Technische Universiteit
Eindhoven
University of Technology

Department of Electrical Engineering

Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
http://w3.ele.tue.nl/nl/

Series title:
Master graduation paper,
Electrical Engineering

Commissioned by Professor:
prof.dr. Kees Goossens

Group / Chair:
Electronic Systems

Date of final presentation:
January 8, 2013

Report number:

A Reconfigurable SDRAM Controller for Verifiable Mixed Time-Criticality Systems

by

Author: Jasper J.A. Kuijsten

Internal supervisor: Sven Goossens M.Sc.

External supervisor: Dr. Benny Åkesson

A Reconfigurable SDRAM Controller for Verifiable Mixed Time-Criticality Systems

Jasper J.A. Kuijsten
Electronic Systems, Eindhoven University of Technology
E-mail: j.j.a.kuijsten@student.tue.nl

***Abstract*—Reconfiguration capabilities for Multiple-Processor-System-on-Chip (MPSoC) platforms are increasingly common, offering increased flexibility and improved performance. However, reconfiguration is problematic for mixed time-criticality systems which may run real-time applications and whose requirements have to be verified. Feasible verification, not exploding in complexity as the number of applications increases, depends on two platform properties: predictability for bounded performance and composability for isolation of applications. For verification of reconfigurable mixed time-criticality platforms, these properties have to be maintained during reconfiguration.**

This paper proposes a solution to the verification problem for reconfigurable mixed time-criticality systems in the context of SDRAM. First, a predictable SDRAM controller is made composable by means of composable arbitration and an extension to the existing predictability method. Second, a reconfiguration method was developed to allow for reconfiguration of the SDRAM controller without violating predictable and composable properties. The solutions were implemented for the CompSoC MPSoC, both as SystemC simulation model and hardware implementation on FPGA. The work presented in this paper allows CompSoC to start and stop applications at run-time and reconfigure the SDRAM controller as required. Moreover, predictability and the established composability are preserved, such that verification complexity does not increase. This research proposes reconfiguration and composability concepts applicable to mixed time-criticality systems in general and increases flexibility for CompSoC.

***Index Terms*—SDRAM, real-time, verification, reconfiguration, latency-rate, TDM, predictability, composability**

I. Introduction

Embedded Multi-Processor-System-on-Chip (MPSoC) platforms offer high computational power while keeping power consumption at practical values [1]–[3], by running multiple applications in parallel. Applications have *requirements* on platform performance, as prerequisites for correct functionality. A subset of those applications may have *real-time requirements*, such as a minimum throughput or maximum response latencies. *Use-cases* are unique sets of concurrently running *real-time applications* and *non-real-time* applications. MPSoCs that host use-cases with mixed real-time requirements, are *mixed time-criticality systems*.

Configuring parameters of mixed time-criticality systems such that all application requirements are satisfied is challenging. Design-time *verification* assures all requirements are met, usually by system-level simulations. However, MPSoCs employ *resource sharing* between applications to reduce cost and power. Applications within a use-case may have to compete for shared resource access. Their behavior becomes inter-dependent due to *interference*, possibly impacting performance. Verification of real-time requirements is only valid if interference is considered by simulating use-cases instead of applications. This becomes unfeasible when the number of considered applications increases and both the number of use-cases and simulation complexity explode [4], [5].

Predictable and *composable* systems offer solutions for the verification feasibility problem. Predictable systems bound interference and provide bounds for performance. Applications for which a model is available can be verified using formal performance analysis frameworks [6], [7]. Other applications can only be verified by simulation, feasable only with composable systems that *isolate* applications from each other, removing application-to-application interference. This allows individual application verification instead of verification of use-cases.

Modern MPSoCs are host to multiple use-cases, all having unique requirements. Upon a *use-case switch*, *reconfiguration* may take place to change the configuration of the system, such that it is able to satisfy current requirements. Reconfiguration is important for MPSoCs because it increases flexibility. Figure 1 shows an example scenario with multiple use-cases and a platform with enough resources to satisfy requirements for at most two applications concurrently. A single use-case A,B,C cannot be allocated. However, application A and C are mutually exclusive and reconfiguration support allows all applications to be allocated over use-cases A,B and B,C.

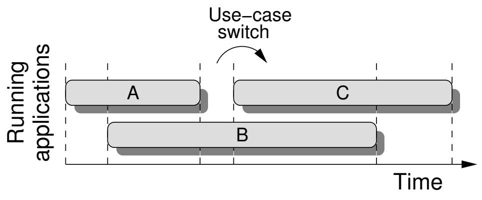


Figure 1. Example use-case switch

Reconfiguration for mixed time-criticality systems further complicates design-time verification of real-time requirements. *Persistent applications* [8] are applications with real-time requirements, which are in multiple use-cases and run during use-case switches. Verification thus has to consider conditions before, after and during use-case switches. Verification by formal performance analysis remains possible if platforms are predictable during use-case switches. Verification by simulation of individual applications is feasible if platforms are composable even during use-case switches, such that applications remain isolated and reconfiguration of one does not affect another.

This paper presents a solution for the verification problem for reconfigurable mixed time-criticality systems. The four main contributions are: 1) A method for composability on top of predictability for SDRAM. 2) A method for reconfiguration of Time-Division-Multiplexing (TDM) arbiters for mixed time-criticality systems. 3) An implementation of the proposed methods for composability and TDM arbiter reconfiguration for SDRAM for CompSoC [9], both as a SystemC simulation model and hardware platform on FPGA. 4) Extensions to an automated design-time tool flow to instantiate CompSoC with reconfiguration capabilities for a predictable and composable SDRAM controller.

Section II gives background information on SDRAM and CompSoC. Contributions begin at Section III, with a method of composability for SDRAM. Next, Section IV proposes a method of arbiter reconfiguration for real-time environments. An implementation of these concepts for CompSoC is discussed in Section V. Section VI presents results to show correct functionality of the proposed concepts and implementation. Section VII discusses related work, followed by closing conclusions in Section VIII and recommendations for future work in Section IX.

II. Background

A. CompSoC

CompSoC is a predictable and composable MPSoC. The predictability property assures that application performance is at all times bounded, despite possible interference from other applications and system processes. Independently from predictability, composability guarantees that concurrently executing applications do not affect each other, removing application-to-application interference.

Relevant components of CompSoC are processor tiles, a network-on-chip interconnect and a memory controller for SDRAM. A SystemC implementation is used for doing simulations. A hardware CompSoC platform runs on FPGA. CompSoC is instantiated by an automated tool flow, considering specified hardware and application specifications. A platform configuration for all system parameters is determined at design-time to allow design-time verification.

Use-cases are *cliques* of simultaneously running applications. Prior the work presented here, the SDRAM of CompSoC is configured based on a single *super-use-case*, the union of all use-cases. Other components are configured based on *maximum cliques* of simultaneously running applications. All cliques part of a maximum clique share a configuration, significantly reducing the number of configurations.

B. SDRAM specifics

SDRAM are natively challenging for use in real-time environments. SDRAM behavior is highly unpredictable due to their internal architecture. SDRAM hold a number of memory banks, each containing rows and columns. Banks have a row buffer, containing the currently active row, which is the only one that can be accessed. SDRAM unpredictability comes from mainly three sources [10]. 1) Row access times depend on whether a row is already active and already stored in the row buffer, or whether it first needs to be activated. 2) The data bus is bi-directional and requires a delay for switching direction, when a read follows a write or vice versa. 3) SDRAM have strict timing constraints between individual commands, of which the largest effect is caused by a *refresh* command. Refreshes are periodically executed to prevent loss of data integrity, during which the SDRAM cannot be accessed.

C. Predictable SDRAM

Timing and architectural constraints have to be considered when scheduling memory commands. Multiple scheduling mechanisms exist with different predictability and performance properties. CompSoC uses *memory patterns* [11] to achieve both acceptable performance and predictable behavior of the memory. Memory patterns are static sequences of SDRAM commands, compiled at design-time and considering the targeted SDRAM specifications. Within a pattern, all timing constraints between individual SDRAM commands are satisfied. Patterns have a certain *bank interleaving* (BI) and *burst count* (BC) number to set bank level parallelism and the number of read or write bursts within a pattern. The combinations of BIs and BCs offer different bandwidth and latency properties and are selected on application requirements [12]. A pattern always gets executed in full such that the SDRAM command sequence is only pre-emptive at the pattern level.

