Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration

Benny Akesson[1], Liesbeth Steffens[2], Eelke Strooisma[3], and Kees Goossens[2,3]
[1]Technische Universiteit Eindhoven, The Netherlands
[2]NXP Semiconductors Research, Eindhoven, The Netherlands
[3]Delft University of Technology, The Netherlands
k.b.akesson@tue.nl

***Abstract*—The convergence of application domains in new systems-on-chip (SoC) results in systems with many applications with a mix of soft and hard real-time requirements. To reduce cost, resources, such as memories and interconnect, are shared between applications. However, resource sharing introduces interference between the sharing applications, making it difficult to satisfy their real-time requirements. Existing arbiters do not efficiently satisfy the requirements of applications in SoCs, as they either couple rate or allocation granularity to latency, or cannot run at sufficiently high speeds in hardware with a low-cost implementation.**

The contribution of this paper is an arbiter called Credit-Controlled Static-Priority (CCSP), consisting of a rate regulator and a static-priority scheduler. The rate regulator isolates applications by regulating the amount of provided service in a way that decouples allocation granularity and latency. The static-priority scheduler decouples latency and rate, such that low latency can be provided to any application, regardless of the allocated rate. We show that CCSP belongs to the class of latency-rate servers and guarantees the allocated service rate within a maximum latency, required by hard real-time applications. We present a hardware implementation of the arbiter in the context of a DDR2 SDRAM controller. An instance with six ports runs at 250 MHz and requires 0.0175 mm^2 in a 90 nm CMOS process.

I. Introduction

A contemporary multi-processor system-on-chip (SoC) consists of a large number of intellectual property components (IP), such as streaming hardware accelerators and processors with caches, running many applications. Resources, such as memories and interconnect, are shared between applications to reduce system cost. However, resource sharing introduces interference between applications, making it difficult to satisfy their real-time requirements. We refer to users of the resources as *requestors*, corresponding to processes in the context of CPUs, or communication channels in case of a memory or an interconnect, that act on behalf of an application. Resource access is provided by arbiters that require a low-cost hardware implementation and run at high speeds. A low-cost implementation allows multiple instances to fit in a limited area and high speed is required to schedule access on a fine level of granularity, reducing latency and buffers.

We consider resource scheduling in *hybrid systems* [1] that contain applications with both soft and hard real-time requirements. Hard real-time applications, such as audio post-processing, typically have predictable and regular request patterns. Their deadlines are not very tight, but must always be met in order to guarantee the functional correctness of the SoC [1], [2]. To satisfy these requirements, hard real-time requestors require a *guaranteed minimum service rate and a bounded maximum latency* that can be analytically verified at design time. In contrast, a soft real-time application, such as software video decoding, is typically very bursty and has tight deadlines on a much coarser grain than their hard real-time counterparts. These deadlines may span thousands of requests, making the worst-case latency of a single request less interesting. Missing a soft deadline reduces the quality of the application output, such as causing a frame skip in video playback, which may be acceptable as long as it does not occur too frequently [1]. Soft real-time requestors require a *guaranteed minimum service rate and a low average latency* to minimize deadline misses.

Existing arbiters fail to cater to the above-mentioned requirements for at least one of the following three reasons: 1) allocation granularity is coupled to latency, resulting in long latencies or over-allocation due to discretization, 2) latency is coupled to rate, preventing low latency from being provided to requestors with low rate requirements without over-allocating, or 3) they cannot run at sufficiently high speeds in hardware with a low-cost implementation.

The contribution of this paper is a novel arbiter called Credit-Controlled Static-Priority (CCSP), consisting of a rate regulator and a static-priority scheduler. The rate regulator isolates requestors by regulating the amount of provided service in a way that decouples allocation granularity and latency. The static-priority scheduler decouples latency and rate, such that low latency can be provided to any requestor, regardless of the allocated rate.

This paper is organized as follows. In Section II, we review related work and discuss why existing arbiters do not satisfy the requirements of hybrid systems in SoCs. We introduce a formal model in Section III and show how service curves are used to describe the interaction between requestors and the arbiter. We introduce the CCSP arbiter in Section IV and explain the operation of the rate regulator and static-priority scheduler. In Section V, we show that CCSP belongs to the class of latency-rate ($\mathcal{LR}$) servers and provides a minimum amount of service within a maximum latency, required by hard real-time requestors. An efficient hardware implementation is presented in Section VI in the context of a DDR2 SDRAM controller. We study experimental results for a hybrid system running an H.264 decoder in Section VII, before finishing with

conclusions in Section VIII.

II. Related work

Many arbiters have been proposed in the context of communication networks. Several of these are based on the Round-Robin algorithm because it is simple and starvation-free. Weighted Round-Robin [3] and Deficit Round-Robin [4] are extensions that guarantee each requestor a minimum service, proportional to an allocated rate, in a frame of fixed size. This type of *frame-based* arbitration suffers from a coupling between allocation granularity and latency, where the allocation granularity is inversely proportional to the frame size [5]. Larger frame sizes result in finer allocation granularities, reducing over-allocation, at the cost of increased latencies for all requestors. This granularity issue is addressed in [6]–[8] with hierarchical framing strategies and in [9], where tracking debits and credits accomplishes exact allocation over multiple frames. The above-mentioned algorithms, as well as the family of Fair Queuing algorithms [5], are unable to efficiently distinguish different latency requirements, as the rate is the only parameter affecting scheduling. This results in an unwanted coupling between latency and rate, where latency is inversely proportional to the allocated rate. Requestors with low rate requirements hence suffer from high latency unless their rates are increased, reducing resource utilization.

Much work has been carried out in the real-time community concerning server-based scheduling of aperiodic and sporadic requestors [10]. However, many of these assume that there is only a single server serving all aperiodic and sporadic requests and are hence unable to distinguish the requirements of multiple requestors. They furthermore often couple latency and rate since they use the server period and allocated budget as a base for computing latencies, similarly to most frame-based arbiters. Hard real-time requestors are scheduled in [1] by an earliest-deadline-first (EDF) scheduler, while a constant-bandwidth server is used for soft real-time requestors. This approach is not applicable to resource arbiters in SoCs, as it is difficult to provide a low-cost hardware implementation of an EDF scheduler that runs at sufficiently high speeds. The implementation of an EDF scheduler in [11] uses a tree of multiple-bit comparators to compare deadlines in the priority queue, which is too slow for SoC resources, such as memories and interconnect. A more scalable implementation is provided in [12]. However, this implementation requires double-ported SRAM memory, which is too expensive for many arbiters in a SoC. Among the work in this community, our work is most similar to a sporadic server with incremental replenishment [10], [13], although with an accounting mechanism that is simple enough to implement in hardware.

The scheduling approaches in [14], [15] employ static-priority arbiters, where high priority is assigned to soft real-time requestors to achieve low average latency. The approach of scheduling hybrid systems using a static-priority arbiter has the benefit of being cheap to implement in hardware. However, the proposed arbiters have significant short-comings, as the rate regulators are frame-based and couples allocation granularity, latency and rate, despite the use of priorities. Rate-Controlled Static-Priority [16] is an arbiter with a static-priority scheduler that decouples latency, rate and allocation granularity. It controls rate by holding requests until certain constraints on minimum and average inter-arrival times between requests from a requestor are satisfied. However, this requires a potentially large number of time-stamps to be stored in the arbiter, which is not feasible for a resource arbiter in a SoC.

We propose Credit-Controlled Static-Priority arbitration for scheduling access to SoC resources. CCSP resembles an arbiter with a rate regulator that enforces a (σ, ρ) constraint [17] on requested service together with a static-priority scheduler, a combination we refer to as Sigma-Rho Static-Priority (SRSP) in this paper. Similarly to SRSP, the CCSP rate regulator replenishes the service available to a requestor incrementally, instead of basing it on frames, decoupling allocation granularity and latency. Both arbiters furthermore use priorities to decouple latency and rate. However, instead of enforcing a (σ, ρ) constraint on *requested service*, like SRSP, CCSP enforces it on *provided service*. Regulating provided service reduces the complexity of the implementation, and allows a preemptive arbiter to efficiently handle requests with unknown sizes. We furthermore show that CCSP has a low-cost hardware implementation that runs at high speeds.

