2009 12th Euromicro Conference on Digital System Design / Architectures, Methods and Tools

Composable Resource Sharing Based on Latency-Rate Servers

Benny Akesson[1], Andreas Hansson[1], Kees Goossens[2,3]
[1]Eindhoven University of Technology
[2]NXP Semiconductors Research
[3]Delft University of Technology

***Abstract*—Verification of application requirements is becoming a bottleneck in system-on-chip design, as the number of applications grows. Traditionally, the verification complexity increases exponentially with the number of applications and must be repeated if an application is added, removed, or modified. Predictable systems offering lower bounds on performance have been proposed to manage the increasing verification complexity, although this approach is only applicable to a restricted set of applications and systems. Composable systems, on the other hand, completely isolate applications in both the value and time domains, allowing them to be independently verified. However, existing approaches to composable system design are either restricted to applications that can be statically scheduled, or share resources using time-division multiplexing, which cannot efficiently satisfy tight latency requirements.**

In this paper, we present an approach to composable resource sharing based on latency-rate servers that supports any arbiter belonging to the class, providing a larger solution space for a given set of requirements. The approach can be combined with formal performance analysis using a variety of well-known modeling frame works. We furthermore propose an architecture for a resource front end that implements our concepts and provides composable service for any resource with bounded service time. The architecture supports both systems with buffers dimensioned to prevent overflow and systems with smaller buffers, where overflow is prevented with flow control. Finally, we experimentally demonstrate the usefulness of our approach with a simple use case sharing an SRAM memory.

***Index Terms*—composability; latency-rate servers; verification; real-time; resource sharing;**

I. Introduction

The convergence of application domains in new systems-on-chip (SoC) results in complex systems with an increasing number of use cases, comprised of concurrently executing applications. These applications consist of tasks that are mapped on processing elements, such as processors and hardware accelerators [1]. Some applications, such as a Software-Defined Radio [2], have hard real-time requirements and must always meet their deadlines to prevent significant quality degradation. On the other hand, applications with soft real-time requirements, for example a video decoder, tolerate occasional deadline misses, as these only result in a modest reduction of quality [3].

Resources, such as memory channels, are shared between applications to reduce system cost. Resource sharing results in interference between applications, making it difficult verify that application requirements are satisfied without slow system-level simulation of all use cases. This traditionally causes the verification complexity of the system to *increase exponentially* with the number of applications [4]. Use-case verification is furthermore a circular process that must be repeated if an application is added, removed, or modified [5]. Together these factors contribute to making the verification and integration process a dominant part of SoC development, both in terms of time and money.

Predictable and *composable* systems are proposed to manage the increasing verification complexity of SoCs. Predictable systems isolate applications using lower bounds on performance, which is only applicable to a restricted set of applications and systems. Applications in a composable system, on the other hand, are completely isolated both in the value and time without any restrictions on their behavior. Composability simplifies the verification process for the following five reasons [6]: 1) Applications can be simulated in isolation, resulting in a linear and non-circular verification process. 2) Simulating only a single application and its required resources reduces simulation time compared to complete system simulations. 3) The verification process can be incremental and start as soon as the first application is available. 4) Intellectual property (IP) protection is improved, since the verification process no longer requires the IP of independent software vendors to be shared. 5) Functional verification is simplified, since bugs caused by, for instance, race conditions in the integrated application, are independent of other applications.

There are currently three approaches to composable system design. The first involves not sharing any resources, which is prohibitively expensive for systems not running safety-critical applications. The second is to share resources using interfaces where communication is statically scheduled at design time. This approach requires a global notion of time and is limited to applications that can be statically scheduled. The third is to share resources using time-division multiplexing (TDM), which cannot efficiently satisfy tight latency requirements.

In this paper, we present a fourth approach to composable resource sharing that is based on *latency-rate ($\mathcal{LR}$) servers* [7], which is a general framework for analyzing scheduling algorithms. The two main contributions of this paper are: 1) an approach to composability that allows resources to be shared using any arbiter in the class of $\mathcal{LR}$ servers, providing greater service differentiation than TDM. The approach can be optionally be combined with formal performance analysis using a variety of well-known frameworks, such as $\mathcal{LR}$ analysis [7], network calculus [8], or data-flow analysis [9]. 2) An architecture of a resource front end, containing the arbiter, that provides composable service for any resource with bounded service time. The architecture furthermore supports both systems with buffers dimensioned to prevent overflow and systems with smaller buffers, where overflow is prevented with flow control.

978-0-7695-3782-5/09 $25.00 © 2009 IEEE
DOI 10.1109/DSD.2009.167

IEEE computer society

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.

The rest of this paper is organized as follows. We start by reviewing related work in Section II, followed by a conceptual overview of our approach in Section III. A formal model is introduced in Section IV in which we provide a definition of composable service. We then show how $\mathcal{LR}$ servers can be used to provide service according to this definition. In Section V, we propose an architecture for a resource front end that implements the presented concepts when combined with a resource with bounded service time. We experimentally show in Section VI that our front end combined with a priority-based arbiter in the class of $\mathcal{LR}$ servers provides composable service, while satisfying low latency requirements for a simple use case sharing an SRAM. Lastly, conclusions are presented in Section VII.

II. Related work

A number of works in the field of high performance computing discuss performance isolation of applications in predictable systems by providing lower bounds on performance. Fair Queuing Memory Systems [10] and Virtual Private Caches [11] are both part of the Virtual Private Machine framework [12] for multi-core resource management. The authors show that the service provided by a Virtual Private Machine running at an allocated fraction of the original capacity is at least as good as a real private machine with the same resources. This allows real-time requirements to be verified in isolation, assuming that the applications executing on the system are *performance monotonic* [13], which means that having additional resources cannot result in worse performance.