Five patterns are used for reads (R), writes (W), switches from read to write (RtW) and write to read (WtR) and refreshes (REF). Read and write patterns are *access patterns*, used to service read and write requests. They access the memory with a certain *access granularity* known as an *atom*. *Switching patterns* execute before either an upcoming read or write pattern if the previously executing access pattern was of the other type. Patterns access a known amount of data within a known amount of time, providing a known bandwidth and latency.

Gross bandwidth defines the maximum guaranteed bandwidth the SDRAM can offer. Applications are guaranteed a certain *net bandwidth*, a fraction of the gross bandwidth. Gross bandwidth is the bandwidth pattern sets offer during back-to-back execution of worst-case ordered patterns. Pattern sets are *read-dominant* or *write-dominant* when the respective access pattern is longer than the combination of the other access pattern and both switching patterns. For these pattern sets, worst-case bandwidth occurs when the dominating pattern is used continuously. Pattern sets are *mixed-dominant* when either combination of switching pattern and following access pattern is longest. These pattern sets offer minimum bandwidth when read and write requests interleave continuously.

Arbiters arbitrate shared resource access over *requesters*, those processes requesting resource usage. In this work, applications are single-threaded processes, such that every application is a requester. A *predictable arbiter* has known worst-case bounds on latency to receiving resource access and the rate of offered service. Effects from interference are bounded such that performance guarantees can be derived.

Arbiters characterize three resource performance aspects, 1) rate of service, 2) latency to service, 3) *over-allocation* of rate of service, the excess provided service to the requested service due to limited arbiter precision. Generally, arbiters suffer from coupling between these aspects, such that setting a specific value for one aspect will affect others. Credit-Controlled-Static-Priority (CCSP) [13] is the most advanced arbitration scheme available to CompSoC, usually used to arbitrate the Predator [14] SDRAM controller. It is able to decouple all three performance aspects by employing both requester priorities and a continuous budgeting system.

In this paper, any use of the pattern name abbreviations, unless specifically mentioned otherwise, refers to pattern lengths, i.e. the number of memory commands of a pattern. Pattern lengths also directly relate to their execution times, since the memory processes one command each clock cycle.

D. $\mathcal{LR}$-servers

SDRAM performance bounds are based on the Latency-Rate ($\mathcal{LR}$) server model [15]. They are able to abstract shared resource behavior. $\mathcal{LR}$-servers guarantee requesters a minimum *allocated rate* of service after a maximum *service latency*, shown graphically in Figure 2. The service guarantee bounds the amount of received service independently of the behavior of other requesters. The service latency and allocated rate are controlled by the arbiter. The $\mathcal{LR}$ service guarantee is only valid during *busy periods*, which are periods during which the requested service is at least as much as is allocated on average. Busy periods are shown in Figure 2 as those intervals where requested service is above the reference *busy line*.

CompSoC SDRAM is abstracted as a $\mathcal{LR}$-server and its $\mathcal{LR}$ guarantees allow for derivation of *worst-case response times* (WCRT). The SDRAM is predictable if requests always finish before their WCRT. For the first request of busy periods, worst-case delay from arrival to receiving service is bounded by the service latency. Following requests at latest receive service at the worst-case *finishing time* of the previous request. A request finishing time is bounded by taking its worst-case delay to service and adding the time required to service the request, with certain size, with the guaranteed allocated rate, known as the *completion latency*.

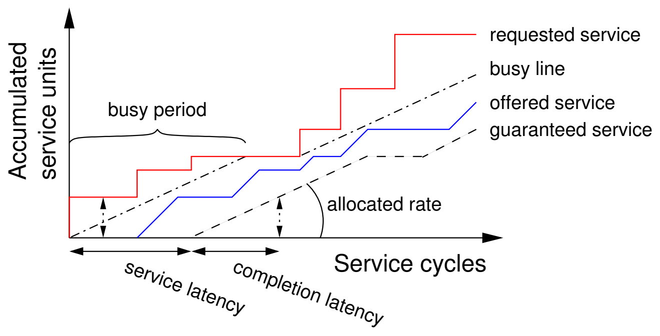


Figure 2. $\mathcal{LR}$-server with annotations

For this work the $\mathcal{LR}$-model is slightly modified to optimize performance for predictable SDRAM [16]. WCRT are determined using a completion latency valid only for Predator. Completion latencies are generally determined using the allocated rate and assuming a fully pre-emptive resource. Predator is only pre-emptive on pattern level, such that requesters are guaranteed maximum rate for a short time instead of the allocated rate for a longer time. Completion latencies for Predator are thus shorter than anticipated by a general $\mathcal{LR}$ model.

E. Composable SDRAM

Composability is achieved by artificially reducing performance of applications to their worst case. Request *service times* may be dependent on behavior of other requesters, but are bounded by the WCRT if the shared resource is predictable. Variations in application behavior due to variations in interference behavior are removed by always delaying requests to WCRT, truly isolating applications. *Delayblocks* [17], sitting in the path from SDRAM to requester, are used to delay to WCRT.

III. Composability Method

The first contribution of this paper is a novel method for composable SDRAM. This section first introduces TDM arbitration and elaborates on its composable properties. Next, composable patterns are proposed.

A. TDM arbitration introduction

TDM arbiters regulate access to a resource across multiple requesters by dividing time into slices and appointing resource access to requesters for slice durations. TDM arbiters use a *frame* consisting of *slots*, each representing a slice of the time required to service the complete frame. Slots are allocated to requesters, giving them resource access in those slots. Slot sizes are set to the service time of one atom. Slots are non-preemptive because atoms cannot be pre-empted. A *scheduling decision* is made every time a request is serviced or, if idle, when the *schedule interval* has passed. This interval is equal to the execution time of the shortest access pattern. Every scheduling decision the *index* is updated, selecting the next slot for scheduling. A new *frame iteration* is started if the index moves from the last to the first slot.

A TDM configuration consists of 1) a frame size parameter which determines the number of slots in the frame, and 2) an allocation of slots to requesters. Many allocation strategies exist, but these are not the focus of this work. A *greedy allocation* strategy is used, which allocates continuous blocks of slots per application.

B. TDM composability

Conceptually, TDM arbiters are composable by design. Requesters always receive service for their allocated duration, independently of who is serviced for the remainder of the frame iteration. However, its composability property depends on what the arbiter decides to do when a scheduled requester does not have any pending requests. A *work-conserving* arbiter schedules another requester with pending requests. A *non-work-conserving* arbiter does not schedule any and the resource falls idle. For a work-conserving TDM arbiter, the instant when requests for a particular requester receive service is dependent on the number of pending requests of other requesters. This application-to-application dependence may cause requests to be served sooner than expected, invalidating composability. Non-work-conserving arbiters do not have this dependence and are conceptually composable.

Using a non-work-conserving TDM arbiter to arbitrate specifically a SDRAM, does not give composable behavior. The reason is varying slot sizes. Slot sizes vary because service times for atoms do, and they cannot be preempted. Service times are variable for three reasons: 1) Read and write requests may have a different service time, depending on the corresponding patterns lengths. 2) Switching patterns are required to execute before the access pattern when a read request follows a write request or vice versa. 3) A scheduled requester may experience delayed execution due to a refresh. Occurrence of the first and second reason is shown in Figure 3. The figure shows execution of four consecutive requests, all with different service times. Only the first two reasons result in application-to-application interference. Firstly, different access pattern lengths cause time instants of upcoming scheduling decisions to vary depending on the current request type, and consequently affecting upcoming request service times. Secondly, the service time of the current request is dependent on the type of the previously serviced request, via a potentially required switching pattern. The third reason, interference from refreshes, can be ignored because it is not induced by applications.

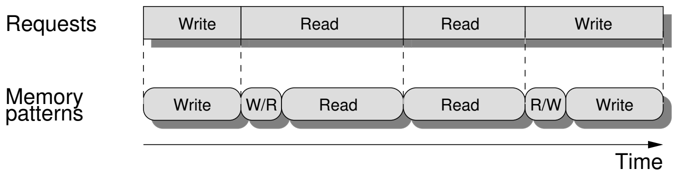


Figure 3. Predictable pattern execution

C. Composable patterns

Composable patterns are proposed to solve slot size dependence on application behavior. Composable patterns are memory patterns with two properties that overcome non-composability of the original *predictable patterns*. First, all access patterns are equally long and second, switching patterns are always executed by incorporating them into the access patterns. The same execution trace as shown in Figure 3 is shown in Figure 4, using composable patterns. Service times are equal and consequently slot sizes are too.

