I/O connections to the SDRAM module. This block was not required at the level of abstraction used in the template, and is also omitted in our SystemC version. The VHDL version's PHY is based on a reference design generated by the Xilinx MIG 3.6 tool.

The SDRAM back-end interfaces with the PHY and is responsible for generating commands to access the memory according to the incoming requests, while making sure that the timing constraints between the commands are satisfied. It translates logical addresses to a physical bank, row and column in the memory, and it also refreshes the memory every tREFI cycles [9]. In contrast to [2], which uses a hard-coded finite-state machine to implement the command scheduler, this paper proposes a flexible reconfigurable back-end, which is discussed in detail in Section 6.2.

The primary function of the resource front-end is enabling sharing of the SDRAM. The *atomizer* splits incoming memory requests into fixed size chunks called *atoms*. This allows clients to be preempted at the granularity of atoms, independently of their actual request behavior. The size of an atom corresponds to the granularity at which the back-end handles requests, which typically ranges from 16 bytes up to 1 KB, depending on its configuration. The configuration port on the atomizer allows the access granularity to be reconfigured at run time.

The *width converter* accepts messages using small datawords from the memory clients (generally 32-bits wide), and converts them to the width the back-end is working at. In essence, this is a common serial-to-parallel converter. This block is new with respect to the template, enabling connecting to memories with wider interfaces than used in the SoC.

The *request/response buffer* holds incoming requests until either all data for an atom is buffered (for write requests), or enough space is available for the response (for read requests). This is required because data has to be provided to and accepted from the back-end without blocking according to the JEDEC specification [9]. A request is only eligible for scheduling once this condition is met.

The novel reconfigurable TDM arbiter schedules one of the eligible requests from the request buffers to be processed by the back-end, and optionally inserts an idle slot if no request is available. The memory is shared at a fine granularity, such that each slot corresponds to one read or write atom. Section 6.1 discusses this in more detail.

The configuration bus allows various memory-mapped registers to be programmed by a configuration host. In the SystemC version, this role is taken by a dedicated resource manager module, while the VHDL version uses a MicroBlaze processor. The derivation of configurations and the reconfiguration procedure is shown later in Section 6.

5. COMPOSABLE AND PREDICTABLE SDRAM RESOURCE

This section discusses how the predictable pattern-based memory controller is extended to create a composable SDRAM resource, in which requests from separate memory clients are temporally isolated. The key idea is to *share the SDRAM through TDM arbitration, and to make the start of a client's time slots independent from other clients.* To ensure that a slot always starts at the same time, all slot lengths have to be equal regardless of the request type or the presence or absence of an eligible request. Also, the state of the memory must return to neutral after each request, such that following requests are not constrained by previous requests from other clients. This implies that a close-page policy must be used. The influence of the request type must also be eliminated, meaning that the timing constraints that allow *both* read and write requests must be satisfied at the end of the slot. To meet these requirements, the predictable memory patterns are converted to *composable patterns*. Section 5.1 discusses that process, after which performance bounds for these patterns are derived in Section 5.2.

5.1 Composable memory pattern generation

Composable memory patterns are constructed at design time in a similar manner as predictable patterns. The goal is to create composable read and write patterns that can be scheduled arbitrarily without violating timing constraints, and are equal in length. Their length determines the length of one TDM slot.

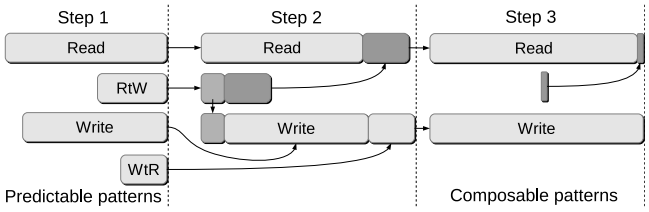

Figure 4: Composable pattern generation example.

The composable patterns are generated in three steps. The first step generates a predictable pattern set using the algorithms described in [1]. The last two steps make the pattern set composable. Fig. 4 shows the relation between a predictable pattern set and its composable counterpart. We proceed by discussing each of these steps in more detail:

1. Read and write patterns are generated based on a known command scheduling algorithm like ASAP scheduling or bank scheduling [1]. NOPs are added at the end of these patterns, such that they can be repeated after themselves without violating SDRAM timing constraints. This determines the minimum length of the predictable access patterns. Based on this, the lengths of the switching patterns is determined.
2. Composable pattern sets cannot contain switching patterns, since they introduce timing dependencies on the previous request type. Instead of having separate switching patterns, the required NOPs are distributed amongst the read and write patterns. NOPs resolving read-to-write (RtW) constraints can be added at the end of the read pattern or the beginning of the write pattern, while NOPs resolving write-to-read (WtR) constraints can be added at the end of the write pattern or the beginning of the read pattern. Equalizing the lengths of the access patterns reduces the switching overhead in terms of bandwidth, so the NOPs are distributed to balance the patterns lengths as much as possible.
3. Finally, any length difference that still remains between the read and write pattern has to be compensated by adding NOPs at the end of the shortest pattern.