Two simulation-based approaches to verification of real-time requirements in predictable systems are presented in [13], [14]. The idea in these papers is to simulate the execution of an application and verify that real-time requirements are satisfied when emulating maximum interference from other applications by delaying responses until their worst-case latency. This is similar to the work presented in this paper, although with some important differences. In contrast to our work, the authors propose to disable worst-case interference emulation when deploying the system to benefit from slack and increase performance. This breaks the isolation between applications, limiting the approach to applications and systems that either have performance monotonic execution, or can be captured in a performance monotonic model, such as deterministic data-flow graphs [15]. Furthermore, no hardware architecture is presented for the approach in [13], although our proposed resource front end can be used to implement the methodology.

The drawback of relying on performance monotonicity is that it severely restricts both the supported platform and application software. The platform has to be free from timing anomalies, which can appear in shared caches or with dynamically scheduled processors, such as PowerPCs [16]. Timing anomalies also appear in multi-processor systems [17], making verification results of distributed applications unreliable. Applications can furthermore not have timing dependent behavior, such as adapting the quality level of a video decoder based on decoding time of previous frames.

Verification of composable systems, on the other hand, does not rely on performance monotonicity, since applications are completely independent of each other in both the value and time domains. There are currently three approaches to composable system design. The first involves not sharing any resources, which is used by federated architectures in the automotive and aerospace industries [18]. This method is trivially composable, but prohibitively expensive for systems that do not have safety-critical applications. The second option is the time-triggered approach [5] that shares resources using interfaces where the time instances for communication are specified in a sparse time base at design time. This approach requires a global notion of time and is limited to applications that can be statically scheduled at design time. The third approach is to dynamically schedule resource access at run time using TDM, as proposed in [6], [19]. Using run-time scheduling has the benefit of supporting event-triggered systems, although a limitation of TDM is that it couples the worst-case latency and the allocated bandwidth of an application. This prevents low latency from being provided to applications with low bandwidth requirements without over allocating and wasting resources.

This work adds a fourth approach to composability that is based on predictability, but adapted to remove the severe restrictions on the platform and the applications. The approach allows resources to be shared using any arbiter in the class of $\mathcal{LR}$ servers. This allows greater flexibility in the choice of arbiter thus increasing the solution space for a given set of application requirements.

III. Conceptual overview

In this section, we provide a conceptual overview of our approach to composable resource sharing and explain the benefits. We consider a system in which the tasks of an application are mapped to one or more processing elements. We refer to the entities sharing the resources as *requestors*, corresponding to ports on the processing elements to which the tasks are mapped. In this work, we focus on sharing of slaves, such as memories and peripherals, and assume that all processing elements and interconnect are either not shared between applications, or shared in a composable manner according to any of the previously mentioned approaches. A requestor and the resource communicate by sending requests and responses, as shown in Figure 1. Requests are stored in a Request Buffer in front of the resource until they are scheduled by an arbiter. The resource processes the request and stores a response in the Response Buffer of the requestor when it is finished. A flow-control mechanism is available that allows a receiving block to stall a sending block to prevent a buffer overflow. A difficulty with run-time arbitration is that it typically causes the times at which the resource accepts requests and sends responses to a requestor to change due to variable interference from other requestors, making the system non-composable.

The key idea behind our approach is to make the system composable again by removing the variation in interference. We accomplish this by delaying responses and flow control

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.

sent to the requestor to emulate maximum interference from other requestors. The interface of each requestor is hence independent of others in the temporal domain, as shown in Figure 1. This makes the system composable on the level of requestors, which is a sufficient condition for it to be composable on the level of applications.

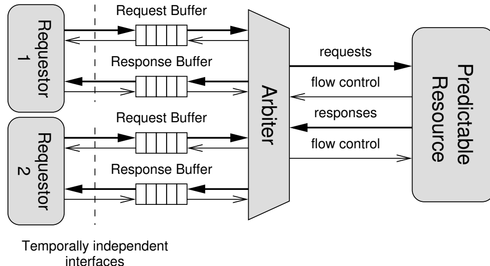

Fig. 1. Temporally independent interfaces are created by delaying responses and flow control.

Our approach to composable resource sharing is based on $\mathcal{LR}$ servers. In essence, a $\mathcal{LR}$ server guarantees a requestor a minimum allocated service rate, ρ', after a maximum service latency, Θ, as shown in Figure 2. The allocated service rate corresponds to reserved bandwidth in case of a memory channel, and can be either a worst-case or average-case allocation, depending on the design methodology. The service latency intuitively corresponds to the maximum interference from other requestors. This separation of *interference due to other requestors* from *self interference*, which is the time a request waits for other requests from its own requestor, is a benefit of the $\mathcal{LR}$ server model, since composability only requires us to eliminate the effects of the former.

The motivation for basing our approach on $\mathcal{LR}$ servers is that it enables us to transparently use any arbiter belonging to the class, hence allowing the choice arbiter to be matched to the given set of requirements. It is shown in [7] that many well-known arbiters, such as Weighted Round-Robin [20], Deficit Round-Robin [21], and several varieties of Fair Queuing [22] are $\mathcal{LR}$ servers. Other arbiters in the class are Credit-Controlled Static-Priority arbitration [23], which uses priorities, and TDM [24]. Note that using different arbiters enable service differentiation even though worst-case service is enforced. For instance, the maximum latency of a high priority requestor in a priority-based arbitration scheme is lower than its corresponding worst-case latency using TDM or Round Robin. Another benefit of $\mathcal{LR}$ servers is that they support formal performance analysis using approaches based on either $\mathcal{LR}$ analysis [7], network calculus [8], or data-flow analysis [9]. This enables the possibility to formally verify applications that can be modeled in any of these frame works.