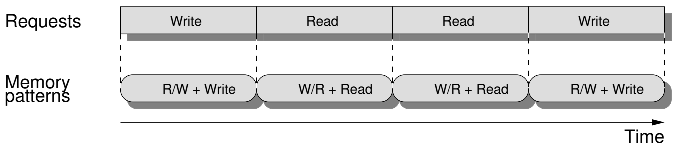


Figure 4. Composable pattern execution

Composable patterns are generated by adapting predictable patterns, still considering the original timing constraints. Two different methods are used, depending on the pattern set dominance property. For mixed-dominant patterns: First, switching patterns are inserted in full in front of their corresponding access pattern, i.e. the read-to-write switching pattern is inserted before the write pattern, and the write-to-read switching pattern is added before the read pattern. Second, after appending the switching patterns the access patterns may have different lengths. SDRAM idle commands (NOP) are appended to the shortest access pattern to make them equally long. For read-dominant or write-dominant pattern sets, both switching patterns are appended to only the shortest access pattern. Again, if after adding switching patterns the access patterns are not equally long, the shortest pattern is appended with NOPs. Equation (1) describes the relation between access patterns (AP) lengths of a composable pattern set P_c and a predictable pattern set P_p, for both mixed-dominant *(md)* and read-dominant or write-dominant *(rdwd)* pattern sets.

$$AP_c = \begin{cases} max(R_p, W_p) & \textit{(rdwd)} \\ max(WtR_p + R_p, RtW_p + W_p) & \textit{(md)} \end{cases} \quad (1)$$

An example result of converting predictable patterns to composable patterns is shown in Table I. Besides the newly sized access patterns and removed switching patterns, also the refresh pattern and introduced idle pattern (I) are shown. Refreshes do not affect composability, hence the refresh pattern can be differently sized. In addition to the previously discussed steps from predictable to composable patterns, an idle pattern is introduced. Idle periods can be invoked by both letting the controller fall idle by executing no pattern, or by executing explicit idle patterns. If not using idle patterns, the scheduling interval enforces a duration of one slot for idle periods. For both methods, the SDRAM controller executes NOP commands and initially timing behavior is identical. However, whether an actual pattern is sent to, buffered in, and executed by the SDRAM controller, or whether the controller just executes a right number of NOPs without an explicit pattern, influences its buffer usage. Without explicit idle patterns, buffer usage differs between situations where requests are scheduled or not, as patterns are used for the first but not the second case. For situations where differences in buffer usage influence timing behavior, application-to-application interference could occur. Idle patterns are hence introduced for composable pattern sets, eliminating SDRAM controller buffer usage dependence on application behavior and the non-composability that follows.

Table I
PATTERN LENGTHS (MIXED-DOMINANT SET)

	P_p	P_c
R	31	38
W	35	38
WtR	5	0
RtW	3	0
REF	44	44
I	-	38

D. Performance analysis and evaluation

The second part of this work, starting with Section IV, focuses on reconfiguration for SDRAM with predictable and composable properties. The proposed method of composability is particularly appealing for reconfiguration, compared to the method using CCSP and delaying to WCRT. Reconfiguration for TDM arbitration is per application, while reconfiguration for CCSP is only possible per use-case. Reconfiguration per use-case is non-composable because it affects all applications within the use-case by definition. Reconfiguration of the SDRAM controller is thus preferably explored for TDM arbitration. However, before spending effort on a reconfiguration solution for the composability method using TDM arbitration and composable patterns, its performance is investigated to determine its usability in practice.

Composable access patterns are on average at least as long, but often longer than the original predictable patterns. The same amount of data is accessed using longer patterns, implying composable patterns have reduced efficiency and offer reduced gross bandwidth. Use-cases with high bandwidth requirements may not be allocated successfully using composable patterns, but would have been with predictable patterns. Equation (2) analytically expresses the relation between gross bandwidth for predictable and composable patterns, using efficiency factor e^{pc}. Gross bandwidth is not affected for read-dominant or write-dominant pattern sets, because the dominant pattern has not changed.

$$e^{\mathrm{pc}} = \begin{cases} 1 & \textit{(rdwd)} \\ \frac{R_p+W_p+WtR_p+RtW_p}{R_c+W_c} & \textit{(md)} \end{cases} \tag{2}$$

Composable patterns also have an effect on request service times. Latencies may increase due to two aspects. 1) Access patterns may incorporate switching patterns and additional appended NOPs. Switching patterns are hence always executed, even though they are not always needed. 2) Requests may experience an increased service latency, because previous requests take longer to finish. It is not possible to analytically determine average request latency increases for arbitrary situations. This depends on the number of required switching patterns, which is arbiter, application and use-case specific. A lower bound can be determined by only taking pattern lengths in account, ignoring possibly increased service latencies, and considering a worst-case ordered request stream. A lower bound on average latency increase is then given by $1/e^{\mathrm{pc}}$.

Three experiments are performed to determine: 1) Gross bandwidth reduction when using composable patterns, for a diverse range of memories and pattern sets. 2) Request latency increase due to composable patterns for real life applications. 3) Request latency performance of TDM arbitration and composable patterns compared to CCSP arbitration and delaying to the WCRT for real life applications. These three experiments together present sufficient information to decide on usability of the composability solution in practice.

Results are discussed in Section VI as part of the final results. For now it is sufficient to know that, respectively for each experiment: 1) Gross bandwidth reduces, but is unlikely to affect use-case allocation chances. 2) Using composable patterns increases request latencies on average by 45% over using predictable patterns. 3) Request latencies on average increase with 42% when using TDM arbitration and composable patterns, compared to using CCSP arbitration, predictable patterns and a delayblock to delay to WCRT. Actual performance differences are highly dependent on platform parameters.

It is concluded that the composability method of TDM arbitration and composable patterns is usable in real life situations. Section IV proposes a reconfiguration method for it.

IV. Reconfigurable TDM Arbitration

This paper presents work on a reconfigurable mixed time-criticality system, such that it can be host to multiple use-cases. Each use-case has a unique set of requirements and upon a use-case switch, the system must be able to stop and start applications and reconfigure affected resources such that they satisfy the updated requirements. Such a reconfiguration is a system-wide process. Events take place on three levels: 1) Reconfiguration on system-level, i.e. use-case switching, initiated by starting and stopping of applications. 2) Reconfiguration of applications, i.e. changes to the configuration of applications during their lifetime. 3) Arbiter reconfiguration, the process of making changes to the TDM frame at run-time. Slots are reconfigured to match the configuration of starting use-cases, as determined at design-time.

Start and stop reconfiguration events are not challenging themselves. Applications may require $\mathcal{LR}$ guarantees and composability for design-time verification, but only during their *lifetime*, the period in which they are *active*. By definition, start and stop events take place outside application lifetime. Starting and stopping applications, on arbiter level adding and removal of their allocated slots, can be done arbitrarily without increasing verification efforts beyond feasibility. The challenge for use-case switching is how other applications, besides the one inducing the use-case switch, experience the reconfiguration. Different types of applications are distinguished, based on their real-time requirements during use-case switches. All applications receive predictable and composable service during regular execution. *Persistent composable applications* are real-time applications and active in multiple use-cases, requiring composability for design-time verification by simulation. The challenge is to process reconfigurations for use-case switching without affecting them. Composability is lost otherwise. *Persistent predictable applications* are real-time applications which depend on $\mathcal{LR}$ guarantees for design-time verification by formal performance analysis. These applications can be affected by reconfiguration events, as long as their $\mathcal{LR}$ guarantees are not invalidated. *Non-persistent applications* are active in only one use-case and never experience arbiter reconfiguration.

This section describes a method of arbiter reconfiguration to reconfigure applications while keeping $\mathcal{LR}$ guarantees valid and preserving composability, allowing for system-level reconfigurations to start and stop applications while keeping verification efforts feasible.

A. Maintaining composability

Use-case switching with persistent composable applications has two prerequisites. 1) The reconfiguration process needs to be independent of the scheduling process, such that processing reconfiguration events does not delay scheduling decisions. This is easily satisfied when scheduling and reconfiguration are handled by separate submodules of the arbiter. 2) Persistent composable applications are never reconfigured during their lifetime. They must have identical slot allocations over all their use-cases. This complicates the allocation process due to coupling in configurations between use-cases. The actual effects depend on the used allocation strategy. For this work, using greedy allocation, fragmentation of the TDM frame becomes an issue.