The idle pattern is made equal to the composable read or write pattern length, which asserts that all slots always take the same number of cycles. Refresh patterns are inserted at the end of a regular slot after every tREFI cycles. The actual insertion time not influenced by the running applications since all slots are equally long, meaning refresh is also composable. The impact of this conversion on the worst-case performance is shown in the next section.

5.2 Predictable performance bounds

The worst-case analysis for predictable patterns is based on the notion of worst-case efficiency. To evaluate the performance of composable patterns, the efficiency loss with respect to the corresponding predictable pattern set has to be determined. The lengths of the predictable read, write, write-to-read and read-to-write patterns are denoted by t^p_r, t^p_w, t^p_{wtr} and t^p_{rtw}, while the composable access pattern lengths are denoted by t^c_r and t^c_w, respectively. We need to distinguish three different cases, depending on the length of the predictable patterns:

1. If the read pattern is longer than the write pattern and both switching patterns combined, then the worst-case request sequence consists of only read requests, and the pattern set is *read dominant*. The composable access patterns are as long as the predictable read pattern, $t^c_r = t^c_w = t^p_r$.
2. If the write pattern is longer than the read pattern plus both switching patterns, then the worst-case request sequence consists of only write requests, and the pattern set is *write dominant*. The composable access patterns are as long as the predictable write pattern, $t^c_r = t^c_w = t^p_w$.
3. Pattern sets that do not fit in class 1 or 2 show worst-case behavior if read and write requests are alternated. These pattern sets are *mix dominant*. The pattern set in Fig. 4 is an example of this class. In this case $t^c_r = t^c_w = \lceil (t^p_r + t^p_w + t^p_{wtr} + t^p_{rtw})/2 \rceil$.

Table 1: (e_{pc}) $\left(\frac{\textbf{composable efficiency}}{\textbf{predictable efficiency}}\right)$ for a range of SDRAM x16 devices

BI	1	1	1	1	2	4	1	2	4	8
BC	1	2	2	4	2	1	8	4	2	1
AG [bytes]	16	32	32	64	64	64	128	128	128	128
DDR2-400	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
DDR2-800	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	0.98	0.98
LPDDR-266	1.0	1.0	1.0	1.0	0.97	0.97	1.0	1.0	0.98	n/a
LPDDR-400	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	n/a
DDR3-800	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.98	0.98	0.98
DDR3-1600	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
LPDDR2-800	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
LPDDR2-1066	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.99

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

If the composable pattern set is based on a mix dominant pattern set, then the worst-case efficiency is only affected if the two switching patterns are smaller than the length difference between the read and write pattern, and NOPs had to be added in Step 3 to balance the patterns. At most one NOP is required for this by definition, or else the pattern set would not be mix dominant. If $t^p_r + t^p_w + t^p_{wtr} + t^p_{rtw}$ is odd, then the conversion efficiency (e_{pc}) for mix dominant pattern sets can be expressed as:

$$e_{pc} = \frac{t^p_r + t^p_w + t^p_{wtr} + t^p_{rtw}}{1 + t^p_r + t^p_w + t^p_{wtr} + t^p_{rtw}} \quad (2)$$

In all other cases $e_{pc} = 1$, and the composable pattern set efficiency is equal to the predictable pattern set efficiency.

The conversion efficiencies for a representative set of SDRAM devices from different generations and of various clock frequencies are shown in Table 1. All shown memories use a 16-bit interface width, and access granularities (AG) up to 128 bytes are considered. There are no BI 8 results for LPDDR memories since they only have 4 banks. The maximum efficiency loss is observed for LPDDR-266 (2.6%). Only pattern sets that require switching patterns are susceptible to efficiency loss. Switching patterns are usually required for patterns that implement large access granularity and have a higher inherent efficiency. The slower the memory, the smaller the access granularity has to be to reach high efficiency, which explains why the slower memories are relatively more likely to suffer, but the timing constraints on which the patterns are based determine the actual losses.

The average loss due to the conversion is 0.22%, so the typical efficiency loss is negligible.

6. RECONFIGURATION

When an application is started or stopped, the active use-case changes. This leads to a change in the system state that is coordinated by a resource manager, which could be either part of an operating system running on a processor in the SoC, or a dedicated hardware module, depending on the required flexibility. A use-case switch leads to reconfiguration if the required configuration in the source use-case and target use-case differ. This section considers the reconfiguration process of the SDRAM controller.