Our approach to composability is based on predictability. More specifically, we require *predictable resources*, where the time to serve a scheduled request is upper bounded. This is not a severe limitation, as it applies to most interesting memories and peripherals. We furthermore require an upper bound on the interference from other requestors. Given a predictable resource, this requirement can be satisfied in three ways: 1) by

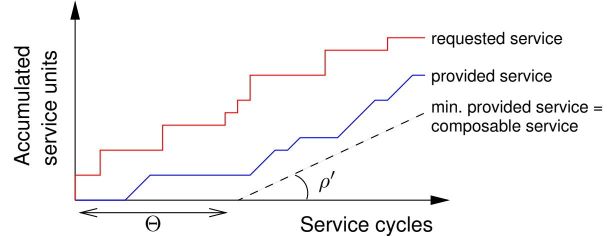

Fig. 2. Example of predictable and composable service in a $\mathcal{LR}$ server.

characterizing the requestors and derive an upper bound on the size of a request, as done in [5]. This allows non-preemptive scheduling to be used, but is not robust in case the characterization is wrong or a requestor malfunctions. 2) Preempt a request in service after a maximum time, accomplished by a TDM scheduler in [19]. This solution is robust and can handle requests whose sizes are initially unknown, but is limited to preemptive schedulers. 3) Use a hardware block to split up requests into small *atomic service units*, referred to as atoms, with known maximum service time, as proposed in [6]. This solution assumes that requests can be split into smaller pieces, which is typically the case for transaction-based resources like memory channels and peripherals. We choose this option for our approach, since it enables preemption of requests on the granularity of atoms using any arbiter in the class of $\mathcal{LR}$ servers, thus providing maximum flexibility.

IV. Formal Model

In this section, we formally show how to provide composable service based on $\mathcal{LR}$ servers by deriving and enforcing temporal bounds. We start by explaining how service curves are used to model the interaction between the requestors and the resource in Section IV-A. This allows us to define composable service. We then proceed in Section IV-B by defining a $\mathcal{LR}$ server and showing that they can provide composable service according to our definition.

Throughout this paper, we use capital letters (A) to denote sets, hats to denote upper bounds ($\hat{a}$), and checks to denote lower bounds ($\check{a}$). Subscripts are used to disambiguate between variables belonging to different requestors, although for clarity these subscripts are omitted when they are not required. To deal with different resources in a uniform way, we adopt an abstract resource view, where a service unit corresponds to the access granularity of the resource. Time is discrete and a time unit, referred to as a *service cycle*, is defined as the time required to serve such a service unit. The translation from service cycles to clock cycles is solved by multiplying the number of service cycles with the maximum service cycle length, which is known and bounded for a predictable resource.

A. Service curves

We use cumulative service curves to model the interaction between the resource and the requestors. We let $\xi(t)$ denote the value of a service curve ξ at service cycle t. We furthermore use $\xi(\tau,t) = \xi(t+1) - \xi(\tau)$ to denote the difference in values between the endpoints of the closed interval $[\tau, t]$.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.

A requestor generates requests of variable but bounded size, as stated in Definition 1. A request is considered to arrive as an impulse when: 1) it has completely arrived in the Request Buffer and 2) there is enough space in the Response Buffer to store a response, as stated by Definition 2. The service requested by a requestor is represented by the requested service curve, w, defined in Definition 3.

Definition 1 (Request): The k:th request ($k \in \mathbb{N}$) from a requestor $r \in R$ is denoted $\omega_r^k \in \Omega_r$. The size of ω_r^k in service units is denoted $s(\omega_r^k) : \Omega_r \rightarrow \mathbb{N}^+$.

Definition 2 (Arrival time): The arrival time of a request ω_r^k from a requestor $r \in R$ is denoted $t_a(\omega_r^k) : \Omega_r \rightarrow \mathbb{N}^+$, and is defined as the smallest t at which the last bit of ω_r^k has arrived in the Request Buffer and there is enough free space in the Response Buffer to store a response.

Definition 3 (Requested service curve): The requested service curve of a requestor $r \in R$ is denoted $w_r(t) : \mathbb{N} \rightarrow \mathbb{N}$, where $w_r(0) = 0$ and

$$w_r(t+1) = \begin{cases} w_r(t) + s(\omega_r^k) & \omega_r^k \text{ arrived at } t+1 \\ w_r(t) & \text{no request arrived at } t+1 \end{cases}$$

The scheduler in the resource arbiter attempts to schedule a requestor every service cycle according to its particular scheduling policy. We let $\gamma(t) : \mathbb{N} \rightarrow R \cup \{\varnothing\}$ denote the scheduled requestor at time t, where $\varnothing$ represents the case where no requestor could be scheduled. We consider preemptive scheduling, as mentioned in Section III, and refer to the first service cycle in which a request ω^k is scheduled as its starting time, $t_s(\omega^k)$, according to Definition 4.

Definition 4 (Starting time of a request): The starting time of a request ω_r^k is denoted $t_s(\omega_r^k) : \Omega_r \rightarrow \mathbb{N}$, and is defined as the smallest t at which ω_r^k is scheduled.

The provided service curve, w', reflects the number of service units provided by the resource to a requestor. The provided service curve is defined in Definition 5. The finishing time of a request corresponds to the first service cycle in which a request is completely served, formally defined in Definition 6. This corresponds to the earliest time at which the response is guaranteed to be available in the Response Buffer. An illustration of a requested service curve and a provided service curve along with their related concepts is provided in Figure 3.

Definition 5 (Provided service curve): The provided service curve of a requestor $r \in R$ is denoted $w'_r(t) : \mathbb{N} \rightarrow \mathbb{N}$, where $w'_r(0) = 0$ and

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

Definition 6 (Finishing time of a request): The finishing time of a request ω_r^k is denoted $t_f(\omega_r^k) : \Omega_r \rightarrow \mathbb{N}$, and is defined as $t_f(\omega_r^k) = \min(\{t \mid t \in \mathbb{N} \land w'_r(t) = w'_r(t_s(\omega_r^k)) + s(\omega_r^k)\})$.