Algorithm 1 determines slot allocations. Persistent composable applications are given priority and receive allocation over all their use-cases. Next, all persistent predictable applications are allocated to their use-cases. Persistent predictable applications may have different allocations over their use-cases, due to already allocated persistent composable applications, and may require reconfiguration during their lifetime. This has to be done without invalidating their $\mathcal{LR}$ guarantees. Last, non-persistent applications receive their allocation.

Algorithm 1 Allocation algorithm

```
Input: persistent composable applications $\vec{A}_c$, other applications $\vec{A}_o$,
  slot requirements $\vec{R}$, use-cases $\vec{U}$,
Output: TDMA configurations $\vec{C}$
  for all a in $\vec{A}_c$ do
    $location \leftarrow 0$
    for all u in $\vec{U}[a]$ do
      repeat
        $\vec{S}[u] \leftarrow findSpace(location, \vec{R}[a])$
        $location \leftarrow increment(location)$
      until $\vec{S}[u]$ or reached end of frame
    end for
    if $\vec{S} = \vec{1}$ then
      $\vec{C}[\vec{U}[a]] \leftarrow setSlots(a, location, \vec{R}[a])$
    end if
  end for
  for all a in $\vec{A}_o$ do
    $location \leftarrow 0$
    for all u in $\vec{U}[a]$ do
      repeat
        $S \leftarrow findSpace(location, \vec{R}[a])$
        $location \leftarrow increment(location)$
      until $S$ or reached end of frame
      if S then
        $\vec{C}[u] \leftarrow setSlots(a, location, \vec{R}[a])$
      end if
    end for
  end for
```

B. Predictable arbiter reconfiguration

Persistent predictable applications require reconfiguration if their slot allocation differs between the two use-cases. If so, allocated slots are *moved* by adding newly allocated slots and removing old ones. Behavior for both configurations has been verified at design-time. However, their behavior is temporarily undetermined during the use-case switch when slots are added and removed. This is only allowed if behavior stays predictable, i.e. still follows the $\mathcal{LR}$ guarantees. Two prerequisites have to be satisfied: 1) An application has to receive service for at least the allocated number of slots during each frame iteration. Failure to satisfy this temporarily decreases the allocated rate and invalidates completion latencies. 2) *Arbiter latencies*, the maximum interval between two slots of an allocation, cannot exceed the value determined at design-time for regular execution. This would invalidate the service latency bound.

Figure 5 shows two reconfiguration scenarios. An application owns one slot in a frame of four. The index moves right as time passes, every fourth update selecting the first slot again. Arrow lengths indicate observed arbiter latency, expressed in slots, being three during regular execution. A reconfiguration event that moves the allocated slot takes places at time t_r when the index is at the appointed location. Scenario $1a$ and $2a$ add and remove slots directly and simultaneously at t_r. For scenario 1, the allocated slot moves to the left. For $1a$, due to the unfortunate reconfiguration time instant, the allocated slot moves past the index. A full frame iteration without service occurs, both increasing arbiter latency beyond the regular value and temporarily reducing the allocated rate to zero. Both service latency and completion latency are invalidated. For 2, the allocated slot moves to the right. For $2a$, arbiter latency again increases beyond its regular value, invalidating the service latency.

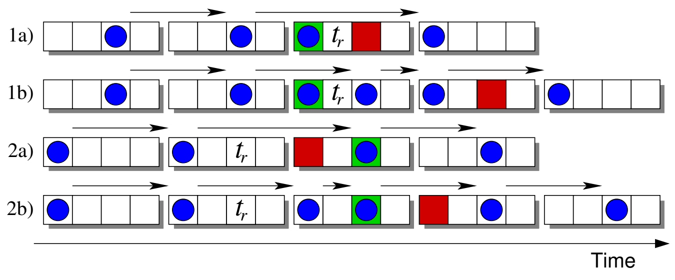


Figure 5. Arbiter latencies during reconfiguration. Unpredictable: $1a$ and $2a$. Predictable: $1b$ and $2b$

It is possible to reconfigure the TDM arbiter without invalidating guarantees for the reconfigured application. The solution consists of three aspects. 1) The reconfiguration event for moving slots consists of two distinct reconfiguration actions to add and remove slots. Of these two actions, adding is always processed before removing. 2) Processing a reconfiguration event is never interrupted by events related to other applications. 3) Removal of slots is only allowed at the beginning of a frame, i.e. at the start of a new frame iteration. These rules assure that adding and removal of slots take place in separate frame iterations and slots do not get added to another requester before being properly removed. Reconfiguration using this method is shown in Figure 5, scenarios $1b$ and $2b$. Arbiter latencies do not increase and frame iterations with too few allocated slots do not occur. For a single frame iteration, moved applications temporarily have more slots allocated to them than during regular execution. Latencies temporarily decrease and offered service increases, temporarily improving application performance. Both service latency and completion latency are valid during reconfiguration and $\mathcal{LR}$ guarantees are obeyed.

It is worth noting that it is possible to offer composability to persistent predictable applications, which may be reconfigured. $\mathcal{LR}$ guarantees are valid during reconfiguration and delayblocks can delay to a valid WCRT. The proposed arbiter reconfiguration method is particularly appealing, because WCRT do not increase. Alternatively to the proposed method, one could also determine WCRT during reconfiguration and apply these to regular execution. This gives more pessimistic worst-case bounds.

V. Implementation

The concepts discussed in Section III and Section IV are only part of a larger effort for composability of the SDRAM and reconfiguration support at system level. This section describes the implementation and integration of composable patterns and a reconfigurable TDM arbiter with supporting functionality into CompSoC. A composable SDRAM controller and use-case switching capabilities require support on multiple levels. Top-down, there are: 1) At design-time, composable pattern generation and support for the tool flow to consider multiple use-cases to determine run-time configurations per use-case 2) At run-time, support for reconfiguration in the arbiter driver software. 3) In hardware, the arbiter module itself and an infrastructure to deliver reconfiguration events at the arbiter. The following subsections elaborate on each of them.

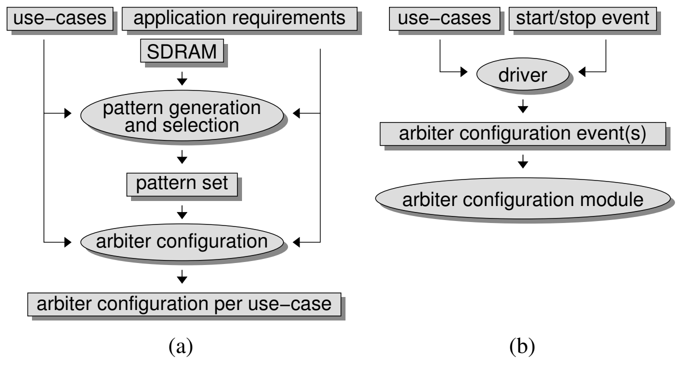

use–cases
application requirements
SDRAM
pattern generation and selection
pattern set
arbiter configuration
arbiter configuration per use–case
(a)
use–cases
start/stop event
driver
arbiter configuration event(s)
arbiter configuration module
(b)

Figure 6. Design-time (a) and run-time (b) flows

A. Design-time tool flow

The CompSoC tool flow determines an arbiter configuration at design-time by the flow shown in Figure 6a. The flow is identical for CompSoC with or without support for multiple use-cases, except for the use-case data. The tool flow previously used a super-use-case for configuration of the SDRAM controller. With reconfiguration support, the flow uses maximum cliques of simultaneously running applications. The tool flow output is now multiple arbiter configurations instead of just one, all using a common pattern set. In the pattern generation and selection process, support is added in two areas. First, there is the pattern generation, to which conversion to composable patterns is added. Second, the pattern selection process is to consider all use-cases instead of the previously used super-use-case. With the ability to switch use-cases, all use-cases are individually considered to determine maximum *slack bandwidth*. The selected pattern set offers most slack summed over all use-cases. Applications which are in many use-cases, influence final pattern selection more than those who are in few.

Arbiter configuration takes place as discussed in Section IV and shown in Algorithm 1. At this stage of the tool flow, application requirements are expressed in number of slots. The selected pattern set is used to convert bandwidth requirements to slot requirements [18].