All the clients that are not active in the target use-case are handled first during a use-case switch. Because they are switched off, no service guarantees have to be given to them during and after reconfiguration. The resource manager first stops the clients at the processor side. It then triggers a final dummy-read request and waits for the response to assert that no requests for the client are left in the system. Since requests are never reordered in this implementation, quiescence is thus asserted [11] before reconfiguration is initiated.

Next, we handle clients that are active in both the source and the target use-case. They can be categorized further into clients that require predictable service or composable service. The challenge in reconfiguration is maintaining the predictable and composable service level of applications during and after reconfiguration. The following two sections discuss the reconfiguration procedure and the required hardware to meet this challenge.

6.1 Reconfigurable TDM arbiter

This section discusses the TDM based arbiter used to share the SDRAM resource across multiple memory clients. Section 6.1.1 describes how arbiter configurations are generated based on the memory clients' requirements. Section 6.1.2 then briefly discusses the arbiter architecture, followed by an explanation of the reconfiguration protocol in Section 6.1.3.

6.1.1 Configuration generation

We assume that a memory client has a certain bandwidth and WCRT requirement that have to be satisfied by the bandwidth and WCRT guarantees of the controller. What these guarantees are depends on the chosen pattern set and the arbiter configuration. The more slots that are allocated to an application, the higher its guaranteed bandwidth and the lower its WCRT is. If for example, a TDM table size of 5 is used (Fig. 2), then each slot corresponds to 1/5'th of the gross bandwidth.

Several slot-allocation strategies that consider real-time constraints exist [17, 21], but they are not the focus of this paper, so we use continuous allocations. We also assume a fixed table size set by the system designer. Based on the specified use-cases, a distinct number of maximum cliques can be found for which allocations are generated at design time. Clients that need composable service require the same allocation across all use-cases in which they are active. They cannot be moved around since this would make their behavior use-case dependent. Composable clients are mapped first, and their slot-allocation is the same in all cliques where they are active. Predictable clients may have different allocations in different maximum cliques, as long as the performance guarantees during reconfiguration are not worse than during normal operation. How this can be guaranteed is explained in Section 6.1.3. Predictable clients are mapped in slots that remain after allocating the composable clients. If the allocation fails at any point, we rely on the designer to change the table size or reduce the load of the failing maximum clique. An allocation example is given in Section 7.2.

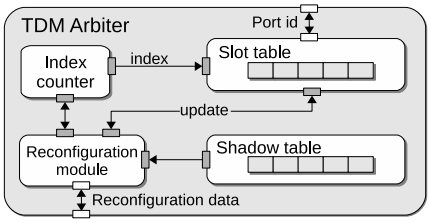

Figure 5: TDM arbiter architecture.

6.1.2 Arbiter architecture

The arbiter architecture is shown in Fig. 5. It consists of a set of registers that represents the active TDM *slot table*. Each slot entry contains the bus-port id of the client to which it belongs. An incrementing wrapping *index counter* selects the next client to be scheduled from the slot table. Both the wrap-around value and the interval at which scheduling decisions are made are configurable.

An extra copy of the TDM table is kept in the *shadow table*, which can be reprogrammed through a configuration port. All slot reconfigurations are first applied to the shadow table. One configuration message can reassign a continuous slot range in the shadow table to a different client. The shadow table is locked from further updates after each configuration message until its contents are copied to the slot table. The purpose of the *reconfiguration module* is to implement our safe reconfiguration protocol. It delays the actual reconfiguration of the slot table until the index counter wraps around. Only then the contents of the shadow table are copied to the slot table, and the new configuration immediately takes effect. If a predictable client is reconfigured to a different set of slots, then we use two configuration messages to perform this action: 1) the new slots are enabled, 2) the old slots are disabled. The implemented reconfiguration mechanism forces the transition phase where both the new and old allocation are given to be at least one table iteration (f). By doing so, we can guarantee that reconfiguration is safe and does not violate the latency-rate guarantees of the client (Fig. 8). In the next section, we formally prove this.

6.1.3 Reconfiguration protocol

Predictable clients can have distinct slot allocations in each maximum clique. The arbiter may be reconfigured to switch between these allocations, while the client continues to run. If reconfiguration is not regulated correctly, the amount of service the client receives may reduce, as shown in Fig. 6, where the response time for the request of A increases from 6 to 10 slots as a result of reconfiguration. This could mean its $\mathcal{LR}$ guarantees are violated, depending on the tightness of its service bound.

This paper concentrates on reconfiguration effects at the slot granularity. We formally prove that the $\mathcal{LR}$ guarantees at this level of abstraction are not invalidated if our reconfiguration protocol is used. This is a sufficient condition to guarantee that the $\mathcal{LR}$ bound expressed in clock cycles is also valid, since the transformation function from slots to clock cycles is monotonically increasing [2].