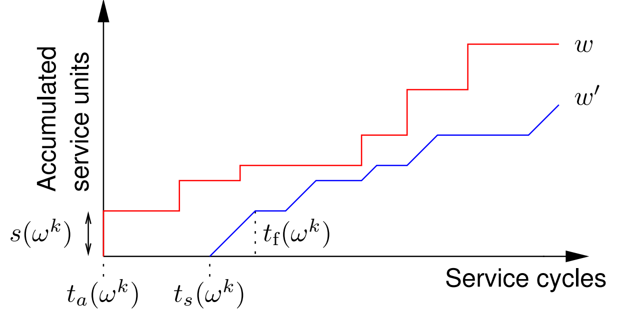

Fig. 3. Service curves and representations of the surrounding concepts.

We conclude this section by providing a definition of composable service in Definition 7.

Definition 7 (Composable service): The service provided to a requestor is defined as composable if both the starting times and finishing times of all requests from the requestor are independent of other requestors.

B. $\mathcal{LR}$ servers

In this section, we define a $\mathcal{LR}$ server in our formal model, and explain how it provides bounds on the starting times and finishing times of the requests by considering the maximum interference from other requestors. This allows us to satisfy our definition of composable service by delaying actual starting times and finishing times until their corresponding bounds, as we will explain in Section V. We start by defining the allocated service of a requestor in Definition 8.

Definition 8 (Allocated service): A requestor $r \in R$ is allocated a fraction of the available resource capacity $\rho'_r \in \mathbb{R}^+, 0 \leq \rho'_r \leq 1$. For a valid allocation it holds that $\sum_{\forall r \in R} \rho'_r \leq 1$.

We continue by defining a $\mathcal{LR}$ server. We use the definitions from [7], adapted to fit with our use of discrete, as opposed to continuous, time. The concept of *busy periods*, defined in Definition 9, is central to the definition of $\mathcal{LR}$ servers. A busy period is intuitively understood as a period in which a requestor requests at least as much service on average as it has been allocated. We refer to a requestor as a busy requestor during its busy periods. Definition 10 defines a $\mathcal{LR}$ server as a server that guarantees a busy requestor its allocated service rate, ρ', after a maximum service latency, Θ, thus providing a lower bound on provided service, w'. This is illustrated in Figure 4. The requestor in the figure is busy from $t_a(\omega^k)$ until τ_1 when it catches up with the reference line with slope ρ' that we informally refer to as the *busy line*. A second busy period starts at τ_2 with the arrival of request ω^{k+3} and lasts throughout the rest of the shown interval.

Definition 9 (Busy period): A busy period of a requestor $r \in R$ is defined as a maximum interval $[\tau_1, \tau_2]$, such that $\forall t \in [\tau_1, \tau_2] : w_r(\tau_1 - 1, t - 1) \geq \rho'_r \cdot (t - \tau_1 + 1)$.

Definition 10 ($\mathcal{LR}$ server): A server is a $\mathcal{LR}$ server if and only if a non-negative constant c_{r_i} can be found such that

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.

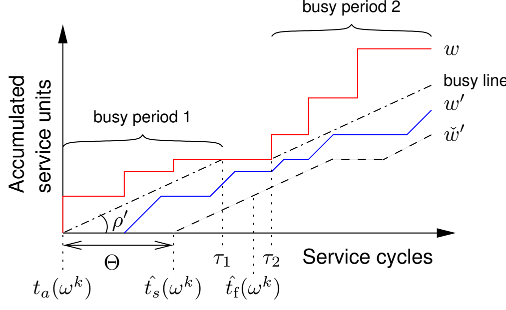

Fig. 4. Example service curves in a $\mathcal{LR}$ server.

Equation (1) holds during a busy period $[\tau_1, \tau_2]$. The minimum non-negative constant c_r satisfying the equation is defined as the *service latency* of the server, denoted Θ_r.

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

The lower bound on provided service in Equation (1) is useful to determine an upper bound on the finishing time of a request in a $\mathcal{LR}$ server. It is shown in [9] that the worst-case finishing time of a request ω_r^k is computed according to Equation (2), where $t_a(\omega_r^{-1}) = 0$.

$$\hat{t}_f(\omega_r^k) = \max(t_a(\omega_r^k) + \Theta_r, \hat{t}_f(\omega_r^{k-1})) + s(\omega_r^k)/\rho'_r \quad (2)$$

The worst-case finishing time in Equation (2) consists of two terms. The first term comprises a max expression that corresponds to the worst-case starting time of the request, $\hat{t}_s(\omega_r^k)$, expressed in Equation (3). This term is determined by the service latency of the arbiter, or by the finishing time of the previous request from the requestor, whichever is larger. The second term represents the time required to finish serving the request once it is scheduled, referred to as the *completion latency*, which is based on the size of the request and the allocated service rate. In Figure 4, the completion latency of ω^k corresponds to the time between $\hat{t}_s(\omega^k)$ and $\hat{t}_f(\omega^k)$.

$$\hat{t}_s(\omega_r^k) = \max(t_a(\omega_r^k) + \Theta_r, \hat{t}_f(\omega_r^{k-1})) \quad (3)$$

The bounds in Equations (2) and (3) both require that requests have completely arrived and that there is enough space in the Response Buffer to store a response before being scheduled. These are preconditions for latency bounds based on $\mathcal{LR}$ servers to ensure that a scheduled requestor cannot stall the resource and prevent accesses from other requestors. Both of these preconditions are satisfied in our approach, since requests with insufficient Response Buffer space are not considered to have arrived, according to Definition 2.

Note that the bounds are based on *worst-case interference from other requestors*, but only on *actual-case self interference* through the dependency on previous requests from its requestor. This means that the maximum time between the arrival time and finishing time is not constant for all requests, but changes depending on the number of requests in the Request Buffer of the requestor. Enforcing a constant delay from arrival time to finishing time requires a conservative bound on the requested service, using for instance a (σ, ρ) characterization [8], to compute the worst-case self interference for every request. This results in very pessimistic finishing times and a lower service rate than allocated, as we will see in Section VI. It is furthermore very difficult to obtain an accurate characterization without unnecessarily restricting the application, which does not fit with our approach to composability.