B. Run-time software

Support for multiple use-cases requires support in CompSoC software and hardware, as shown in Figure 6b. Upon a system-level application start or stop, the involved processes are, 1) checking the need for reconfiguration, 2) creating configuration events 3) sorting reconfiguration events.

1) Checking the need for reconfiguration: Not all use-case switches require reconfiguration for other applications besides the one starting or stopping. Use-case switches within the same maximum clique share configurations and reconfiguration is thus not necessary. If the upcoming use-case is part of a different maximum clique reconfiguration may be required and applications that are in both the current and next use-case are marked.

2) Creating reconfiguration events: The driver compiles a list of reconfiguration events. Such events contain information on which slots are to be added or removed for a requester. For marked applications, allocations on both sides of the use-case switch are compared and a reconfiguration event is created if they differ, to move their slots, consisting of a separate adding and removal action. Overlap between allocations of the current and next configuration is handled when creating the removal action. Recently added slots, by the adding action of the same reconfiguration event, should not be removed immediately after by the following removal action. Without this precaution, overlapping slots between the two configurations are missing from the new allocation and the resulting reduced allocated rate leads to violation of the $\mathcal{LR}$ guarantees. Reconfiguration events for starting or stopping applications are straightforward.

3) Sorting reconfiguration events: The reconfiguration event list is sorted to determine in which sequence they should be performed. The reconfiguration order of active applications matters, because the current and next allocations of different applications might interfere, i.e. the allocation for an application in the next use-case that is reconfigured first, uses slots still allocated to another application in the current use-case. Occurrence of such a situation is the switching condition for a bubble-sort algorithm to order all reconfiguration events such that allocations do not interfere. The worst-case computational complexity $O(n^2)$ is not an issue for CompSoC for any practical number of applications. Circular dependencies are conceptually possible if two applications switch slots, but do not occur with the used allocation strategy.

C. Run-time hardware

Hardware related to reconfiguration support is limited to the infrastructure used to deliver reconfiguration events at the arbiter, and the arbiter itself. The infrastructure was already in CompSoC prior to this work. The TDM arbiter is implemented as discussed in Section IV. Scheduling and reconfiguration processes are implemented as distinct submodules, such that they work in parallel. Furthermore, a *shadow frame* is used. This frame is a second TDM frame, not used for making scheduling decisions, but only for reconfiguration purposes. Shadow frame contents are copied into the regular frame at transitions between frame iterations, making reconfigurations effective only at this time. Removal of slots is only allowed during a transition between frame iterations and adding slots is only allowed directly if no removal event is waiting. Adding events that can be processed directly are processed on the regular frame. Removal and queued adding of slots are performed on the shadow frame.

VI. Experimental Results

A. Experimental setups

Experiments are performed both on the SystemC simulation environment and hardware platform on FPGA. A Xilinx ML605 [19] is used, equipped with a Micron MT4JSF6464HY 512MB DDR3 SDRAM SODIMM. A model of this SDRAM is available and used for the tool flow and simulations.

1) SystemC simulation model: A SystemC implementation of Predator is used to access the SDRAM. Applications are implemented by either a *traffic generator* or *trace player*.

Traffic generators generate requests as specified by the application requirements, i.e. latency and read and write bandwidth. Variations in request frequency around the mean bandwidth are controllable. Alternatively to synthetically generated traffic, *traces* can be used. Trace players execute cycle-precise request level traces of real life applications. Traces are generated by running applications on a *SimpleScalar* [20] ARM simulator.

2) FPGA platform: The CompSoC hardware implementation uses Xilinx µBlaze processors to execute applications. For this work, no operating system is used such that they run only a single application at a time. The applications used here are synthetic applications, doing read or write requests with a certain bandwidth as specified. Like the traffic generators, burstiness in traffic is specifiable. Raptor, a VHDL implementation of the Predator SDRAM controller is used.

B. Composability method performance analysis

1) Gross bandwidth reduction: As discussed in Section III, the conversion from predictable to composable patterns may cause a gross bandwidth reduction. An experiment is performed to determine exact losses for a diverse range of SDRAM and pattern sets. Gross bandwidth for a SDRAM is known after pattern set generation, hence each time only the tool flow is run and no simulations or executions on FPGA are required.

Figure 7 shows gross bandwidth decrease with composable patterns for every supported BI and BC pattern set configuration for the Micron MT4JSF6464HY. Results are highly dependent on pattern configurations, ranging from 0% reduction for every pattern with BI1, to 7.9% for the pattern set with BI4 and BC2. It is out of scope of this work to elaborately discuss reasons for particular performance of each pattern configuration, as it mainly depends on the pattern generation algorithm. However, two observations are listed next. 1) e^{pc} is mostly dependent on switching pattern sizes, because most pattern sets are mixed-dominant with equally long access patterns. 2) Pattern sets with BI1 do not suffer reduced gross bandwidth. For these, predictable and composable pattern sets are identical.

Table II shows summarized gross bandwidth reductions for twelve SDRAM, divided over multiple SDRAM types. For each memory type DDR2, DDR3, LPDDR and LPDDR2, the average and maximum gross bandwidth reduction across pattern configurations are given. Results show that for specific pattern and memory configurations, gross bandwidth reduction can go up to 12%. For all SDRAM types on average over all pattern sets less than 1.6% of gross bandwidth is lost. For the majority of combinations of pattern sets, targeted SDRAM and use-case requirements, allocation chances are hardly affected.

Table II
GROSS BANDWIDTH REDUCTION SUMMARY

[%]	Average Gross Bandwidth Reduction	Maximum Gross Bandwidth Reduction
DDR2	1.25	9.53
DDR3	1.58	12.00
LPDDR	0.77	9.53
LPDDR2	0.26	5.00

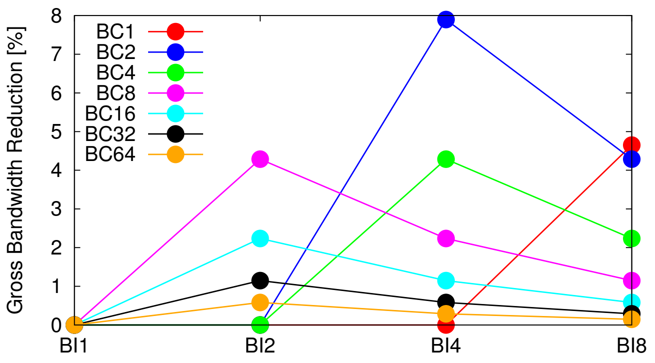


Figure 7. Gross bandwidth reduction per pattern set for MT4JSF6464HY

2) Composable pattern latency performance: Section III states that composable patterns have increased latency over predictable patterns, for which a lower bound is given. Simulation experiments are performed to determine request latency increases when using composable patterns, for real life applications using traces. Mediabench [21] applications *g721decode* and *h263decode* are selected, both having a representable mixture of read and write requests. They are executed both with predictable and composable patterns, for a number of pattern set configurations. The selected pattern sets have practical access granularities. All other system parameters are kept constant. A TDM arbiter is used with frame size set to 20. Applications receive 30 MB/s of guaranteed bandwidth in both read and write directions. Latency requirements are set such that they are satisfied by the selected pattern sets.

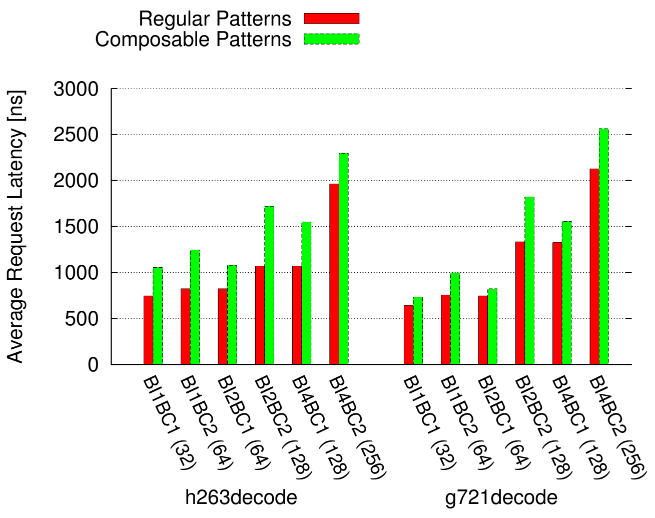


Figure 8. Latency performance for predictable and composable patterns

Results for the selected pattern sets, with access granularities annotated between brackets, are presented in Figure 8. The figure shows that composable patterns increase the request latencies by an amount significantly greater than the respective lower bounds. All selected pattern sets have a lower bounded latency increase of 0% except for the pattern set with BI4 and BC2, for which the lower bound guarantees a minimum increase of 8.6%. For the shown applications, average latencies increase with a range of 10% up to 61%, and on average across all selected pattern sets with 45%. Similar results hold for more applications in the mediabench benchmark set, whose results are omitted here.