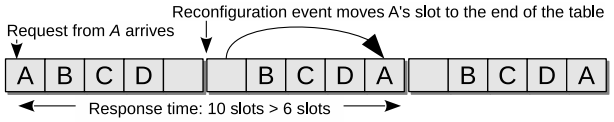

Figure 6: Example of potentially violated $\mathcal{LR}$ guarantees for client A during reconfiguration.

A $\mathcal{LR}$ server offers a linear lower bound on the provided service within a busy period [18].

Definition 1. Let τ be the starting time of a busy period for server s_i with a service latency Θ_i and allocated rate ρ_i. For any time t until the end of this busy period, a lower-bound on the provided service by s_i is given by:

$$w_i(t) = \max(0, \rho_i \cdot (t - \tau - \Theta_i)) \tag{3}$$

A property of a $\mathcal{LR}$ guarantee is that it is maximal if the client is continuously busy. If reconfiguration does not lead to a violation of the bound under this condition, it is also safe in all other cases. If a $\mathcal{LR}$ server is reconfigured, for example by changing the underlying TDM slot allocation, its Θ and ρ may change. We assume the allocations before and after reconfiguration are chosen such that they satisfy the $\mathcal{LR}$ requirements of the client. This is satisfied by our design flow.

Definition 2. The required $\mathcal{LR}$ service bound of the client, $w_r(t)$ is given by:

$$w_r(t) = \max(0, \rho_r \cdot (t - \tau - \Theta_r)) \tag{4}$$

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

Definition 3. Let c_1 and c_2 be two different TDM slot allocations for a client. The corresponding $\mathcal{LR}$ parameters for allocation c_1 and c_2 are denoted by (Θ_1, ρ_1) and (Θ_2, ρ_2), respectively. Both of these parameter sets satisfy the client's $\mathcal{LR}$ requirements, such that $\rho_r \leq \min(\rho_1, \rho_2)$ and $\Theta_r \geq \max(\Theta_1, \Theta_2)$.

We only consider the case where c_1 and c_2 do not overlap. A similar analysis is possible in case there is overlap by considering only the non-overlapping slots as different allocations and adding a third constant allocation representing the overlapping slots, but we do not show it here for space reasons. We assume that c_1 is initially active. To model the behavior of a TDM arbiter while it is reconfigured from c_1 to c_2, we define two time instances: t_A is the time at which allocation c_2 is enabled in the slot table, t_R is the time at which allocation c_1 is disabled in the slot table.

The total service guaranteed by a TDM-based server to a client is equal to the sum of the service provided by each slot that is allocated to it. This property allows us to describe the guaranteed service during reconfiguration as the sum of the service provided by allocations c_1 and c_2. Combining Definition 1 and 3 yields:

Definition 4. For a time t during a busy period, the service guarantee of a server that is reconfigured from c_1 to c_2 is given by:

$$\begin{aligned} w_g(t) = &\max(0, \rho_1 \cdot (\min(t, t_R) - \tau - \Theta_1)) \\ &+ \max(0, \rho_2 \cdot (t - \max(\tau, t_A) - \Theta_2)) \end{aligned}$$

The required $\mathcal{LR}$ service bound may not be violated before, during or after reconfiguration. In other words, $w_g(t)$ has to be larger or equal than $w_r(t)$ for all t. This is formally proven in Theorem 1.

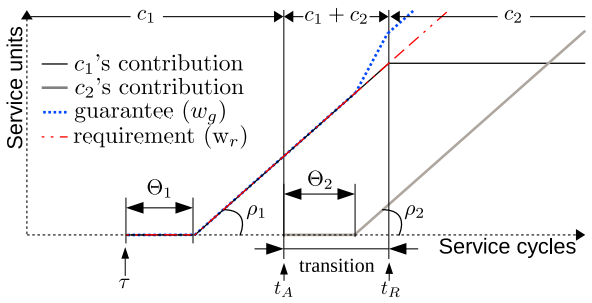

Figure 7: Example of the latency-rate guarantees during reconfiguration.

THEOREM 1. *If* $t_R - t_A \geq \max(\Theta_1, \Theta_2)$ *then* $\forall t$, $w_g(t) \geq w_r(t)$.

PROOF. In the general case, we have to assume there is no over-allocation, such that $\rho_1 = \rho_2 = \rho_r$. We can also conservatively substitute Θ_1 and Θ_2 by $\Theta' = \max(\Theta_1, \Theta_2)$. This means that

$$\max(0, \rho_r \cdot (\min(t, t_R) - \tau - \Theta')) + \tag{5}$$
$$\max(0, \rho_r \cdot (t - \max(\tau, t_A) - \Theta')) \geq \tag{6}$$
$$\max(0, \rho_r \cdot (t - \tau - \Theta')) \tag{7}$$

has to hold for all t.