III. Formal model

In this section, we introduce the formal model used in this paper. We explain how service curves are used to model the interaction between the requestors and the resource in Section III-A. We proceed by discussing the models used to bound *requested service* and *provided service* in Section III-B and Section III-C, respectively.

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 emphasize the generality of our approach, and its applicability to a wide range of resources, we abstract from a particular target resource, such as memories or (multi-hop) interconnects. 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 *cycle*, is defined as the time required to serve such a service unit. We use closed discrete time intervals and $[\tau, t]$ hence includes all cycles in the sequence $\langle \tau, \tau + 1, ..., t - 1, t \rangle$.

A. Service curves

We use service curves [18] to model the interaction between the resource and the requestors. These service curves are typically cumulative and monotonically non-decreasing in time. We start by defining two operators for working with service curves in Definition 1 and Definition 2.

Definition 1. *$\xi(t)$ denotes the value of a service curve ξ at the beginning of a cycle t.*

Definition 2. *$\xi(\tau, t)$ denotes the difference in values between the endpoints of the closed interval $[\tau, t]$, where $t \geq \tau$, and is defined as $\xi(\tau, t) = \xi(t+1) - \xi(\tau)$.*

The resource is shared between a set of requestors, as stated in Definition 3. A requestor generates requests of variable but bounded size, as defined in Definition 4.

Definition 3 (Set of requestors)**.** *The set of requestors sharing the resource is denoted R.*

Definition 4 (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 $s(\omega_r^k) : \Omega_r \to \mathbb{N}^+$.*

Requests arrive in separate buffers per requestor at the resource according to Definition 5. For clarity, it is assumed that only a single request arrives per requestor in a particular cycle, although this is easy to generalize. A request is considered to arrive as an impulse when it has completely arrived, which for instance in the case of a memory controller is upon arrival of the last bit of the request. This is captured by the requested service curve, w, defined in Definition 6. Note that Definitions 5 and 6 state that a requested service curve at time $t+1$ accounts for a request with arrival time $t+1$.

Definition 5 (Arrival time)**.** *The arrival time of a request ω_r^k from a requestor $r \in R$ is denoted $t_a(\omega_r^k) : \Omega_r \to \mathbb{N}^+$, and corresponds to the cycle in which ω_r^k has completely arrived.*

Definition 6 (Requested service curve)**.** *The requested service curve of a requestor $r \in R$ is denoted $w_r(t) : \mathbb{N} \to \mathbb{N}$, where $w_r(0) = 0$ and*

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

The scheduler in the resource arbiter attempts to schedule a requestor every cycle according to its particular scheduling policy, according to Definition 7. The first cycle in which a request ω^k is scheduled is referred to as its starting time, $t_s(\omega^k)$, defined in Definition 8.

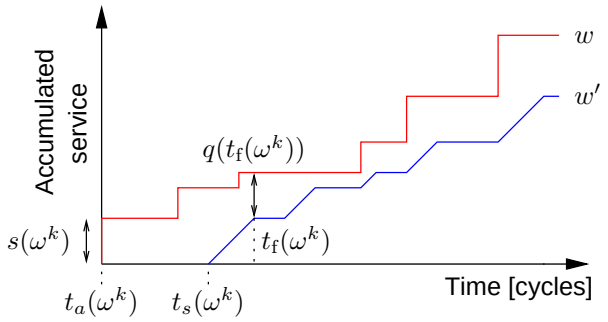

Fig. 1. A requested service curve, a provided service curve and representations of the surrounding concepts.

Definition 7 (Scheduled requestor)**.** *The scheduled requestor at a time t is denoted $\gamma(t) : \mathbb{N} \to R \cup \{\varnothing\}$.*

Definition 8 (Starting time of a request)**.** *The starting time of a request ω_r^k is denoted $t_s(\omega_r^k) : \Omega_r \to \mathbb{N}$, and is defined as the smallest t at which ω_r^k is scheduled.*

The provided service curve, w', defined in Definition 9, reflects the amount of service units provided by the resource to a requestor. A service unit takes one cycle to serve. The provided service is hence increased at $t+1$, if a requestor is scheduled at t. A request leaves the resource when the last service unit of the request has been served, corresponding to when the last bit is read or written in case of a memory controller. An illustration of a requested service curve and a provided service curve is provided in Figure 1.

Definition 9 (Provided service curve)**.** *The provided service curve of a requestor $r \in R$ is denoted $w'_r(t) : \mathbb{N} \to \mathbb{N}$, where $w'_r(0) = 0$ and*

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

The finishing time of a request corresponds to the first cycle in which a request is completely served, as defined in Definition 10. The amount of requested service that has not been served at a particular time is referred to as the backlog of a requestor and is defined in Definition 11.

Definition 10 (Finishing time of a request)**.** *The finishing time of a request ω_r^k is denoted $t_f(\omega_r^k) : \Omega_r \to \mathbb{N}$, and is defined as $t_f(\omega_r^k) = \min(\{t \mid t \in \mathbb{N} \wedge w'_r(t) = w'_r(t_s(\omega_r^k)) + s(\omega_r^k)\})$.*

Definition 11 (Backlog)**.** *The backlog of a requestor $r \in R$ at a time t is denoted $q_r(t) : \mathbb{N} \to \mathbb{N}$, and is defined as $q_r(t) = w_r(t) - w'_r(t)$.*

Definition 12 (Set of backlogged requestors)**.** *The set of requestors that are backlogged at t is defined as $R_t^q = \{r \mid \forall r \in R \wedge q_r(t) > 0\}$.*

To work with service curves analytically, traffic models are used to characterize the behavior of the curves. This abstraction has the benefit that analytical results can be derived without exact knowledge of a service curve [5]. Characterizations that bound the requested and provided service curves are required to provide an upper bound on latency, which is needed to satisfy the requirements of hard real-time requestors.

B. Requested service model

We use the (σ, ρ) model [17] to characterize the requested service curve. The model uses a linear function to express a burstiness constraint, and is frequently used to upper bound the requested service curve in an interval. The bounding function is determined by two parameters, σ and ρ, corresponding to burstiness and average request rate, respectively. Definition 13 defines a (σ, ρ)-constrained service curve, and its graphical interpretation is shown in Figure 2.

Definition 13 ((σ, ρ) constraint)**.** *A service curve, ξ, is defined to be (σ, ρ) constrained in an interval $[\tau, t]$ if $\hat{\xi}(\tau, t) = \sigma + \rho \cdot (t - \tau + 1)$. $\sigma, \rho \in \mathbb{R}^+$ and $\rho \leq 1$.*

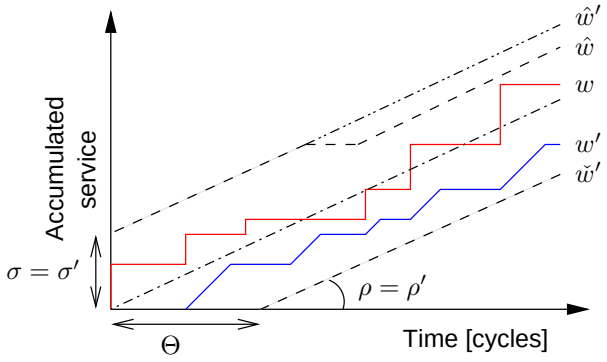

Fig. 2. A requested service curve and a provided service curve along with their corresponding bounds.