V. Front-end Architecture

In this section, we introduce the architecture of our proposed resource front end that implements the concepts from Section III based on the model from Section IV. We start by presenting an overview of the architecture in Section V-A, followed by descriptions of the functional blocks in Sections V-B through V-D.

A. Architecture overview

The proposed resource front end is located in front of a predictable resource, as shown in Figure 5. The architecture is comprised of three main simple and reusable blocks: an Atomizer, a Delay Block, and a Data Bus. Additionally, there is a Configuration Bus that allows registers inside the different blocks to be programmed via memory mapped I/O during use-case transitions [25]. The blocks communicate using a *device transaction level* (DTL) protocol, which is a standardized communication protocol similar to AXI. All ports shown in Figure 5 are DTL ports. The black ports are used to communicate requests and responses, and the white ports are for configuration data.

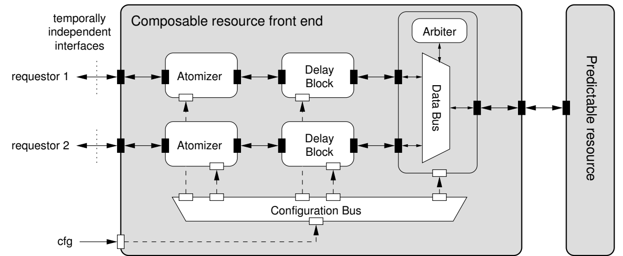

Fig. 5. An instance of the proposed architecture supporting two requestors.

The architecture achieves composability by combining the approaches to composable system design, explained in Section II, at the block level. The Atomizer and Delay Blocks are composable because they are not shared with other requestors, corresponding to the first approach. The Data Bus shares the predictable resource using an arbiter in the class of $\mathcal{LR}$ servers. The Delay Block hides the interference caused by scheduling and accessing the resource by emulating worst-case interference from other requestors, according to the proposed fourth approach. This creates an interface per requestor that is temporally independent of the behavior of other requestors, as shown in Figure 5.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.

B. Atomizer

The Atomizer is responsible for splitting requests into homogeneous atoms with a fixed programmable size. This ensures that requests have a known size that can be served in a bounded time by the resource, making the design predictable, as explained in Section III. Fixed-sized requests furthermore simplify other blocks in the architecture. The size of an atom corresponds to the service unit of the resource, mentioned in Section IV, and we choose it to be the minimum request size that can be efficiently served by the resource. For a single-bank SRAM, the natural service unit is a single word, but for an SDRAM it might be bursts of four or eight words, or even much larger [26]. The original sizes of the requests are stored in the Atomizer to allow it to merge arriving responses back into the size expected by the requestor.

C. Delay Block

The most complex block in the architecture is the Delay Block, shown in Figure 6, and we hence explain this block in greater detail than the rest. The purpose of the Delay Block is to absorb the variation in interference from other requestors to provide a composable interface towards the Atomizer. This makes the interface of the entire front end composable, since the Atomizer is not shared. The Delay Block is composable if all arrows on the interface in Figure 6 pointing left towards the Atomizer exhibit composable behavior, which implies that both response data and flow control signals must emulate maximum interference. We proceed by discussing how to accomplish this in Section V-C1 and Section V-C2, respectively. We then discuss how to configure the Delay Block in Section V-C3, before presenting a mechanism to approximate non-integer completion latencies in Section V-C4.

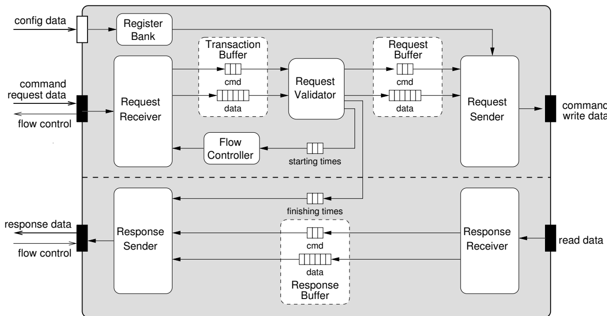

Fig. 6. Delay Block architecture.

1) Composable responses: Requests are received by the Request Receiver according to the DTL protocol. Incoming requests are split into a command (read/write information and request size) and data (for write requests), and are stored in a Transaction Buffer. The Request Validator monitors the incoming request and holds it until it has completely arrived in the Transaction Buffer and there is enough space to store its response in the Response Buffer, implementing the definition of arrival in Definition 2. Once a request has arrived, the Request Validator enqueues it in the Request Buffer and computes the worst-case finishing time and the worst-case starting time, according to Equations (2) and (3), and stores the results in two respective FIFO buffers.

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

Responses are received by a Response Receiver and are stored in a Response Buffer. The Response Sender pops the worst-case finishing time from the head of the FIFO buffer and waits until the appropriate clock cycle to release the response, thus emulating maximum interference according to the $\mathcal{LR}$ server model. This ensures that the finishing times of the requestor are unaffected by variations in the interference from others, which is one of the two requirements to be composable according to Definition 7.

2) Composable flow control: Having taken care of composable responses, we proceed by discussing the issue of composable flow control. Both the DTL and AXI protocols feature flow-control mechanisms that allow a receiving block to stall a sending block in case the receiving buffer is full. This may cause non-composable behavior if a request is scheduled earlier than its worst-case starting time, causing its space in the Request Buffer to be released prematurely. If the Request Buffer was previously full, the next request gets an earlier arrival time and possibly also an earlier worst-case finishing time than it would if there had been maximum interference, violating the definition of composability in Definition 7. We address this problem by basing the flow control on the worst-case buffer filling, which is emulated by a Flow Controller. The Flow Controller has a counter that is initialized to the size of the Request Buffer. The counter is decremented whenever a request enters the Request Buffer and incremented at the computed worst-case starting times. This ensures that the starting times of the requestor are unaffected by variations in the interference from others, which is the remaining requirement to provide composable service according to Definition 7.