Case 1: As long as c_1 is not disabled ($t \leq t_R$), this inequality is satisfied by the contribution of the server corresponding to the first use-case (5). Case 2: Similarly, if the busy period starts after allocation c_2 is enabled ($t_A \leq \tau$), then the inequality is satisfied by the contribution of the server corresponding to the second use-case (6). Case 3: As long as $w_r(t)$ is 0 ($t \leq \tau + \Theta'$), the inequality is also satisfied.

Case 4: This only leaves the inverse of the union of the previous three cases: $t > t_R$ and $t_A > \tau$ and $t > \tau + \Theta'$. Applying these case constraints to the main inequality:

$$\begin{aligned} \max(0, \rho_r \cdot (t_R - \tau - \Theta')) + \max(0, \rho_r \cdot (t - t_A - \Theta')) \geq \\ \rho_r \cdot (t - \tau - \Theta') \end{aligned} \tag{8}$$

Both max-terms have to contribute to satisfy the equation for all t, so for now we have to assume that both

$$t_R > \tau + \Theta', \text{ and} \tag{9}$$
$$t > t_A + \Theta' \tag{10}$$

hold. Removing the common terms from Eq. (8) leaves:

$$t_R - t_A \geq \Theta' \tag{11}$$

If we assert that Eq. (11) holds, then $w_g(t) \geq w_r(t)$ holds in Case 4, given that Assumptions (9) and (10) are true. Combining Eq. (11) with case constraint $t_A > \tau$ yields (9), and combining Eq. (11) with case constraint $t > t_R$ yields (10), confirming the assumptions.

This means that if Eq. (11) holds, then $w_g(t) \geq w_r(t)$ holds for all t which concludes the proof. □

Equation (11) enforces a minimum interval of $\max(\Theta_1, \Theta_2)$ where both the c_1 and c_2 have to be provided by the server. During that transition period, the server temporarily assigns both slot allocations to the client. Fig. 7 illustrates this. By moving t_R further to the left, the interval in which w_g is larger than w_r will get smaller, until $t_R - t_A = \max(\Theta_1, \Theta_2)$, where it disappears.

Our arbiter implementation and reconfiguration method forces a minimum transition period of at least one full table iteration (f). To satisfy the safety condition Eq. (11), this has to be larger than Θ expressed as Eq. (1):

$$f \geq \max(\Theta_1, \Theta_2) = f \cdot (1 - \min(\rho_1, \rho_2)) + 1 \qquad (12)$$

This equation is true if $\min(\rho_1, \rho_2) \geq 1/f$. This condition is always satisfied for slot based TDM arbiters, since the minimum allocation for ρ is $1/f$, which means our reconfiguration protocol is safe.

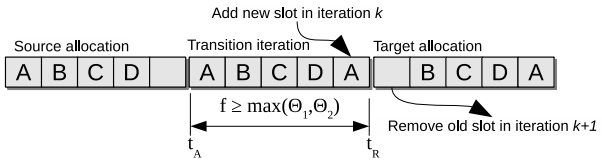

Figure 8: Splitting the reconfiguration in two steps that take place in separate table iterations guarantees that the service during reconfiguration is always greater than the required service.

6.2 Reconfigurable SDRAM back-end

The proposed SDRAM back-end is shown in Fig. 9. It gets requests from the resource bus that consist of a type (read/write) and a logical address. Its main function is to select patterns from the *pattern memory*, and to transfer their commands to the PHY.

An incoming request first arrives at the *pattern selector*. It generates an index for the *pattern Look-Up Table (LUT)* based on the request type (read or write). The index represents the pattern that should be executed. An optional offset can be added to this index to select patterns from different pattern sets. The pattern LUT contains the starting addresses and the number of commands of all patterns in the pattern memory. Its output is used by the command player to read commands from the pattern memory. Both the pattern LUT and the pattern memory are exposed to the resource manager through the configuration bus and are thus reconfigurable. Changing the pattern set to trade-off bandwidth, response times and power consumption [7], implies changing BI and/or BC and reprogramming the address generator. This would prevent running clients from retrieving their data. The pattern set can thus only change if all clients are stopped, i.e. only if there are no active clients during a use-case switch.

Each command in the pattern memory consists of a 6-bit control field and a 3-bit bank field. The control field contains values for the standard RAS, CAS, CS and WE signals, and the value for the 10'th address bit in the physical address, which is the auto-precharge flag. The last bit is reserved for a strobe signal that is specific to the used PHY, and selects the desired data-bus (read/write) direction. The 3-bit bank field specifies the bank for which the command is meant. The pattern memory is implemented as a simple SRAM memory. The *command player* increments the command address every clock cycle, and triggers a new pattern selection when the current pattern ends.