Hard real-time requestors typically correspond to hardware IPs with regular and predictable access patterns that lend themselves to characterization. Soft real-time requestors, however, are typically burstier than their hard real-time counterparts, and may hence have a σ that is very large. Soft real-time requestors may additionally be very difficult to characterize, as applications become more dynamic and input dependent. However, in this paper, we assume that all requestors have been accurately characterized, according to Definition 14.

Definition 14 (Requestor)**.** *A requestor $r \in R$ is characterized by (σ_r, ρ_r), which is a (σ, ρ) constraint on w_r.*

C. Provided service model

The purpose of the provided service model is to give a lower bound on the provided service curve based on the service allocation of a requestor. The service allocated to a requestor in our model depends on two parameters, as defined in Definition 15. These are the allocated service rate, ρ', and allocated burstiness, σ', respectively. The definition states three constraints that must be satisfied in order for a configuration to be valid: 1) the allocated service rate must be at least equal to the average request rate, ρ, to satisfy the service requirement of the requestor, and to maintain finite buffers, 2) it is not possible to allocate more service to the requestors than what is offered by the resource, and 3) the allocated burstiness must be sufficiently large to accommodate a service unit. The last condition is required for the latency bound derived in Section V to be valid.

Definition 15 (Allocated service)**.** *The service allocation of a requestor $r \in R$ is defined as $(\sigma'_r, \rho'_r) \in \mathbb{R}^+ \times \mathbb{R}^+$. For a valid allocation it holds that $\forall r \in R : \rho'_r \geq \rho_r$, $\sum_{\forall r \in R} \rho'_r \leq 1$, and $\forall r \in R : \sigma'_r \geq 1$.*

Our provided service model is based on the notion of *active periods*. Definition 16 states that a requestor is active at t if it is either live at t (Definition 17), backlogged at t, or both. Definition 17 states that a requestor must on average have requested service according to its allocated rate since the start of the latest active period to be considered live at a time t.

Definition 16 (Active period)**.** *An active period of a requestor $r \in R$ is defined as the maximum interval $[\tau_1, \tau_2]$, such that $\forall t \in [\tau_1, \tau_2] : w_r(\tau_1 - 1, t-1) \geq \rho'_r \cdot (t - \tau_1 + 1) \vee q_r(t) > 0$. Requestor r is active $\forall t \in [\tau_1, \tau_2]$.*

Definition 17 (Live requestor)**.** *A requestor $r \in R$ is defined as live at a time t during an active period $[\tau_1, \tau_2]$ if $w_r(\tau_1 - 1, t-1) \geq \rho'_r \cdot (t - \tau_1 + 1)$.*

Definition 18 (Set of active requestors)**.** *The set of requestors that are active at t is defined as $R^a_t = \{r \mid \forall r \in R \wedge r \text{ active at } t\}$.*

Definition 19 (Set of live requestors)**.** *The set of requestors that are live at t is defined as $R^l_t = \{r \mid \forall r \in R \ \wedge \ r \text{ live at } t\}$.*

Figure 3 illustrates the relation between being live, backlogged and active. Three requests arrive starting from τ_1, keeping the requestor live until τ_3. The requestor is initially both live and backlogged, but the provided service curve catches up with the requested service curve at τ_2. This puts the requestor in a live and not backlogged state until τ_3. The requestor is neither live nor backlogged between τ_3 and τ_4, as no additional requests arrive at the resource. The requestor becomes live and backlogged again at τ_4, since two additional requests arrive within a small period of time. The requestor stays in this state until τ_5, since not enough service is provided to remove the backlog. The requestor is hence backlogged but not live at τ_5, and remains such until τ_6. The requestor in Figure 3 is active between τ_1 and τ_3 and between τ_4 and τ_6, according to Definition 16. Note from this example that a live requestor is not necessarily backlogged, nor vice versa.

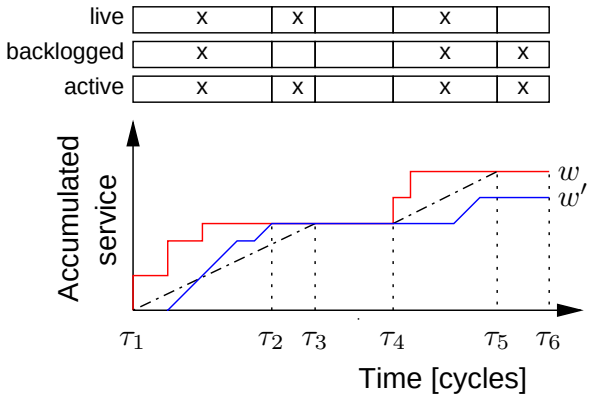

Fig. 3. Example service curves illustrating the relation between being live, backlogged, and active.

The service provided to a requestor is defined by two parameters Θ and ρ', being latency and allocated rate, respectively. To disambiguate, we refer to Θ, defined in Definition 20, as *service latency* throughout this paper. The definition states that service is provided to an active requestor according to the allocated rate, ρ', after the service latency, Θ. This means that ρ' and Θ define a lower bound, $\check{w}'$, on the provided service curve during an active period, as shown in Figure 2.

Definition 20 (Service latency)**.** *The service latency of a requestor $r \in R$ is defined as the minimum $\Theta_r \in \mathbb{N}$, such that during any active period $[\tau_1, \tau_2]$ it holds that $\forall t \in [\tau_1, \tau_2]$:* $\check{w}'_r(\tau_1, t) = \max(0, \rho'_r \cdot (t - \tau_1 + 1 - \Theta_r))$.

We show in Section V that CCSP belongs to the class of $\mathcal{LR}$ servers [19], which is a general frame-work for analyzing scheduling algorithms. The lower bound on provided service in Definition 20 is a key characteristic of $\mathcal{LR}$ servers. The authors of [19] use this bound to derive general bounds on buffering and latency that are valid for any combination of $\mathcal{LR}$ servers in sequence. It is furthermore shown in [20] that a $\mathcal{LR}$ server can be modeled as a cyclo-static data-flow graph with two tasks. This allows $\mathcal{LR}$ servers to be used also in data-flow analysis, which has the added benefits that the presence of flow control can be accurately modeled and that application-level throughput constraints can be satisfied.

IV. Credit-Controlled Static-Priority

A CCSP arbiter consists of a rate regulator and a scheduler, following the decomposition from [16]. We start in Section IV-A by providing an overview of the main idea, before discussing the rate regulator and scheduler separately in Sections IV-B and IV-C, respectively.

A. Overview

A rate regulator provides *accounting* and *enforcement* and thus determines which requests that are *eligible* for scheduling at a particular time, considering their allocated service. There are two types of enforcement. A *work-conserving arbiter* is never idle when there is a backlogged requestor. In contrast, a rate regulator in a *non-work-conserving arbiter* does not schedule a request until it becomes eligible, even though the resource may be idle. To conserve space, we only discuss the non-work-conserving case in this paper. The work-conserving case is covered in [21].

The purpose of a rate regulator is to isolate requestors from each other and to protect requestors that do not ask for more service than they are allocated from those that do. This form of protection is a key property in providing guaranteed service to requestors with timing constraints [5]. A rate regulator protects requestors by enforcing burstiness constraints on either requested service or provided service.

A rate regulator that enforces an upper bound on provided service, such as those in [3], [4], [14], [15] and the CCSP rate regulator, is shown in Figure 4. As seen in the figure, the rate regulator is positioned after the request buffers. It is hence only aware of requests at the heads of the buffers, and cannot constrain arrival of requests in any way. The scheduler communicates the id of the scheduled requestor, $\gamma(t)$, back to the rate regulator every cycle. The regulator uses this information to update the accounting mechanism. This type of rate regulator operates by simply determining if the request at the head of each request buffer is eligible for scheduling.