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

For the computed finishing times to be correct, the number of pipeline stages between the Request Buffer and the Response Buffer have to be considered. Every block in our implementation is output registered, resulting in a total of four pipeline stages. Four clock cycles should hence be added to the service latency to account for the pipelining in the implementation. It might seem more intuitive to add this term to the completion latency, which accounts for the time between the scheduling time and the finishing time. This would, however, not correctly model that a pipeline adds a constant latency to all finishing times. Instead, each request

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.

during a busy period would be delayed an additional four cycles compared to the previous one, resulting in reduced throughput.

Composable service can be dynamically disabled by programming both the service latency and completion latency to be zero clock cycles. This feature can be used to implement the METERG methodology [13], where worst-case interference is emulated only during verification. This, however, restricts the supported hardware and software, as mentioned in Section II.

4) Discrete approximation mechanism: A problem arises if the completion latency, $1/\rho'$, is not an integer multiple of clock cycles, which it typically is not. Rounding off the programmed value causes the enforced worst-case finishing times to diverge from the exact values over time for a busy requestor. As we will see in Section VI, this divergence is significant for requestors with high allocated rates for resources with small service units, where completion latencies are in the order of a few clock cycles. Rounding the programmed value downwards makes the finishing times too optimistic, leading to non-composable behavior. On the other hand, rounding upwards makes the finishing time too pessimistic and causes the actual provided service rate, ρ^*, to be less than the allocated service rate, ρ'. This problem is illustrated in Figure 7. Note that the requestor in the figure is busy throughout the entire shown interval, although the busy line has been omitted for clarity.

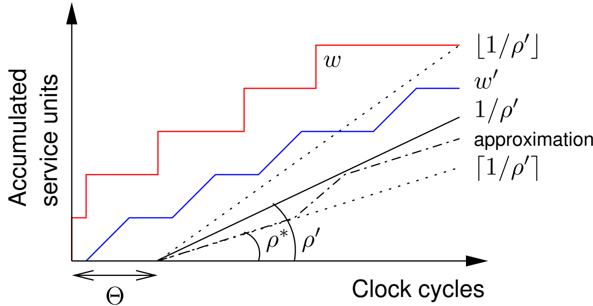

Fig. 7. Diverging finishing times prevented by discrete approximation of the completion latency.

Our solution to this problem is to implement a mechanism that alternates between using the rounded up and rounded down completion latencies in a weighted fashion to conservatively approximate the actual value, as shown in Figure 7. The fraction of the service units for which the rounded down value should be used is expressed as $\mathrm{f} = \lceil 1/\rho' \rceil - 1/\rho'$. Since $\mathrm{f} \in \mathbb{R}^+$ and $0 \leq \mathrm{f} < 1$, our mechanism uses a discrete approximation based on integer arithmetic that has a fast and simple hardware implementation. Similarly to the rate approximation technique in [27], we represent f as a fraction of integers according to $\mathrm{f} = n/d$, where $n, d \in \mathbb{N}^+$ and $n \leq d$. The values of n and d are chosen to be the $(n,\ d)$ pair that provides the closest approximation of f, referred to as a closest rate approximation in [27]. The accuracy of this approximation is only limited by the number of bits used to represent n and d. The n and d are computed for all requestors and use cases at design time and are programmed per use case at run time.

The behavior of the approximation mechanism is based on a credit counter c, as described by the pseudo code in Figure 8. The credit counter is set to zero at the start of a busy period, which is detected by checking if the first parameter of the max expression in Equation (2) is larger than the second. The mechanism then alternates between the rounded up and the rounded down completion latencies based on the value of the counter. The approximation done by the mechanism is conservative and guarantees that the maximum difference between the approximated and actual completion latency is less than one clock cycle at any time.

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

  if $c_r < d_r - n_r$ **then** // Rounding up
    $c_r := c_r + n_r$
    $\hat{t}_{\mathrm{f}}(\omega_r^k) := \max(t_a(\omega_r^k) + \Theta_r, \hat{t}_{\mathrm{f}}(\omega_r^{k-1})) + \lceil 1/\rho_r' \rceil$
  else // Rounding down
    $c_r := c_r + n_r - d_r$
    $\hat{t}_{\mathrm{f}}(\omega_r^k) := \max(t_a(\omega_r^k) + \Theta_r, \hat{t}_{\mathrm{f}}(\omega_r^{k-1})) + \lfloor 1/\rho_r' \rfloor$
  end if
end for

Fig. 8. Mechanism for discrete approximation of completion latency.

D. Data bus

The Data Bus is a regular DTL bus that periodically schedules requests, according to the policy of an attached arbiter that belongs to the class of $\mathcal{LR}$ servers. The periodic scheduling signal is generated by a simple clock cycle counter that repeatedly counts down from the programmed maximum service cycle length. When the arbiter schedules a request, the Data Bus stores an identifier to the scheduled port so that responses are demultiplexed to their respective Delay Blocks. These identifiers are stored in separate FIFO buffers for read and write requests, since the DTL protocol does not enforce ordering between reads and writes.

VI. Experiments

In this section, we experimentally evaluate our approach to composable resource sharing using a simple use case. First, we present the experimental setup in Section VI-A. We then proceed in Section VI-B by illustrating the problem of satisfying tight service latency requirements using TDM, and show how this problem is resolved by using a priority-based arbiter in the class of $\mathcal{LR}$ servers. We conclude in Section VI-C by experimentally verifying that the computed bounds on finishing time are conservative and evaluate their tightness. We furthermore show that the starting times and finishing times of a requestor are independent of other requestors and hence that our design provides composable service according to Definition 7.

A. Experimental setup

Our experimental setup consists of a cycle-accurate SystemC implementation of a predictable and composable multiprocessor SoC. Traffic generators with exponential request distributions are used to simulate processing elements that are

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.

interconnected using a model of the Æthereal [28] network-on-chip. As an example resource, we use a model of an SRAM controller running at 200 MHz with a 32-bit data path, offering a bandwidth of 800 MB/s. The service unit size of this controller is a single word (4 bytes), and the length of a service cycle is one clock cycle.