The *address generator* unit translates a logical address to the corresponding bank, row and column addresses. The command player controls the address generator such that the correct address is given to the PHY at the right time, i.e. the row address when activating and the column address during read or write commands. The address generator has

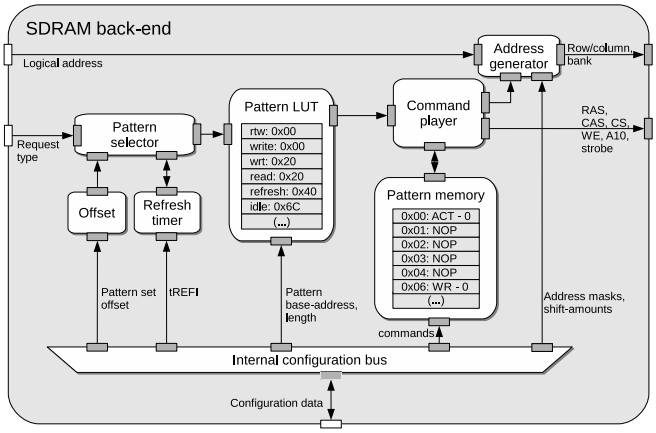

Figure 9: Reconfigurable SDRAM back-end.

four configurable masks/shift units through which the logical to physical memory-mapping function can be selected.

The final block to consider is the *refresh timer*. It consists of a cycle counter with a configurable threshold value. When the counter reaches the threshold, it resets to zero and a refresh is scheduled as soon as the current slot finishes.

7. EXPERIMENTS

Two experiments are presented to demonstrate the SDRAM controller and the reconfiguration protocol proposed in this paper. The first experiment, Section 7.2, uses the SystemC version of the controller, while the second experiment, Section 7.3, uses the VHDL version.

7.1 Hardware setup

All the features that have been presented in Sections 4 through 6 are implemented and tested in SystemC. The VHDL version used in these experiments is less generic, and two options are omitted. 1) The atomizer is not configurable at run time. Instead, a fixed access granularity is configured at design time. 2) The pattern set offset is tied to 0, such that only one set can be stored in the pattern memory at a time. This is done because we do not require single-cycle reconfiguration of the pattern set in our experiments. Instead, we choose to use a smaller pattern memory. Multiple pattern sets can still be used, but it requires the resource manager to re-program the pattern LUT and pattern memory. This still allows for specialization of the pattern set based on power, bandwidth or latency constraints [7], but only in unconnected maximum cliques.

In SystemC, we use a model of the SDRAM module that is used on the FPGA. It contains a memory module based on DDR3-1066 devices that have a combined data-bus width of 64 bits. The module is under-clocked to a command rate of 400 MHz to compensate for the relative slowness of the FPGA fabric. Only 32 of the 64 available data pins are connected in the experiments to reduce synthesis time, such that it effectively behaves like a DDR3-800x32 device. This configuration is similar to that of the default MPMC SDRAM controller by Xilinx for this FPGA board.

The two-port instance of the controller used in the experiments uses 13754 registers, 9543 LUTs and 1 BRAM, which implements the pattern memory. A four-port instance uses 22065 registers, 14016 LUTs and 1 BRAM. Most registers are used for pipelining and to implement the request/response buffers in the front-end that have 256 bytes

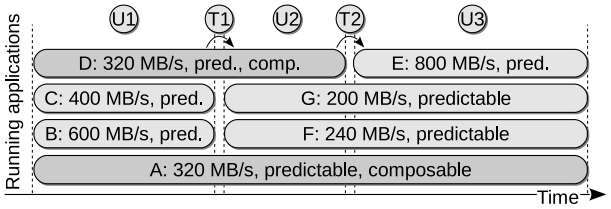

Figure 10: Temporal use-case transition behavior.

of storage space per port. Note that this design was not optimized for area, and buffer sized could be reduced, for example by using techniques from [20].

7.2 Temporal behavior during reconfiguration

Seven synthetic applications (A-G) share the SDRAM resource in this experiment. Fig. 10 shows the properties of the applications and the different use-cases that are traversed. Three maximum cliques are identified, annotated with U1 (A, B, C, D), U2 (A, D, G, F) and U3 (A, E, F, G). At T1 and T2, use-case transitions take place that change the active maximum clique. The applications are implemented by four traffic generators connected to four ports on the memory controller (applications on the same horizontal line are mutually exclusive and share a memory port and traffic generator). Fig. 10 also shows the applications' bandwidth and service requirements. For simplicity of the example, we assume all applications have a relaxed WCRT requirement of 2000 ns and issues only 512-byte requests.