Enforcing an upper bound on provided service as opposed to requested service has two benefits: 1) the implementation of the regulator is less complex, and 2) the amount of work

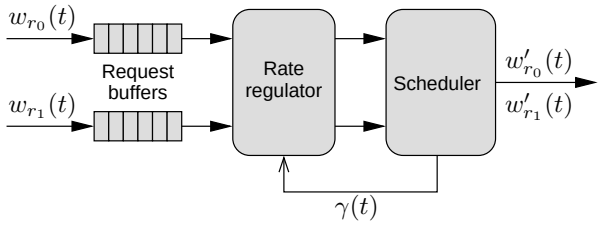

Fig. 4. An arbiter with a rate regulator that enforces an upper bound on provided service.

associated with a particular request does not have to be known. We discuss these benefits in more detail.

A regulator that enforces an upper bound on provided service only requires knowledge about the request at the head of each request queue. Conversely, regulators that enforce an upper bound on requested service, such as [16], [17], need information about all requests that arrive during a cycle. This incurs additional complexity in a hardware implementation, especially if requests can arrive with higher frequency than with which they are parsed.

A difficulty in arbitration is that the amount of work associated with a particular request is not always known before it has been served. For instance, the amount of time required to decode a video frame on a processor is not known when the work is scheduled. This situation cannot be handled if requested service is regulated, unless worst-case assumptions are used to estimate the amount of work, which is very inefficient if the variance in the amount of work is large. This is efficiently handled when regulating provided service by charging for a single service unit at a time. This allows a preemptive scheduler to interrupt a requestor that runs out of budget and schedule another one.

Unlike SRSP, CCSP enjoys the aforementioned benefits. CCSP's incremental replenishment of service furthermore decouples allocation granularity and latency, in contrast to the frame-based provided service regulators in [3], [4], [14], [15].

B. Rate regulator

The CCSP rate regulator enforces an upper bound on provided service, as explained in Section IV-A. We regulate provided service based on active periods, and define the upper bound on provided service according to Definition 21. The intuition behind the definition is that the upper bound on provided service of an active requestor increases according to the allocated rate every cycle. Conversely, for an inactive requestor, the bound is limited to $w'(t) + \sigma'$, a value that depends on the allocated burstiness. This prevents that a requestor that has been inactive for an extended period of time increases its bound, possibly resulting in starvation of other requestors once it becomes active again. Note that this implies that the upper bound on provided service is not necessarily monotonically non-decreasing in time, as shown in Figure 5. The requestor in the figure is live until τ_1, but remains active until τ_2 where w' catches up to w. According to Definition 21, this results in $\hat{w}'(\tau_2+1) < \hat{w}'(\tau_2)$, since $\hat{w}'(\tau_2) > w'(\tau_2)+\sigma'$.

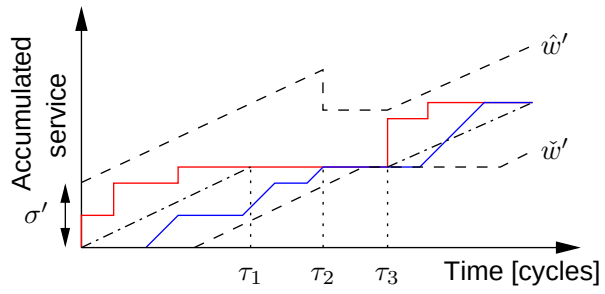

Fig. 5. The upper bound on provided service is not necessarily monotonically non-decreasing.

The requestor starts a new active period at τ_3, causing $\hat{w}'$ to increase again.

Definition 21 (Provided service bound)**.** *The enforced upper bound on provided service of a requestor $r \in R$ is denoted $\hat{w}'_r(t) : \mathbb{N} \rightarrow \mathbb{R}^+$, where $\hat{w}'_r(0) = \sigma'_r$ and*

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

It is not possible to perform accounting and enforcement in hardware based on $\hat{w}'$, since $\lim_{t\rightarrow\infty} \hat{w}'(t) = \infty$, resulting in overflow of finite counters. Instead, the accounting mechanism in the rate regulator is based on the potential of a requestor, as defined in Definition 22. The potential of a requestor is bounded since the arbiter guarantees a lower bound on provided service, as we will show in Section V. The accounting used by the CCSP rate regulator is defined according to Definition 23. It is shown in [21] that the accounting mechanism in Definition 23 corresponds to a recursive definition of potential, and hence that $\forall t \in \mathbb{N} : \ \pi(t) = \pi^*(t)$.

Definition 22 (Potential)**.** *The potential of a requestor $r \in R$ is denoted $\pi_r(t) : \mathbb{N} \rightarrow \mathbb{R}$, and is defined as $\pi_r(t) = \hat{w}'_r(t) - w'_r(t)$.*

Definition 23 (Accounting)**.** *The accounted potential of a requestor $r \in R$ is denoted $\pi^*_r(t) : \mathbb{N} \rightarrow \mathbb{R}$, where $\pi^*_r(0) = \sigma'_r$ and*

$$\pi^*_r(t+1) = \begin{cases} \pi^*_r(t) + \rho'_r - 1 & r \in R^a_t \wedge \gamma(t) = r \\ \pi^*_r(t) + \rho'_r & r \in R^a_t \wedge \gamma(t) \neq r \\ \sigma'_r & r \notin R^a_t \wedge \gamma(t) \neq r \end{cases} \quad (2)$$

Enforcement in the rate regulator takes place before the accounting is updated, and is performed by determining if a request from a requestor is eligible for scheduling. A request becomes eligible at its eligibility time. Definition 24 states three conditions that must be satisfied for a request at this time: 1) all previous requests from the requestor must have been served, 2) the requestor must be backlogged, and 3) the requestor must have at least enough potential to serve one service unit, including the service earned when the accounting is updated. The eligibility criterion for a requestor is formally defined in Definition 25.

Definition 24 (Eligibility time)**.** *The eligibility time of a request ω^k_r from a requestor $r \in R$ is denoted $t_e(\omega^k_r)$, and is defined as the smallest t at which: 1) $\forall i < k : \ t \geq t_f(\omega^i_r)$, and 2) $w_r(t) > w'_r(t)$, and 3) $\pi_r(t) \geq 1 - \rho'_r(t)$.*

Definition 25 (Eligible requestor)**.** *Requestor r is defined as eligible at t if $\exists k \in \mathbb{N} : \ t \in [t_e(\omega^k_r), t_f(\omega^k_r) - 1] \ \wedge \ \pi_r(t) \geq 1 - \rho'_r(t) \wedge w_r(t) > w'_r(t)$.*

Definition 26 (Set of eligible requestors)**.** *The set of requestors that are eligible for scheduling at t is defined as $R^e_t = \{r \mid \forall r \in R \wedge r \text{ eligible at } t\}$.*

C. Scheduler

The CCSP arbiter uses a static-priority scheduler, as it decouples latency and rate and is cheap to implement in hardware. Each requestor is assigned a priority level, p, as stated in Definition 27, where a lower level indicates higher priority. We do not allow requestors to share priority levels. Sharing priorities, as done in [16], results in a situation where equal priority requestors must assume that they all have to wait for each other in the worst-case, resulting in less tight bounds. In this paper, we consider a scheduler that is preemptive on the granularity of a single service unit. A preemptive non-work-conserving static-priority scheduler schedules the highest priority eligible requestor, as defined in Definition 29. The case of a non-preemptive scheduler is covered in [21].