For clarity, we use a simple use case with four requestors, shown in Table I, for our experiments. Three of the requestors issue read requests, and one issues write requests. The request sizes are different for all requestors, but they are all integer multiples of the service unit size. One of the requestors, r_0, is latency critical, but only requires a bandwidth of 20 MB/s. On the other hand, r_1, r_2, and r_3 are latency tolerant, but process large amounts of data, requiring a bandwidth of 260 MB/s each. This results in a total allocated load of 100 % of the offered bandwidth.

B. Comparison with TDM

Using a TDM scheduler, the best-case service latency is achieved if the reserved slots of a requestor are placed equidistantly in the schedule. In this case, $\Theta_r^{tdm} = \lceil 1/\rho'_r - 1 \rceil$. The service latencies of the requestors in the use case, including the four clock cycles accounting for the pipeline stages in the architecture, are shown in Table I. We see that the low allocated service rate results in a very high service latency for r_0, who is latency critical. The only way to reduce the service latency using TDM is to increase the allocated service rate, wasting bandwidth. Our use case, however, already uses all the available bandwidth, resulting in that a tight latency requirement cannot be satisfied with TDM.

TABLE I
REQUESTOR CONFIGURATION AND RESULTS.

Req.	*Type*	*Size [B]*	p	σ'	ρ'	Θ^{tdm} *[cc]*	Θ^{ccsp} *[cc]*	$1/\rho'$ *[cc]*
r_0	Read	32	0	1.0	0.025	43	4	40.00
r_1	Read	64	1	1.0	0.325	7	5	3.08
r_2	Read	4	2	1.0	0.325	7	7	3.08
r_3	Write	16	3	1.0	0.325	7	13	3.08

Instead, we use a Credit-Controlled Static-Priority (CCSP) arbiter [23], which combines the use of rate regulation and priorities to provide differentiated service guarantees. The service latency of a requestor r_i using CCSP, expressed in service cycles, is computed according to Equation (4). In the equation, $R_{r_i}^+$ denotes the set of requestors with higher priority than r_i, ρ'_{r_i} the allocated service rate, and σ'_{r_i} the allocated burstiness. The allocated burstiness is configured according to $\sigma' = 1$ for all requestors in the use case, which is the smallest allocated burstiness that allows CCSP to act as a $\mathcal{LR}$ server. The requestors in the table have descending priorities, indicated by unique priority levels, p.

$$\Theta_{r_i}^{ccsp} = \left\lceil \frac{\sum_{\forall r_j \in R_{r_i}^+} \sigma'_{r_j}}{1 - \sum_{\forall r_j \in R_{r_i}^+} \rho'_{r_j}} \right\rceil \quad (4)$$

Table I shows how these priority levels affect the service latencies. We note that the highest priority requestor has a service latency of just four clock cycles, corresponding to the four pipeline stages in the architecture. This shows the benefit of using a priority-based arbiter to satisfy tight latency requirements. The completion latencies, $1/\rho'_r$, are the same with both arbiters, since they guarantee the same allocated rate. Note, however, that the exact completion latencies of r_1, r_2, and r_3 are 3.08 clock cycles. As mentioned in Section V-C4, rounding this value downwards might lead to non-composable behavior, and rounding it upwards results in that the provided bandwidth is reduced from 260 MB/s to 200 MB/s (1 word / 4 clock cycles), failing to satisfy the bandwidth requirements of the requestors. This is prevented by our proposed approximation mechanism.

C. Simulation results

For our first experiment, we simulate the use case in Table I during 1 ms to observe the behavior of the front end and the predictable resource. We let the size of the Response Buffer be large enough to prevent overflow, since it allows us to evaluate both the added latency and buffering that follows from delaying responses. Figure 9 plots the worst-case finishing times, the actual finishing times and the actual scheduling times versus the arrival times of the first 200 requests from requestor r_2. We see that the worst-case finishing times are larger than the actual finishing times, indicating that the bound is conservative in the shown interval. The minimum difference between the worst-case and actual finishing times during this simulation is 3 clock cycles, suggesting that the bound is rather tight. We note that the difference between the worst-case finishing time and the arrival time in Figure 9 is not constant for all requests, as mentioned in Section III. The drawback of enforcing a constant time between the arrival time and finishing time is that the constant would have to be equal to our longest value (115 clock cycles in this simulation). This value would still assume a perfect characterization of the requested service and its resulting self interference, which is very difficult to obtain. The average actual finishing time and the average worst-case finishing time during the simulation are 17.6 and 23.6 clock cycles after the corresponding arrivals, respectively. This corresponds to an increase of 34%, showing that enabling composable service makes it more difficult to satisfy requirements on average-case latency. Delaying responses furthermore implies that more data has to be stored in the Response Buffer to prevent reducing throughput. The amount of extra data to buffer is related to the tightness of the bound on finishing time, since this determines the extra time an atom spends in the Response Buffer before being released. Without delaying responses, the read requestors have a maximum Response Buffer filling of one word each, since responses are immediately passed on to the Atomizer. When enabling delays, the maximum buffer filling increases with one word for r_1 and two words for r_2. These results are not unexpected, since the requests of r_2 are buffered an extra 6 clock cycles on average, roughly corresponding to two completion latencies.

We proceed by experimentally showing that our design provides composable service. In this experiment, we show the consequences of small changes in application software by simulating the use case twice (case 1 and case 2) with different

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.