The pattern set that offers most gross bandwidth given the request size requirement is chosen. It uses BI 4 and BC 1, delivers 1862 MB/s and is write dominant, meaning the conversion to composable patterns does not impact worst-case performance. The slot table size is set to 20, such that each slot corresponds to $1862/20 = 93$ MB/s. Because applications A and D run in composable virtual platforms, they get the same slot allocation in all use-cases where they are active. This is reflected in the allocation algorithms' output, which is shown in Fig. 11. Note that without reconfiguration support, the slots for application F and G would not be movable, and application E would be unmappable due to fragmentation.

U1:	A	A	A	A	D	D	D	D	B	B	B	B	B	B	B	C	C	C	C	C
U2:	A	A	A	A	D	D	D	D	F	F	F	G	G	G						
U3:	A	A	A	A	F	F	F	G	G	G	E	E	E	E	E	E	E	E	E	

Figure 11: Slot allocation results.

Fig. 12 shows the temporal behavior of application F in two separate experiments. Each request is chopped into four atoms, and each bar represents an atom, which explains the periodicity. The red x-markers show the WCRT bound for the atom, which varies slightly due to self-interference. The bar height shows the actual measured response time.

In the first experiment (green markers), the safe reconfiguration mechanism in the TDM arbiter is switched off. At 68 μs, the reconfiguration from U2 to U3 takes place. The WCRT bounds of some atoms are violated as a consequence of reconfiguring the arbiter, which is unacceptable.

A second run (blue bars) is performed with the safe reconfiguration mechanism switched on. Here the WCRT is valid, and the actual response time is slightly lower during the reconfiguration process, since the client temporarily gets

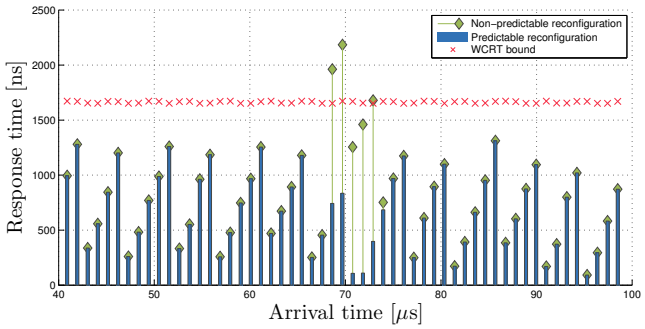

Figure 12: Response times with and without predictable reconfiguration.

more slots. This experiment suggests that our reconfiguration protocol is safe and predictable.

7.3 Composable memory operation

The second experiment shows that composable patterns turn the SDRAM controller into a composable resource. We use a two port VHDL instance of the controller. A pattern set with BI 1 and BC 2 is used, which guarantees a gross bandwidth of 934 MB/s. The slot table size is set to 8.

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

The request/response buffers are instrumented with timers that keep track of the arrival and response times of the requests. These timestamps are recorded and read out after the experiment. For each experiment, we wait until the PHY finishes its self-calibration, and then program the initial configuration in the memory controller. For the purpose of this experiment, the start of the refresh timer and the first arbiter iteration are synchronized to make the behavior across multiple runs repeatable, although this is not strictly required for composability.

Six different runs are performed, divided in two groups of three runs each. In all runs, MB1 gets 4 slots in the table.

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

The first three runs use predictable patterns (Fig. 13), meaning the slot length varies with the request that is executed. Even though application MB1 is not changed across the three runs, its behavior is heavily affected by the interference from MB2. This makes it unfeasible to verify MB1 by simulation in systems with many use-cases.

The second group of runs uses composable patterns (Fig. 14) to effectively eliminate all interference across the two applications. The figure illustrates that MB1 is not affected by any of the actions of MB2, its behavior is constant. The reference run is thus representative for the behavior after integration, and can be used to verify that the application's real-time requirements are satisfied. This enables independent verification of applications in isolation.

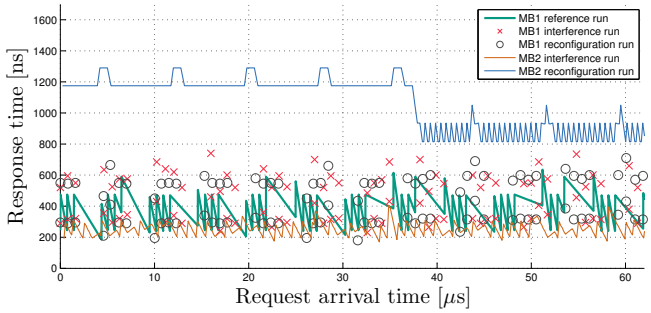

Figure 13: Request latencies using predictable patterns.