Definition 27 (Priority level)**.** *A requestor $r \in R$ has a priority level p_r, such that $\forall r_i, r_j \in R, r_i \neq r_j : p_{r_i} \neq p_{r_j}$.*

Definition 28 (Set of higher priority requestors)**.** *The set of requestors with higher priority than $r_i \in R$ is defined as $R^+_{r_i} = \{r_j \mid \forall r_j \in R \wedge p_{r_i} > p_{r_j}\}$.*

Definition 29 (Static-priority scheduler)**.** *The scheduled requestor at a time t in a preemptive non-work-conserving static-priority scheduler is defined as*

$$\gamma(t) = \begin{cases} r_i \text{ s.t. } r_i \in R^e_t \wedge \nexists r_j \in R^e_t : \ p_{r_j} < p_{r_i} & R^e_t \neq \varnothing \\ \varnothing & R^e_t = \varnothing \end{cases}$$

V. Arbiter Analysis

In this section, we derive analytical properties of the CCSP arbiter. First, we define and upper bound the interference experienced by a requestor during an interval. We then use this bound to derive the service guarantee of CCSP, and to prove that it belongs to the class of $\mathcal{LR}$ servers. Lastly, we upper bound the finishing time of a request, based on the derived service guarantee.

Definition 30 states that the interference experienced by a requestor in an interval consists of two parts. The first part is concerned with the potential of higher priority requestors at the start of the interval and the second with the increase of their provided service bounds during the interval. Together, these parts determine how much an interfering requestor can maximally be scheduled before being throttled by the rate regulator.

Definition 30 (Interference)**.** *The interference from higher priority requestors experienced by a requestor $r \in R$ during an interval $[\tau_1, \tau_2]$ is denoted $i_r(\tau_1, \tau_2) : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$, and is defined as*

$$i_r(\tau_1, \tau_2) = \sum_{\forall r_j \in R_{r_i}^+} (\pi_{r_j}(\tau_1) + \hat{w}'_{r_j}(\tau_1, \tau_2)) \quad (3)$$

To compute the upper bound on interference, we will bound the two parts of Equation (3) separately. First, we introduce two lemmas proven in [21]. Lemma 1 shows some important relations between the requested service curve and the provided service curve at the start of an active period and Lemma 2 establishes a relation between potential and eligibility for active requestors. We then proceed in Lemma 3 by bounding the increase in the upper bound on provided service during an interval, corresponding to the second part of Equation (3).

Lemma 1. *If τ_1 is the start of an active period then* $w(\tau_1) > w(\tau_1 - 1) = w'(\tau_1) = w'(\tau_1 - 1)$.

Lemma 2. $\forall r \in R_t^a : \pi_r(t) > \sigma'_r - \rho'_r \Rightarrow r \in R_t^e$.

Lemma 3. $\hat{w}'_r(\tau, t) \leq \rho' \cdot (t - \tau + 1)$.

Proof: We prove the lemma by showing that the inequality holds when $\hat{w}'_r(\tau, t)$ is maximal. This occurs when $\tau, t \in [\tau_1, \tau_2]$, where $[\tau_1, \tau_2]$ is an active period. This in turn is proved by showing that the first rule of Equation (1) implies $\hat{w}'_r(t+1) > \hat{w}'_r(t)$, while the second rule implies $\hat{w}'_r(t+1) \leq \hat{w}'_r(t)$.

The first rule in Equation (1) implies that $\hat{w}'_r(t+1) > \hat{w}'_r(t)$, since it follows from Definition 13 and Definition 15 that $\rho'_r \geq 0$.

We split the analysis of the second rule in Equation (1) into two cases. In the first case, the requestor is inactive at both $t-1$ and t, corresponding to multiple cycles of inactivity. In the second case, the requestor is active at $t-1$ and inactive at t, meaning it is ending its active period.

Case 1: $r \notin R_{t-1}^a \wedge r \notin R_t^a$
From the second rule in Equation (1), we get that $\hat{w}'_r(t+1) = w'_r(t) + \sigma'_r$. Since an inactive requestor cannot be scheduled, it must hold that $w'_r(t) = w'_r(t-1)$. It hence follows that $\hat{w}'_r(t+1) = \hat{w}'_r(t)$ if $r \notin R_{t-1}^a \wedge r \notin R_t^a$.

Case 2: $r \in R_{t-1}^a \wedge r \notin R_t^a$
We proceed by showing that this case implies $\hat{w}'_r(t+1) < \hat{w}'_r(t)$. Let $t = \tau_2 + 1$, where $[\tau_1, \tau_2]$ defines an active period. We must hence show that

$$\hat{w}'_r(\tau_2 + 2) < \hat{w}'_r(\tau_2 + 1) \quad (4)$$

According to Definition 2, $\hat{w}'_r(\tau_2+1) = \hat{w}'_r(\tau_1) + \hat{w}'_r(\tau_1, \tau_2)$. From Lemma 1 and the second rule in Equation (1), we get that $\hat{w}'_r(\tau_1) = w'_r(\tau_1 - 1) + \sigma'_r = w'_r(\tau_1) + \sigma'_r$, since $r \notin R_{\tau_1 - 1}^a$. We furthermore know from the first rule in Equation (1) that $\hat{w}'_r(\tau_1, \tau_2) = \rho'_r \cdot (\tau_2 - \tau_1 + 1)$, since $\forall t \in [\tau_1, \tau_2] : r \in R_t^a$. This results in

$$\hat{w}'_r(\tau_2 + 1) = w'_r(\tau_1) + \sigma'_r + \rho'_r \cdot (\tau_2 - \tau_1 + 1) \quad (5)$$

The second rule in Equation (1) states that $\hat{w}'_r(\tau_2 + 2) = w'_r(\tau_2 + 1) + \sigma'$ since $r \notin R_{\tau_2+1}^a$. Rewriting this using Definition 2 results in $\hat{w}'_r(\tau_2+2) = w'_r(\tau_1) + w'_r(\tau_1, \tau_2) + \sigma'_r$. From Definition 16 and Lemma 1, we know that $r \notin R_{\tau_2+1}^a \Rightarrow w'_r(\tau_1 - 1, \tau_2) = w_r(\tau_1 - 1, \tau_2) < \rho'_r \cdot (\tau_2 - \tau_1 + 1)$, as the requestor is neither live nor backlogged at $\tau_2 + 1$. Putting these results together gives us

$$\hat{w}'_r(\tau_2 + 2) < w'_r(\tau_1) + \sigma'_r + \rho'_r \cdot (\tau_2 - \tau_1 + 1) \quad (6)$$

By substituting Equation (5) and Equation (6) into Equation (4), we see that $\hat{w}'_r(\tau_2 + 2) < \hat{w}'_r(\tau_2 + 1)$.

We hence conclude that $\hat{w}'_r(\tau, t)$ is maximal when $\tau, t \in [\tau_1, \tau_2]$, where $[\tau_1, \tau_2]$ is an active period. According to Definition 22 and the first rule of Equation (2), this implies that $\hat{w}'_r(\tau, t) \leq \rho' \cdot (t - \tau + 1)$. □

We define the concept of aggregate potential of a set of requestors in Definition 31 and show in Lemma 4 that it cannot increase, as long as a requestor in the set is scheduled every cycle. This is a key result that bounds the first part of Equation (3) in Lemma 5 and leads to an upper bound on interference in Lemma 6.

Definition 31 (Aggregate potential)**.** *The aggregate potential of a set of requestors $R' \subseteq R$ is defined according to* $\sum_{\forall r \in R'} \pi_r(t) = \sum_{\forall r \in R'} \hat{w}'_r(t) - \sum_{\forall r \in R'} w'_r(t)$.

Lemma 4. *For a set of requestors $R' \subseteq R$, it holds that* $\forall t \in \mathbb{N} : \ (\exists r_k \in R' : \ \gamma(t) = r_k) \Rightarrow \sum_{\forall r \in R'} \pi_r(t+1) \leq \sum_{\forall r \in R'} \pi_r(t)$.