3) Average-case performance composability methods: Composable SDRAM by means of composable patterns and TDM arbitration is an alternative method to the approach of using delayblocks. Performance of both methods is compared experimentally by simulation. Traces of the g721decode and h263decode applications are executed in two situations: A) Using CCSP arbitration, predictable patterns and an active delayblock, delaying requests to their WCRT. Only the highest priority application is considered. B) Using TDM arbitration, composable patterns and an inactive delayblock. The frame size is set to 20. For both A and B, applications receive 30 MB/s of both guaranteed read and write bandwidth.

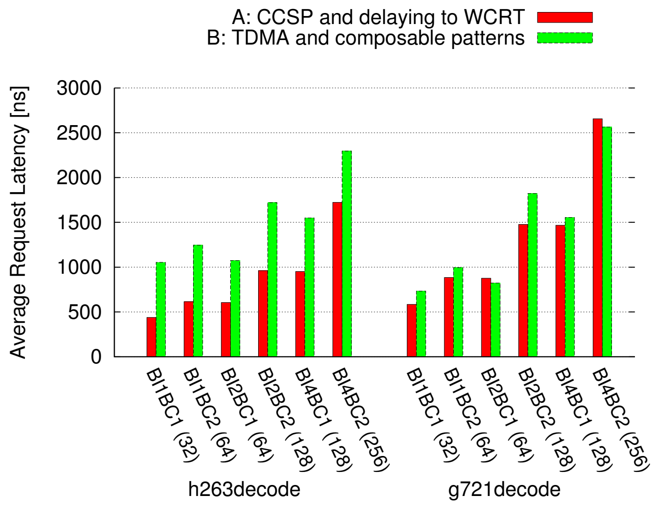


Figure 9. Latency performance for both composability methods

Figure 9 shows the average-case request latencies for A and B. For B, latencies increase for all but two pattern sets, with 46% up to 140%. For two pattern sets method B performs better than A with reductions of 3% and 6%. Averaged across all pattern sets, latencies for B increase with 42%. This indicates CCSP and delaying to WCRT as the best performing composability method. However, a fair comparison considers more than just the shown latencies. Some points for a fair comparison are: 1) TDM arbitration suffers from coupling while CCSP arbitration does not. This is not very obvious in the results presented here, because the system parameters are chosen specifically to avoid extreme settings, such as a very small or large frame, or very low or high bandwidth settings. CCSP performance is far superior for these situations. 2) Shown behavior is actual-case which for A is equal to worst-case. Worst-case behavior for B is far worse than worst-case behavior for A, possibly affecting satisfaction of application latency requirements. 3) Not shown in the figure, A offers more gross bandwidth using predictable patterns. 4) B has increased over-allocation up to 5% per applications with frame size 20, which may be cause for unsuccessful allocation for larger use-cases. 5) Performance for method A considers the highest priority application only. Latencies are larger for applications with lower priorities. This is particularly noticeable for larger use-cases. TDM arbitration uses no priorities and all applications receive equal service.

Generally, the method of composability using CCSP arbitration and delaying to WCRT offers superior performance over TDM arbitration and composable patterns. Many aspects influence performance of TDM arbitration and an exact comparison is difficult. However, it is concluded the composability method of TDM arbitration and composable patterns is usable in practice.

C. Temporal application behavior

1) Predictable reconfiguration: A simulation experiment is set up with use-cases as shown in Figure 10. Applications $A-G$ are traffic generators (TG) with bandwidth requirements as given in Table III. They are divided over use-cases $U1$, $U2$ and $U3$. Applications A and D are specified persistent composable and applications F and G specified persistent predictable. Each is active during at least one of the use-case switches $T1$ at 30 µs and $T2$ at 68 µs. Total execution time is 100 µs. Side by side applications share traffic generators, which is possible because they are mutually exclusive. All applications have a maximum latency requirement of 2000 ns.

Table III
BANDWIDTH REQUIREMENTS FOR EXPERIMENT IN FIGURE 10

Application	A	B	C	D	E	F	G
Requested [MB/s]	320	240	200	320	800	600	400

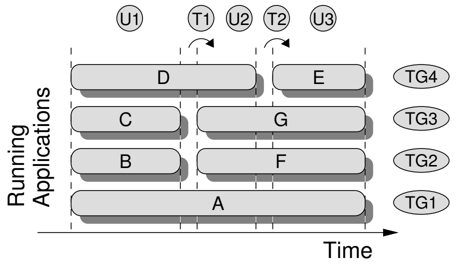
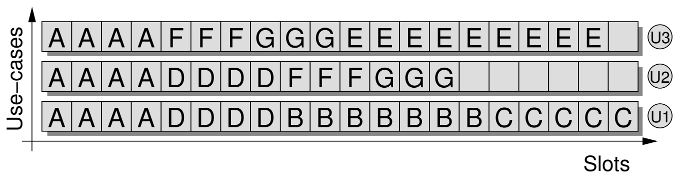


Figure 10. Use-cases for reconfiguration experiment

A pattern set with BI4 and BC1 is selected, offering a peak net bandwidth of 1862 MB/s. A super-use-case containing all applications requires 2880 MB/s and the experiment cannot be allocated successfully. Figure 11 shows the arbiter configuration for this experiment for CompSoC with use-case switching capabilities. The allocation is determined by Algorithm 1 with a set frame size of 20 slots. Persistent composable applications A and D have identical allocations over their use-cases, while persistent predictable applications F and G do not. Were they to have identical allocations between $U2$ and $U3$, application E would not be successfully allocated due to fragmentation.

Figure 12 shows finishing latencies for requests originating from $TG2$, which first runs application B and later F, with their WCRT as determined by their $\mathcal{LR}$ guarantees. The test is run twice, once without and once with the proposed method of predictable arbiter reconfiguration. Two reconfiguration events occur during the total run-time. The first event is visible at $T1$ at 30 µs, when application B is stopped and application F is



Use−cases
A A A A F F F G G G E E E E E E E E E
A A A A D D D D F F F G G G
A A A A D D D D B B B B B B B C C C C C
U3
U2
U1
Slots

Figure 11. Arbiter configuration for experiment in Figure 10

started. The second event at $T2$ at 68 µs stops application D and starts application E, inducing a use-case switch from $U2$ to $U3$. Applications F and G are reconfigured, moving their allocations to accommodate application E. Figure 11 shows that both their allocations are moved left. The time instant for $T2$ is chosen to cause the situation shown in Figure 5 ($1a$) for application F.

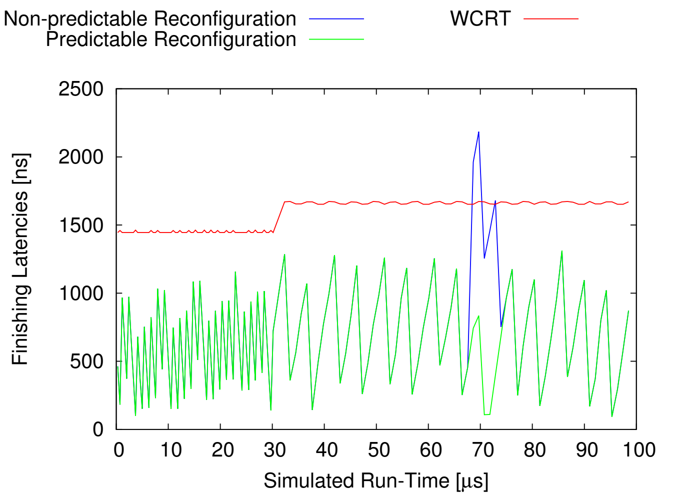


Figure 12. Reconfiguration at $T2$ for application F

Both runs show identical behavior during regular execution and when stopping and starting applications. Only when application F is reconfigured the behavior for F differs between runs. Arbitrarily adding and removal slots leads to invalidation of the $\mathcal{LR}$ guarantees, as some requests finish above their allowed WCRT. The proposed arbiter reconfiguration method maintains the $\mathcal{LR}$ guarantees.