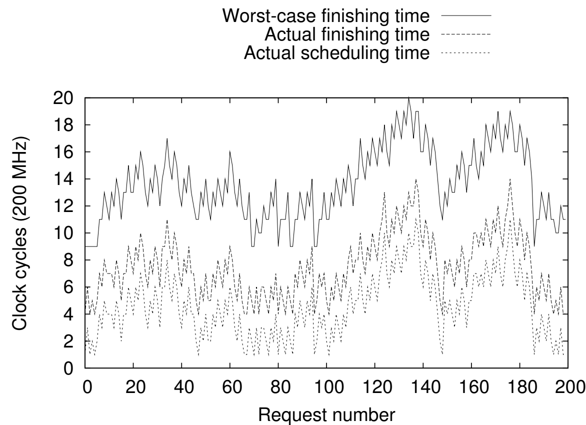

Fig. 9. The first 200 request of r_2 in the use case.

variances in the request generation for r_0. We additionally increase the allocated burstiness of r_0 in Table I according to $\sigma'_{r_0} = 8$. This creates larger service variations for lower priority requestors, allowing us to visualize our point more clearly. The results for requestor r_2 are shown in Figure 10. We see that changing the variance causes the actual finishing times of the requests to change, making the system non-composable. However, in our design the requests are held in the Delay Block until their worst-case finishing times, which are completely overlapping for the two cases, indicating that requests are released from the Delay Block at the same time regardless of these changes.

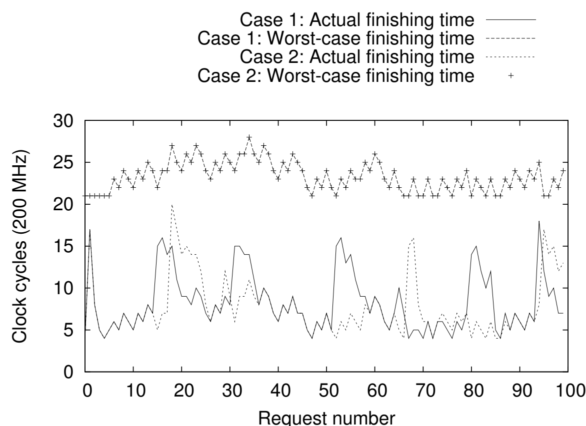

Fig. 10. Request releases of r_2 are unaffected when the behavior r_0 changes.

VII. Conclusions

Composable systems are proposed to mitigate the increasing verification complexity of application requirements in systems-on-chip, since they allow applications to be independently verified by simulation. However, current approaches to composable system design are either restricted to applications that can be statically scheduled, or share resources using time-division multiplexing (TDM), which cannot efficiently satisfy tight latency requirements.

This paper introduces an approach to composable resource sharing that supports any arbiter in the class of latency-rate ($\mathcal{LR}$) servers. The key idea is to delay responses and flow control sent from the resource to an application to emulate maximum interference from other applications. We furthermore present an architecture for a resource front end that provides composable service for any predictable resource using our approach. We show that TDM fails to satisfy tight latency requirements in a simple use case sharing an SRAM controller. We then demonstrate that our front end provides composable service that satisfies the requirements when combined with a priority-based arbiter in the class of $\mathcal{LR}$ servers. We experimentally show that the cost and performance impact of our approach is limited for the considered use case. The average latency of a memory request is increased by 6 clock cycles and the additional buffering requirements are in the range of a few words, compared to if the same arbiter is used without delays.

References

[1] S. Dutta *et al.*, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," *IEEE Des. Test. Comput.*, 2001.

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

[3] L. Abeni and G. Buttazzo, "Resource Reservation in Dynamic Real-Time Systems," *Real-Time Systems*, vol. 27, no. 2, 2004.

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

[5] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, 2003.

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

[7] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Trans. Netw.*, vol. 6, no. 5, 1998.

[8] R. Cruz, "A calculus for network delay. I. Network elements in isolation," *IEEE Trans. Inf. Theory*, vol. 37, no. 1, 1991.

[9] M. H. Wiggers *et al.*, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," in *Proc. SCOPES*, 2007.

[10] K. J. Nesbit *et al.*, "Fair queuing memory systems," in *Proc. MICRO 39*, 2006.

[11] ——, "Virtual private caches," in *Proc. ISCA*, 2007.

[12] ——, "Multicore resource management," *IEEE Micro*, vol. 28, no. 3, 2008.

[13] J. Lee and K. Asanovic, "METERG: Measurement-Based End-to-End Performance Estimation Technique in QoS-Capable Multiprocessors," in *Proc. RTAS*, 2006.

[14] M. Paolieri *et al.*, "Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems," in *Proc. ISCA*, 2009.

[15] M. Bekooij *et al.*, "Performance guarantees by simulation of process networks," in *Proc. SCOPES*, 2005.

[16] T. Lundqvist and P. Stenstrom, "Timing anomalies in dynamically scheduled microprocessors," in *Proc. RTSS*, 1999.

[17] R. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, pp. 416–429, 1969.

[18] H. Kopetz *et al.*, "Composability in the time-triggered system-on-chip architecture," in *Proc. SOCC*, 2008.

[19] M. Bekooij *et al.*, "Predictable and Composable Multiprocessor System Design: A Constructive Approach," in *Bits&Chips Symposium on Embeddd Systems and Software*, 2007.

[20] M. Katevenis *et al.*, "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip," *IEEE J. Sel. Areas Commun.*, vol. 9, no. 8, Oct. 1991.

[21] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," in *Proc. SIGCOMM*, 1995.

[22] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proceedings of the IEEE*, vol. 83, no. 10, Oct. 1995.

[23] B. Akesson *et al.*, "Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration," in *Proc. RTCSA*, Aug. 2008.

[24] J. Vink *et al.*, "Performance analysis of SoC architectures based on latency-rate servers," *Proc. DATE*, 2008.

[25] A. Hansson and K. Goossens, "Trade-offs in the configuration of a network on chip for multiple use-cases," in *Proc. Int'l Symposium on Networks on Chip (NOCS)*, 2007.

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

[27] ——, "Efficient Service Allocation in Hardware Using Credit-Controlled Static-Priority Arbitration," in *Proc. RTCSA*, 2009.

[28] K. Goossens *et al.*, "The Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Des. Test. Comput.*, vol. 22, no. 5, Sep. 2005.

Authorized licensed use limited to: Eindhoven University of Technology. Downloaded on April 09,2010 at 10:49:11 UTC from IEEE Xplore. Restrictions apply.