Proof: According to Definition 2 and the definition of aggregate potential in Definition 31

$$\sum_{\forall r \in R'} \pi_r(t+1) = \sum_{\forall r \in R'} \pi_r(t) + \sum_{\forall r \in R'} \hat{w}'_r(t, t) - \sum_{\forall r \in R'} w'_r(t, t)$$

According to Lemma 3, $\sum_{\forall r \in R'} \hat{w}'(t, t) \leq \sum_{\forall r \in R'} \rho'_r$, where equality is reached if all requestors are active at t. From Definition 9, we also get that $\sum_{\forall r \in R'} w'_r(t, t) = 1$ if a requestor in R' is scheduled at t. Hence, if $\forall r \in R' : \ r \in R_t^a$ and $\exists r_k \in R' : \ \gamma(t) = r_k$, then

$$\sum_{\forall r \in R'} \pi_r(t+1) \leq \sum_{\forall r \in R'} \pi_r(t) + \sum_{\forall r \in R'} \rho'_r - 1$$

Finally, $\sum_{\forall r \in R'} \rho'_r \leq 1$, according to Definition 15, which concludes the proof. □

Lemma 5. *For a requestor $r_i \in R$, it holds that $\forall t \in \mathbb{N} :$ $\sum_{\forall r_j \in R_{r_i}^+} \pi_{r_j}(t) \leq \sum_{\forall r_j \in R_{r_i}^+} \sigma'_{r_j}$. The equality occurs at any time t for which $\forall r_j \in R_{r_i}^+ : \ r_j \notin R_{t-1}^a$.*

Proof: We prove the lemma by induction on t.
Base case: The lemma holds at $t = 0$, since Definition 23 states that $\forall r \in R : \ \pi_r(0) = \sigma'_r$.

Inductive step: At $t+1$, we examine two different cases for the premise at t. In the first case there exists a higher priority eligible requestor, and in the second case there does not.
Case 1: $(R_{r_i}^+ \cap R_t^e) \neq \varnothing$
Picking $r_k \in (R_{r_i}^+ \cap R_t^e)$, according to Definition 29

and applying Lemma 4 results in the first inequality in Equation (7). The second inequality follows from the induction hypothesis.

$$\sum_{\forall r_j \in R^+_{r_i}} \pi_{r_j}(t+1) \leq \sum_{\forall r_j \in R^+_{r_i}} \pi_{r_j}(t) \leq \sum_{\forall r_j \in R^+_{r_i}} \sigma'_{r_j} \quad (7)$$

Case 2: $(R^+_{r_i} \cap R^e_t) = \varnothing$
No higher priority requestor is eligible in this case. We will show that this implies that $\pi(t+1) \leq \sigma'$ both for requestors with $\pi(t) > \sigma' - \rho'$ and $\pi(t) \leq \sigma' - \rho'$.

According to Lemma 2, it must hold that $\forall r_j \in R^+_{r_i} \wedge r_j \notin R^e_t : \pi_{r_j}(t) > \sigma'_{r_j} - \rho'_{r_j} \Rightarrow r_j \notin R^a_t$. The third rule of Equation (2) hence states that $\forall r_j \in R^+_{r_i} : \pi_{r_j}(t) > \sigma'_{r_j} - \rho'_{r_j} \Rightarrow \pi_{r_j}(t+1) = \sigma'_{r_j}$. For the other case by Definition 23, $\forall r_j \in R^+_{r_i} : \pi_{r_j}(t) \leq \sigma'_{r_j} - \rho'_{r_j} \Rightarrow \pi_{r_j}(t+1) \leq \sigma'_{r_j}$. Hence, $\forall r_j \in R^+_{r_i} : \pi_{r_j}(t+1) \leq \sigma'_{r_j}$. This means that $\sum_{\forall r_j \in R^+_{r_i}} \pi_{r_j}(t+1) \leq \sum_{\forall r_j \in R^+_{r_i}} \sigma'_{r_j}$, which proves the second case.

The aggregate potential of higher priority requestors is maximal when $\forall r_j \in R^+_{r_i} : \pi_{r_j}(t) = \sigma'_{r_j}$, which occurs at any time t for which $\forall r_j \in R^+_{r_i} : r_j \notin R^a_{t-1}$. □

Lemma 6 (Maximum interference)**.** *The maximum interference from higher priority requestors experienced by a requestor $r_i \in R$ during an interval $[\tau_1, \tau_2]$ occurs when all higher priority requestors start an active period at τ_1 and remain active $\forall t \in [\tau_1, \tau_2]$, and equals*

$$\hat{i}_{r_i}(\tau_1, \tau_2) = \sum_{\forall r_j \in R^+_{r_i}} \sigma'_{r_j} + \rho'_{r_j} \cdot (\tau_2 - \tau_1 + 1) \quad (8)$$

Proof: We know from Equation (3) that interference is defined as $i_{r_i}(\tau_1, \tau_2) = \sum_{\forall r_j \in R^+_{r_i}} (\pi_{r_j}(\tau_1) + \hat{w}'_{r_j}(\tau_1, \tau_2))$. Lemma 5 states that $\sum_{\forall r_j \in R^+_{r_i}} \pi_{r_j}(\tau_1) \leq \sum_{\forall r_j \in R^+_{r_i}} \sigma'_{r_j}$, which is maximal when all higher priority requestors are inactive at $\tau_1 - 1$. We furthermore know from Lemma 3 that $\sum_{\forall r_j \in R^+_{r_i}} \hat{w}'_{r_j}(\tau_1, \tau_2) \leq \sum_{\forall r_j \in R^+_{r_i}} \rho'_{r_j} \cdot (\tau_2 - \tau_1 + 1)$, which is maximal when $\forall t \in [\tau_1, \tau_2] : r_j \in R^a_t$. Hence, $\hat{i}_{r_i}(\tau_1, \tau_2) = \sum_{\forall r_j \in R^+_{r_i}} \sigma'_{r_j} + \rho'_{r_j} \cdot (\tau_2 - \tau_1 + 1)$ when all higher priority requestors start an active period at τ_1, and remain active $\forall t \in [\tau_1, \tau_2]$. □

We continue in Theorem 1 by deriving the service guarantee of a CCSP arbiter, and to compute its service latency. We then prove in Theorem 2 that CCSP belongs to the class of $\mathcal{LR}$ servers. These theorems hold only for requestors that are eligible during backlogged periods, i.e. when $r \in R^q_t \Rightarrow r \in R^e_t$. This is accomplished by configuring $\rho' \geq \rho$, according to Definition 15, and letting $\sigma' \geq \sigma$. We configure $\sigma' = \sigma$ for hard real-time requestors, since there is no benefit in allocating higher burstiness than requested. Configuring $\sigma' < \sigma$ causes the regulator to limit the burstiness of a requestor, resulting in that the bound on service latency is increased. This is useful to protect hard real-time requestors from bursty soft real-time requestors that are not interested in bounds on service latency.

Theorem 1 (Service guarantee)**.** *An active requestor $r_i \in R$, for which $\sigma'_{r_i} \geq \sigma_{r_i}$, is guaranteed a minimum service during an active period $[\tau_1, \tau_2]$ according to $\forall t \in [\tau_1, \tau_2] : \check{w}'_{r_i}(\tau_1, t) = \max(0, \rho'_{r_i} \cdot (t - \tau_1 + 1 - \Theta_{r_i}))$, where*

$$\Theta_{r_i} = \frac{\sum_{\forall r_j \in R^+_{r_i}} \sigma'_{r_j}}{1 - \sum_{\forall r_j \in R^+_{r_i}} \rho'_{r_j}} \quad (9)$$

Proof: It suffices to show that the theorem holds for intervals where $\tau_2 - \tau_1 + 1 > \Theta_{r_i}$, as these are the only intervals for which $\check{w}'_{r_i}(\tau_1, \tau_2) > 0$. For these intervals, we must show that

$$\forall t \in [\tau_1, \tau_2] : \check{w}'_{r_i}(\tau_1, t) = \rho'_{r_i} \cdot (t - \tau_1 + 1 - \Theta_{r_i}) \quad (10)$$

We prove the theorem by splitting the active period in two cases. In the first case, we look at the behavior of r_i during backlogged periods within the active period, where the k:th backlogged period is denoted $[\alpha_k, \beta_k]$. It is assumed that $\forall t \in [\alpha_k, \beta_k] : r_i \in R^e_t$. In the second case, the requestor is in a live and not backlogged state.