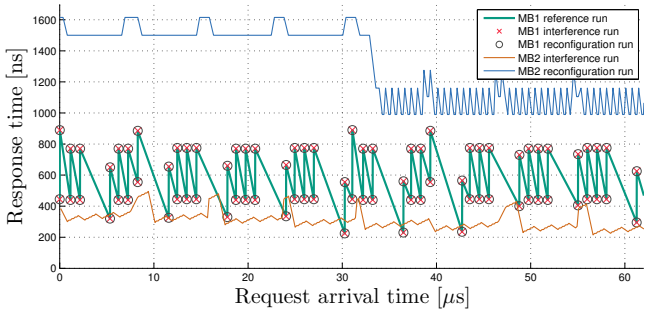

Figure 14: Request latencies using composable patterns.

8. CONCLUSIONS

This paper introduced a run-time reconfigurable SDRAM controller architecture. The command scheduler, based on the concept of memory patterns, is re-programmable, such that a different pattern set can be loaded into the controller to select the best trade-off between bandwidth, response-times and power for the active use-case. The controller offers both predictable and composable service, making it a suitable SDRAM resource for use in virtual platforms. Composability is enabled by the use of composable memory patterns combined with TDM arbitration, and we show that the worst-case performance degradation is negligible. We also introduced a reconfiguration protocol for such an arbiter, which allows run-time changes to the slot table without degrading the guarantees offered to the running memory clients. The controller and proposed methods have been prototyped in SystemC and on an FPGA platform. Experiments show that the controller operates in a predictable and composable manner, even while it is being reconfigured.

9. ACKNOWLEDGEMENTS

This work was partially funded by projects EU FP7 288008 T-CREST and 288248 Flextiles, Catrene CA104 COBRA, CA505 BENEFIC and CA703 OpenES, and NL STW 10346 NEST.

10. REFERENCES

[1] B. Akesson *et al.* Classification and Analysis of Predictable Memory Patterns. In *Proc. RTCSA*, 2010.

[2] B. Akesson *et al.* Architectures and modeling of predictable memory controllers for improved system integration. In *Proc. DATE*, 2011.

[3] S. Bayliss and G. Constantinides. Methodology for designing statically scheduled application-specific SDRAM controllers using constrained local search. In *Proc. FPT*, 2009.

[4] M. Bojnordi and E. Ipek. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *Proc. ISCA*, 2012.

[5] M. Garcia-Valls *et al.* Real-time reconfiguration in multimedia embedded systems. *Consumer Electronics, IEEE Transactions on*, 57(3), 2011.

[6] K. Goossens *et al.* Virtual Execution Platforms for Mixed-Time-Criticality Applications: The CompSOC Architecture and Design Flow. *SIGBED Review*, 2013. To appear.

[7] S. Goossens *et al.* Memory-Map Selection for Firm Real-Time Memory Controllers. In *Proc. DATE*, 2012.

[8] A. Hansson *et al.* Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip. In *Proc. DATE*, 2007.

[9] JEDEC. *DDR3 SDRAM Specification*, JESD79-3E edition, 2010.

[10] P. Kollig *et al.* Heterogeneous Multi-Core Platform for Consumer Multimedia Applications. In *Proc. DATE*, 2009.

[11] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11), 1990.

[12] A. K. Mok *et al.* Real-time virtual resource: A timely abstraction for embedded systems. In *Proc. EMSOFT*, 2002.

[13] O. Moreira *et al.* Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proc. EMSOFT*, 2007.

[14] M. Negrean *et al.* Timing Analysis of Multi-Mode Applications on AUTOSAR conform Multi-Core Systems. In *Proc. DATE*, 2013.

[15] M. Paolieri *et al.* An Analyzable Memory Controller for Hard Real-Time CMPs. *Embedded Systems Letters, IEEE*, 1(4), 2009.

[16] J. Reineke *et al.* PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation. In *Proc. CODES+ISSS*, 2011.

[17] R. Stefan *et al.* An improved algorithm for slot selection in the æthereal network-on-chip. In *Proc. INA-OCMC*, 2011.

[18] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Trans. Netw.*, 1998.

[19] N. Stoimenov *et al.* Resource adaptations with servers for hard real-time systems. In *Proc. EMSOFT*, 2010.

[20] S. Stuijk *et al.* Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proc. DAC*, 2006.

[21] S. Stuijk *et al.* Resource-efficient routing and scheduling of time-constrained streaming communication on networks-on-chip. *Journal of Systems Architecture*, 54(3), 2008.

[22] Xilinx. ML605 Documentation. `http://www.xilinx.com/support/#nav=sd-nav-link-140997&tab=tab-bk`, 2012.

[23] H. Yun *et al.* Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. RTAS*, 2013.