2) $\mathcal{LR}$-server validation: The experiment presented here investigates $\mathcal{LR}$ guarantees for the hardware instance of CompSoC on FPGA, using composable patterns and TDM arbitration. The automated tool flow is used to instantiate and configure a platform with two µBlazes. Each µBlaze runs a single application doing 64 Byte SDRAM requests. *Application 1* (A1) only does read requests and *Application 2* (A2) only does write requests to ensure interference. A2 requests a bandwidth of 80 MB/s, but periods of high and low request frequencies occur. During high request frequency intervals up to 180 MB/s is requested. A2 has a maximum allowed request latency of 1650 ns. A1 requests 300 MB/s with maximum allowed latencies of 2500 ns. Frame size is set to eight slots. With these parameters, the tool flow selects a composable pattern set with BI1 and BC2. A2 receives a single slot, guaranteeing 90 MB/s of bandwidth. A1 has four slots in the TDM frame.

Figure 13 illustrates accumulated requested service, provided service, guaranteed service and busy lines for A2. Only a small part of the total execution is shown to keep $\mathcal{LR}$ properties visible. The accumulated requested service increases with one atom for every request arrival and periods of high request frequencies initiate busy periods. The provided service is always above the guaranteed service of 90 MB/s, but the distance between them changes per busy period. The actual service latencies differ, because of the differing arbiter states at the start each busy period. The TDM index location at the time of request arrival affects schedule latencies. However, service latency is correctly bounded and service is provided with at least the allocated bandwidth, as according to $\mathcal{LR}$ guarantees. The discrete nature of the provided service is explained by the pattern level pre-emption of the SDRAM controller. A2 requirements are met. The SDRAM, with composable patterns and TDM arbitration, is correctly abstracted by a $\mathcal{LR}$-server and the $\mathcal{LR}$ guarantees derived at design-time are shown to be valid at run-time on hardware.

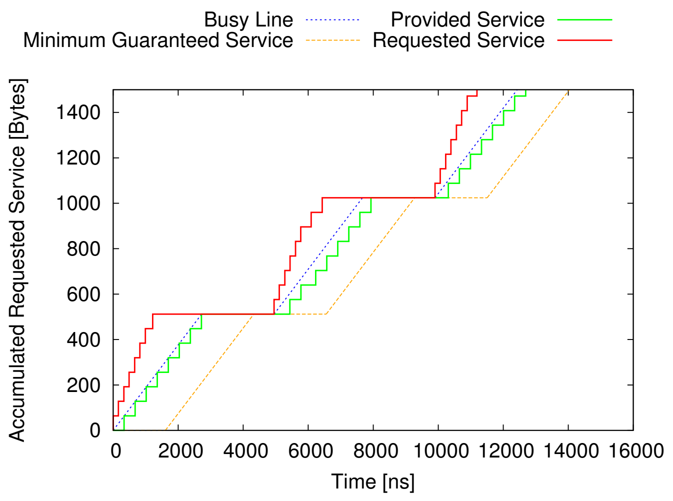


Figure 13. $\mathcal{LR}$-server behavior for SDRAM

3) Composability: The previous FPGA experiment is extended to show composable behavior for A1. Persistent composable application A1 performs hundred read requests at 360 MB/s and is run three times, each time under different circumstances. A) A2 is inactive and does not attempt to interfere. It has no allocated slots in the TDM frame. B) A2 is active, does write requests and has a bandwidth allocation of 270 MB/s, four slots in the TDM frame. C) A2 is active and does writes. Its initial bandwidth allocation of 90 MB/s is reconfigured to 180 MB/s after 35 µs. Its allocation changes from one to two slots. Furthermore for all cases, frame size is set to eight and latency requirements are chosen such that again a pattern set with BI1 and BC2 is selected. The experiment is performed twice, once with predictable patterns and once with composable patterns.

Figure 14 shows temporal behavior of the applications when using predictable patterns. The green line shows behavior of A1 for A, when it is the only active application and no attempts to interfere are made. This is considered its reference behavior. Blue and red lines indicate behavior for A2 for B and C. For these cases, A1 behavior is shown respectively as exes and circles. Composability is shown if A1 has identical behavior for A, B and C, i.e. if exes and circles are positioned on top of each other and on the green line. This is not the case. Interference from A2 causes most requests of A1 to finish later, and some sooner, than without interference. Run-times for A, B, and C are different due to the specific interference behavior.

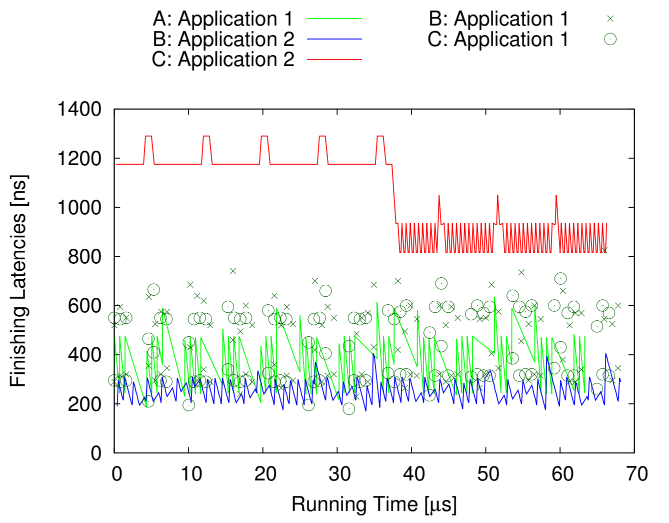


Figure 14. Non-composable behavior using predictable patterns on FPGA

Figure 15 shows results for the experiment when using composable patterns. A1 is not affected by interfering requests from A2 during B, and is not affected by reconfiguration of A2 during C. The green line, exes and circles are on identical positions. Vertically, individual request latencies are shown to be identical. Horizontally, request arrivals occur on the same time instants, resulting in equal run-times. Furthermore, identical behavior over multiple runs indicates that the platform is deterministic, although it is not required for composability. The results show that the reconfigurable SDRAM controller is a composable resource.

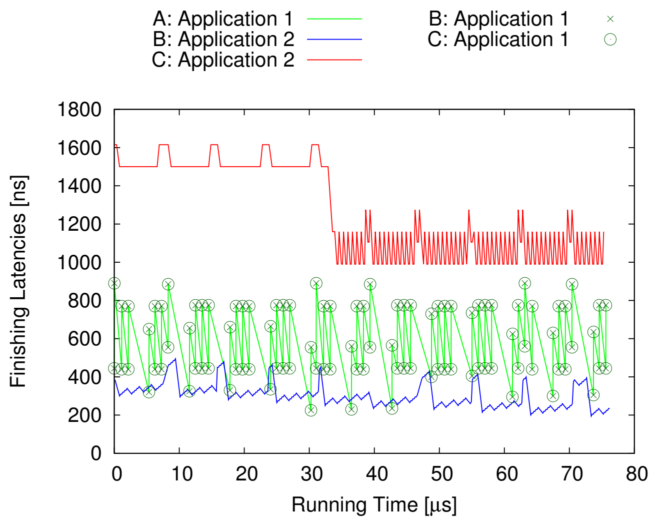


Figure 15. Composable behavior using composable patterns on FPGA

Composable patterns cause an increase in request latency. Using the simulation trace experiments, the pattern set with BI1 and BC2 was found to have an average latency increase of 42% over the tested mediabench applications. For this experiment, average request latencies for A1 are 28% to 56% larger when using a composable pattern set, losses increasing as interference reduces. Run-times when using composable patterns are longer than when using predictable patterns, ranging from 10.4% to 16.5%. Run-time increases are smaller than individual latency increases, because overall run-time also depends on computational delays, which have not changed.

VII. Related Work

Related works are grouped into two categories. Works exist that focus on SDRAM for real-time platforms. Other works focus on reconfiguration of resources for real-time systems.

Predator is not the only SDRAM controller for real-time environments. [22] employs static SDRAM command scheduling at design time. The controller presented in [23], similarly to Predator, dynamically schedules static sequences of commands for isolated SDRAM banks. Alternatively, [24] is able to predictably schedule SDRAM commands dynamically. All of these SDRAM controllers are predictable and can be used for real-time environments. However they are not composable and thus not suitable for mixed time-criticality systems. Furthermore, they are not reconfigurable. [25] presents a SDRAM controller with a programmable Instruction Set Architecture. Reconfiguration options include scheduling policies, address mapping and refresh management and performance nears that of hardwired controllers. Unfortunately, it is not able to give guarantees on performance and is unsuitable for real-time systems.