Case 1: $\forall t \in [\alpha_k, \beta_k] : r_i \in R^q_t$
The requestor is eligible in the interval since $\sigma'_{r_i} \geq \sigma_{r_i}; \wedge r \in R^q_t \Rightarrow r \in R^e_t$. There are $(\beta_k - \alpha_k + 1)$ units of service available in the backlogged interval. An eligible requestor in a static-priority scheduler cannot access the resource whenever it is used by higher priority requestors. The minimum service available to r_i, denoted $\check{w}^a_{r_i}$, can hence be expressed according to $\check{w}^a_{r_i}(\alpha_k, \beta_k) = \beta_k - \alpha_k + 1 - \hat{i}_{r_i}(\alpha_k, \beta_k)$. Since r_i is continuously backlogged and eligible in the interval, it follows that $\check{w}'_{r_i}(\alpha_k, \beta_k) = \check{w}^a_{r_i}(\alpha_k, \beta_k)$. We proceed by using the result from Lemma 6 to bound the maximum possible interference.

$$\check{w}'_{r_i}(\alpha_k, \beta_k) = \beta_k - \alpha_k + 1 - \sum_{\forall r_j \in R^+_{r_i}} \sigma'_{r_j} - \sum_{\forall r_j \in R^+_{r_i}} \rho'_{r_j} \cdot (\beta_k - \alpha_k + 1) \quad (11)$$

Combining Equation (10) and Equation (11) results in

$$\rho'_{r_i} \cdot (\beta_k - \alpha_k + 1 - \Theta_{r_i}) = \beta_k - \alpha_k + 1 - \sum_{\forall r_j \in R^+_{r_i}} \sigma'_{r_j} - \sum_{\forall r_j \in R^+_{r_i}} \rho'_{r_j} \cdot (\beta_k - \alpha_k + 1)$$

We replace ρ'_{r_i} by $1 - \sum_{\forall r_j \in R^+_{r_i}} \rho'_{r_j}$, which is valid since $1 - \sum_{\forall r_j \in R^+_{r_i}} \rho'_{r_j} \geq \rho'_{r_i}$, according to Definition 15. Solving for Θ_{r_i} results in Equation (9), proving the first case.

Case 2: $r_i \in R^l_t \wedge r_i \notin R^q_t$
According to Definition 17, $r_i \in R^l_t$ implies that $\check{w}_{r_i}(\tau_1 - 1, t - 1) = \rho'_{r_i} \cdot (t - \tau_1 + 1)$. On the other hand, Definition 11 states that $r_i \notin R^q_t$ means that $w_{r_i}(t) = w'_{r_i}(t)$. By combining these results we get that

$$\check{w}'_{r_i}(\tau_1 - 1, t - 1) = \rho'_{r_i} \cdot (t - \tau_1 + 1) \quad (12)$$

We know from Lemma 1 that $w'_{r_i}(\tau_1 - 1) = w'_{r_i}(\tau_1)$. We also know from Definition 9 that $w'_{r_i}(t, t) \geq 0$.

Substituting these results into Equation (12) gives us $\check{w}_{r_i}(\tau_1, t) = \rho'_{r_i} \cdot (t - \tau_1 + 1)$, proving the second case. □

Theorem 2 ($\mathcal{LR}$ server)**.** *A CCSP arbiter belongs to the class of $\mathcal{LR}$ servers, and the service latency of an active requestor $r_i \in R$, for which $\sigma'_{r_i} \geq \sigma_{r_i}$, is equal to Equation* (9)*.*

Proof: According to [19], it is sufficient to show that $\forall t \in [\tau_1, \tau_2] : r \in R_t^e \Rightarrow \check{w}'_{r_i}(\tau_1, \tau_2) = \max(0, \rho'_{r_i} \cdot (\tau_2 - \tau_1 + 1 - \Theta_{r_i}))$. This is shown in the first case of the proof of Theorem 1. □

Theorem 2 proves that CCSP belongs to the class of $\mathcal{LR}$ servers. Our derived service latency is furthermore the same as that of SRSP, derived in [22]. Note in Equation (9) that latency and rate are decoupled by the priority level of a requestor. We conclude this section by using the service guarantee to derive a bound on the finishing time of a request in Theorem 3.

Theorem 3 (Finishing time)**.** *The finishing time of a request ω_r^k from a requestor $r \in R$, for which it holds that $\forall t \in [t_e(\omega_r^k), t_f(\omega_r^k) - 1] : r \in R_t^a$, is bounded according to*

$$t_f(\omega_r^k) \leq t_e(\omega_r^k) + \Theta_r + \frac{s(\omega_r^k)}{\rho'_r}$$

Proof: We know from Theorem 1 that a requestor in an active period $[\tau_1, \tau_2]$ receives service according to $\forall t \in [\tau_1, \tau_2] : \check{w}'_{r_i}(\tau_1, t) = \rho'_r \cdot (t - \tau_1 + 1 - \Theta_r)$. The maximum finishing time of ω_r^k equals $t + 1$ for the minimum t for which it holds that $\check{w}'_{r_i}(t_e(\omega_r^k), t) = s(\omega_r^k)$. We hence get that $\rho'_r \cdot (t - t_e(\omega_r^k) + 1 - \Theta_r) \geq s(\omega_r^k)$. Solving for t results in $t \geq t_e(\omega_r^k) + \Theta_r + \frac{s(\omega_r^k)}{\rho'_r} - 1$, which implies that $t_f(\omega_r^k) \leq t_e(\omega_r^k) + \Theta_r + \frac{s(\omega_r^k)}{\rho'_r}$. □

VI. Hardware implementation

The proposed arbiter, shown in Figure 6, has been implemented in VHDL and integrated into the Predator DDR2 SDRAM controller [23]. This controller is used in the context of a multi-processor SoC that is interconnected using the Æthereal NoC [24]. Requests arrive at a network interface (NI) on the edge of the network, where they are stored in separate buffers per requestor.

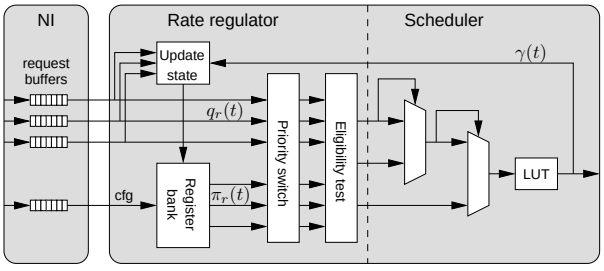

Fig. 6. A CCSP arbiter supporting three requestors.

A register bank contains a discrete representation of the service allocation and accounted potential for every requestor. These registers are programmable using memory mapped IO for run-time (re)configuration via the NoC. It is shown in [21] that the amount of over-allocation can be made arbitrarily small by increasing the precision of this representation without affecting the latency of a requestor. The static-priority scheduler is implemented by a tree of multiplexers that simply grants access to the highest priority requestor that is eligible, an operation that is faster than comparing multiple-bit deadlines, as done in [11]. The scheduled requestor is output from the arbiter, but also fed back to a unit that updates the register bank to reflect changes in potential, as discussed in Section IV-A. Configurable priorities are implemented with a programmable priority switch that maps the request buffers according to their priority levels. The switch is combined with a look-up table (LUT) that remaps the index of the scheduled requestor, as shown in Figure 6.

Synthesis of the arbiter in a 90 nm CMOS process with six ports results in a cell area of 0.0223 mm^2 at a frequency of 250 MHz, which is above 200 MHz required for a DDR2-400 SDRAM device. Figure 7 illustrates the scalability of the implementation by showing the area of the arbiter for an increasing number of ports. The speed target of 200 MHz is satisfied for up to ten requestors and the figure suggests that the area increases rather linearly in this range.

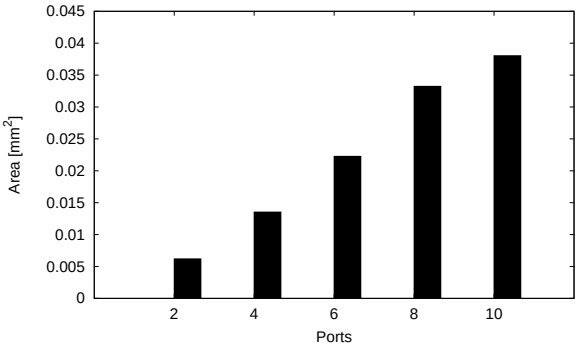

Fig. 7. The area of the arbiter for a different number of ports

VII. Experimental results

We have used CCSP as a DDR2 memory controller arbiter in a SystemC simulation of a use-case involving an H.264 video decoder. The H.264 decoder contains a number of requestors communicating through external memory. Access to a DDR2-400 SDRAM is provided by a Predator SDRAM controller [23]. A benefit of this controller is that the arbiter schedules memory accesses of 64 byte (B) to the requestors, as opposed to scheduling time, which means that the amount of work associated with a request is always known. This allows us to use the same setup to experiment with both CCSP and SRSP. The time required by the memory controller to serve a service unit corresponds to approximately 80 ns.

The use-case contains a file reader (FR) that reads an encoded image and stores it in external memory. This requestor issues requests of 64 B each and is extremely bursty. The decoder software is running on a TriMedia 3270 [25]. The TriMedia uses separate read and write connections (TM_{rd}, TM_{wr}) to communicate with external memory through an L1 cache with a line size of 128 B. Finally, a display controller

TABLE I
REQUESTOR CONFIGURATION AND RESULTS.

Requestor	σ'	ρ'	p	*avg.* Θ	$\max \Theta$	Θ
TM_{rd}	8.0	0.106	0	3.19	9	N/A
TM_{wr}	4.0	0.061	1	8.60	18	N/A
DC	2.0	0.047	2	0.10	2	N/A
FR	2.0	0.017	3	55.67	63	N/A
HRT_1	4.4	0.340	4	0.17	10	20
HRT_2	3.4	0.340	5	2.23	23	47

(DC) reads the decoded image in blocks of 128 B and shows it on a display. For the purpose of this paper, the application is considered as soft real-time with deadlines at the granularity of decoded frames. We add two hard real-time requestors, (HRT_1, HRT_2), mimicked by traffic generators, to create a hybrid system. These issue read and write requests of 128 B to external memory. High priority is assigned to the soft real-time requestors and lower priorities to the hard real-time requestors, according to the assignment strategy in [14].

We simulated the system with a number of different service allocations. The allocation parameters (σ' and ρ') of the hard real-time requestors were chosen such that the rate regulator never slowed them down and violated their bounds on service latency. For the soft real-time requestors, ρ' was chosen based on measurements such that $\rho' \geq \rho$ and $\sigma' < \sigma$. Table I lists one of the simulated configurations. A total of 600 MB/s is allocated to the requestors, corresponding to a load of 90.7% of the capacity offered by the memory controller for a 16-bit DDR2-400 device after taking unavoidable access overhead into account [23]. Table I presents average service latencies and the maximum measured service latencies for all requestors after $2{\cdot}10^8$ ns of simulation. The corresponding service latency bounds, obtained using Equation (9), are also listed for hard real-time requestors. Note that the average service latency of the soft real-time requestors includes the time required to build up sufficient potential, since $\sigma' < \sigma$. The maximum measured service latencies are lower than the bounds for both hard real-time requestors, as expected. However, we note that the difference between the maximum measured value and the bound increases with lower priorities. A reason for this is that the risk of simultaneous maximum interference from all higher priority requestors becomes increasingly unlikely with lower priorities. As a comparison, we inverted the priorities of all requestors in the use-case, resulting in maximum measured service latencies of 4 and 0 and bounds of 5 and 0 for HRT_1 and HRT_2, respectively.

All simulations have been repeated with an SRSP arbiter, and the latency results proved to be identical for every single request for all configurations. This result, suggests that CCSP, unlike SRSP, has the benefits of regulating provided service, mentioned in Section IV-A, without introducing additional latency. It is furthermore shown in [21] that the buffering requirements and burstiness at the output of the two arbiters are the same since they have identical service latencies.

VIII. CONCLUSIONS

We present a Credit-Controlled Static-Priority (CCSP) arbiter to schedule access to resources, such as interconnect and memories in systems-on-chip. CCSP is an arbiter with a rate regulator that enforces a burstiness constraint on provided service together with a static-priority scheduler. Regulating *provided* service, as opposed to regulating *requested* service has two benefits: the implementation of the regulator is less complex, and the amount of work associated with a particular request does not have to be known. We show that CCSP enjoys these benefits, without increasing latency, compared to an arbiter regulating requested service. We show that CCSP belongs to the class of latency-rate ($\mathcal{LR}$) servers and guarantees the allocated service rate within a maximum latency, required by hard real-time applications. CCSP decouples rate and allocation granularity from latency and has a low-cost implementation. An instance with six ports runs at 250 MHz and requires 0.0175 mm^2 in a 90 nm CMOS process.

REFERENCES

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

[2] K. Goossens *et al.*, "Interconnect and memory organization in SOCs for advanced set-top boxes and TV — Evolution, analysis, and trends," in *Interconnect-Centric Design for Advanced SoC and NoC*, 2004, ch. 15.

[3] 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.

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

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

[6] C. R. Kalmanek and H. Kanakia, "Rate controlled servers for very high-speed networks," *Proc. GLOBECOM*, 1990.

[7] S. J. Golestani, "A stop-and-go queueing framework for congestion management," in *Proc. SIGCOMM*, 1990.

[8] S. S. Kanhere and H. Sethu, "Fair, efficient and low-latency packet scheduling using nested deficit round robin," *High Performance Switching and Routing, 2001 IEEE Workshop on*, 2001.

[9] D. Saha *et al.*, "Carry-over round robin: a simple cell scheduling mechanism for ATM networks," *IEEE/ACM Trans. Netw.*, vol. 6, no. 6, 1998.

[10] G. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2004.

[11] J. Rexford *et al.*, "A router architecture for real-time point-to-point networks," in *Proc. ISCA*, 1996.

[12] B. Kim and K. Shin, "Scalable Hardware Earliest-Deadline-First Scheduler for ATM Switching Networks," *Proc. RTSS*, 1997.

[13] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *The Journal of Real-Time Systems*, no. 1, 1989.

[14] S. Hosseini-Khayat and A. Bovopoulos, "A simple and efficient bus management scheme that supports continuous streams," *ACM TOCS*, vol. 13, no. 2, 1995.

[15] S. Heithecker and R. Ernst, "Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements," in *Proc. DAC*, 2005.

[16] H. Zhang and D. Ferrari, "Rate-controlled service disciplines," *Journal of High-Speed Networks*, vol. 3, no. 4, 1994.

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

[18] J.-Y. L. Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag New York, Inc., 2001.

[19] 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.

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

[21] B. Akesson *et al.*, "Real-Time Scheduling of Hybrid Systems using Credit-Controlled Static-Priority Arbitration ," NXP Semiconductors, Tech. Rep., 2007, http://www.es.ele.tue.nl/~kakesson/publications/pdf/NXP-TN-2007-00119.pdf.

[22] R. Agrawal and R. Rajan, "Performance bounds for guaranteed and adaptive services," IBM Research, Tech. Rep. RC20649 (91385), May 1996.

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

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

[25] J.-W. van de Waerdt *et al.*, "The TM3270 Media-Processor," in *Proc. MICRO 38*, 2005.