Studies on reconfiguration of platforms for real-time applications exist. [26] focuses on guaranteed Quality-of-Service (QoS) of applications, for weak-real-time platforms. Their QoS definition still allows applications to have unpredictable behavior under some circumstances. This is unacceptable for CompSoC, which should be able to give firm-real-time guarantees if requested. [27] presents a reconfigurable platform that isolates applications by use of Variable Bandwidth Servers. However, they still accept jitter on expected behavior during reconfigurations. [28] employs a reconfiguration method which offers no service during reconfigurations. They account for this by slightly over-allocating applications service budgets, the excess eventually building up enough reserves for processing a reconfiguration event. Unfortunately, this requires knowledge of reconfiguration latencies and puts a limit on the amount of permissible reconfiguration per time unit. [29], [30] analytically determine worst-case behavior due to application reconfigurations. They reconfigure by use-case and do not isolate applications. [31] is most related to the work presented in this paper. It presents reconfiguration of servers, which are scheduled by non-work-conserving TDM. Servers are dynamically allocated. When an application stops, remaining applications are moved left. Slots for starting applications are added at the end of frame. Arbiter latencies for persistent applications never increase, they may decrease, such that reconfigurations are predictable. Although they use servers for application isolation, they do not consider server reconfiguration under isolation.

Overall, none of these works combine aspects of reconfigurability, real-time requirements, composability and SDRAM as this work does. Neither provides a solution for mixed time-criticality systems. Furthermore, none of the works presenting reconfiguration for real-time platforms have an actual predictable and composable implementation on a real MPSoC.

VIII. Conclusions

This paper addresses the verification problem for reconfigurable mixed time-criticality systems. To allow design-time verification, real-time applications need their real-time requirements satisfied during both regular execution and reconfiguration.

The four main contributions are: 1) A method for composability for SDRAM using TDM arbitration and composable patterns. 2) A reconfiguration method for TDM arbiters, which predictably reconfigures applications in isolation. 3) Implementation and integration of these solutions into CompSoC, both as a SystemC simulation model and hardware platform on FGPA. 4) Extensions to an automated tool flow that instantiates and configures CompSoC with a reconfigurable, predictable and composable SDRAM controller.

These contributions present a reconfigurable, predictable and composable SDRAM controller for CompSoC. $\mathcal{LR}$ guarantees and composability are shown to be valid, such that design-time verification is possible. Furthermore, the method of predictable arbiter reconfiguration is valuable outside CompSoC, as it can be generally applied to time-criticality MPSoCs that employ resource sharing with TDM arbitration.

IX. Future work

Future work on reconfiguration for mixed time-criticality can expand on the work presented here. Extra flexibility within real-time constraints can be added by supporting application *mode changes*. Currently, applications are only reconfigured to accommodate persistent composable applications. Mode changing is the process of applications dynamically changing requirements during their lifetime, initiating their reconfiguration themselves. The current arbiter reconfiguration method already supports mode changing, but support would have to be added in CompSoC. Alternatively, improving performance of current work, effort can be put into allocation strategies, frame size optimization and optimizing composable pattern lengths. Further investigation of latency performance of composable patterns and TDM arbitration may also be appealing. Composable patterns and TDM arbitrations may affect pattern selection because their guarantees on latency performance may be unable to satisfy application latency requirements, whereas predictable patterns and CCSP would have satisfied these. The issue is not investigated for this work because effects are expected to be negligible, but more information on it may prove useful for future usage.

References

[1] C. van Berkel, “Multi-core for Mobile Phones,” in *Proc. DATE*, 2009.

[2] P. Kollig *et al.*, “Heterogeneous Multi-Core Platform for Consumer Multimedia Applications,” in *Proc. DATE*, 2009.

[3] STMicroelectronics and CEA, “Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology,” 2010. White paper.

[4] B. Akesson *et al.*, “Composability and predictability for independent application development, verification, and execution,” in *Multiprocessor System-on-Chip — Hardware Design and Tool Integration* (M. Hübner and J. Becker, eds.), ch. 2, Springer, 2010.

[5] R. Pellizzoni, P. Meredith, M. Nam, M. Sun, M. Caccamo, and L. Sha, “Handling mixed-criticality in soc-based real-time embedded systems,” in *Proceedings of the seventh ACM international conference on Embedded software*, pp. 235–244, ACM, 2009.

[6] R. Cruz, “A calculus for network delay. I. Network elements in isolation,” *Information Theory, IEEE Transactions on*, vol. 37, no. 1, 1991.

[7] S. Sriram and S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*. CRC, 2000.

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

[9] A. Hansson *et al.*, “CoMPSoC: A template for composable and predictable multi-processor system on chips,” *ACM TODAES*, vol. 14, no. 1, 2009.

[10] B. Jacob *et al.*, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann Pub, 2007.

[11] B. Akesson *et al.*, “Automatic Generation of Efficient Predictable Memory Patterns,” in *Proc. RTCSA*, 2011.

[12] S. Goossens, T. Kouters, B. Akesson, and K. Goossens, “Memory-map selection for firm real-time sdram controllers,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pp. 828 –831, march 2012.

[13] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, “Real-time scheduling using credit-controlled static-priority arbitration,” tech. rep., 2008.

[14] B. Akesson *et al.*, “Predator: a predictable SDRAM memory controller,” in *Proc. CODES+ISSS*, 2007.

[15] D. Stiliadis and A. Varma, “Latency-rate servers: a general model for analysis of traffic scheduling algorithms,” *IEEE/ACM Transactions on Networking (ToN)*, vol. 6, no. 5, pp. 611–624, 1998.

[16] H. Shah, A. Knoll, and B. Akesson, “Bounding SDRAM Interference: Detailed Analysis vs. Latency-Rate Analysis,” in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2013.

[17] B. Akesson *et al.*, “Composable resource sharing based on latency-rate servers,” in *Proc. DSD*, 2009.

[18] B. Akesson, *Predictable and Composable System-on-Chip Memory Controllers*. PhD thesis, Eindhoven University of Technology, 2010.

[19] Xilinx, “ML605 Documentation.” http://www.xilinx.com/support/#nav=sd-nav-link-140997&tab=tab-bk, 2012. [Online; accessed 20-Dec-2012].

[20] T. Austin, E. Larson, and D. Ernst, “Simplescalar: An infrastructure for computer system modeling,” *Computer*, vol. 35, pp. 59–67, Feb. 2002.

[21] C. Lee, M. Potkonjak, and W. Mangione-Smith, “Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems,” in *Proc. ACM/IEEE international symposium on Microarchitecture*, 1997.

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

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

[24] M. Paolieri *et al.*, “An Analyzable Memory Controller for Hard Real-Time CMPs,” *Embedded Systems Letters, IEEE*, vol. 1, no. 4, 2009.

[25] M. Bojnordi and E. Ipek, “Pardis: A programmable memory controller for the ddrx interfacing standards,” in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pp. 13 –24, june 2012.

[26] H. Kooti, D. Mishra, and E. Bozorgzadeh, “Reconfiguration-aware real-time scheduling under qos constraint,” in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pp. 141 –146, jan. 2011.

[27] S. Craciunas, C. Kirsch, H. Payer, H. Röck, and A. Sokolova, “Programmable temporal isolation in real-time and embedded execution environments,” in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pp. 19–24, ACM, 2009.

[28] M. Garcia-Valls, P. Basanta-Val, and I. Estevez-Ayres, “Real-time reconfiguration in multimedia embedded systems,” *Consumer Electronics, IEEE Transactions on*, vol. 57, pp. 1280 –1287, august 2011.

[29] L. Santinelli, G. Buttazzo, and E. Bini, “Multi-moded resource reservations,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pp. 37 –46, april 2011.

[30] N. Fisher and M. Ahmed, “Tractable real-time schedulability analysis for mode changes under temporal isolation,” in *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, pp. 130 –139, oct. 2011.

[31] N. Stoimenov, L. Thiele, L. Santinelli, and G. Buttazzo, “Resource adaptations with servers for hard real-time systems,” in *Proceedings of the tenth ACM international conference on Embedded software*, pp. 269–278, ACM, 2010.