Realizing Compositional Scheduling through Virtualization*

Jaewoo Lee[1†] Sisu Xi[2†] Sanjian Chen[1] Linh T.X. Phan[1]
Chris Gill[2] Insup Lee[1] Chenyang Lu[2] Oleg Sokolsky[1]
[1]University of Pennsylvania, USA
[2]Washington University in Saint Louis, USA
E-mail: {jaewoo,sanjian,linhphan,lee,sokolsky}@cis.upenn.edu, {xis,cdgill,lu}@cse.wustl.edu

***Abstract*—We present a co-designed scheduling framework and platform architecture that together support compositional scheduling of real-time systems. The architecture is built on the Xen virtualization platform, and relies on compositional scheduling theory that uses periodic resource models as component interfaces. We implement resource models as periodic servers and consider enhancements to periodic server design that significantly improve response times of tasks and resource utilization in the system while preserving theoretical schedulability results. We present an extensive evaluation of our implementation using workloads from an avionics case study as well as synthetic ones.**

I. Introduction

Modular development of real-time systems using time-aware components is an important means of reducing complexity of modern real-time systems. Components encapsulate real-time workloads, such as tasks, and are supported by a *local scheduler* that handles those workloads. Components share computational resources with other components. A *higher-level scheduler* is then used to allocate resources to local schedulers, guided by the components' resource needs, which they expose in their interfaces.

Several compositional scheduling frameworks (CSF) have been proposed to support such a component-based approach. Scheduling needs to be compositional to achieve a desirable separation of concerns: on the one hand, the high-level scheduler should not have access to the component internals and should operate only on component interfaces; on the other hand, schedulability analysis of a component's workload and generation of the component interface need to be performed independently from any other components in the system. Further, schedulability analysis at the higher level should be performed only on the basis of component interfaces.

In this paper, we present the Compositional Scheduling Architecture (CSA), which is an implementation of a CSF that relies on periodic resource models as component interfaces. Theoretical background for such an architecture, which provides interface computation for real-time workloads and schedulability analysis, has been laid out in [1], [2]. CSA is built on the virtualization framework provided by Xen, with the VMM being the root component and the guest operating systems (domains) being its subcomponents. Each domain interface is implemented as a periodic server [3], which behaves like a periodic task. The virtual machine monitor (VMM) allocates resources to the domains by scheduling the corresponding servers in the same manner as scheduling a set of tasks. The above combination of compositional scheduling and virtualization enables legacy real-time systems to be consolidated into a single host machine without the need to modify or reconfigure their guest operating systems (OS), thus providing both benefits of real-time scheduling theory and mainstream virtualization. In addition, virtualization can readily support different local schedulers of sub-components within a compositional scheduling framework (by running a different guest OS), which cannot be achieved in a reservation-based native system without major modifications to the OS.

We also discuss challenges encountered during our implementation of the CSA and our approach to overcome those challenges. In particular, we discovered that CSF theory needed to be modified because of the fixed scheduling quantum imposed by Xen. This precludes direct use of the interface computation algorithm described in [4], since the resource bandwidth computed for the interface has to be an integer multiple of the quantum. Moreover, we discovered that a naive implementation of the periodic server is not work conserving and may lead to significant underutilization of the available computational resources.

Contributions. This paper makes the following contributions to the state-of-the-art real-time systems research:

- We illustrate the feasibility of compositional scheduling using Xen virtualization platform. Our implementation, called CSA, enables timing isolation among virtual machines and supports timing guarantees for real-time tasks running on each virtual machine. CSA includes a wide range of real-time scheduling algorithms at the VMM level, and it is easily extensible with new algorithms.
- We introduce several enhancements to the periodic server design in CSA to optimize the performance of both hard and soft real-time applications. Our enhancements preserve conservative CSF schedulability analysis, while yielding substantial improvements in observed response times and resource utilization, which are desirable for not only soft real-time but also many classes of hard real-time applications.
- We provide an extension of the CSF theory for quantum-

* This research is supported in part by ARO grant W911NF-11-1-0403 and NSF grants CNS-0834524, CNS-1117185 and CNS-0834755.
† The first two authors have made equal contributions to this work.

based platforms and fixed-priority scheduling.
- We offer an extensive evaluation of the performance of CSA with respect to a variety of workloads, some of which originate from the avionics system reported in [5], and others that are synthetic.

To the best of our knowledge, CSA is the first open source implementation of a real-time virtualization platform with support for compositional scheduling. It is implemented on Xen, but is portable to other virtualization platforms.

II. Background

A. Compositional Scheduling Framework (CSF)

In a CSF, the system consists of a set of *components*, where each component is composed of either a set of subcomponents or a set of tasks. Each component is defined by $C = (W, \Gamma, A)$, where: W is a workload, i.e., a set of tasks (components); Γ is a resource interface; and A is a scheduling policy used to schedule W, which in our setting is Rate Monotonic (RM). All tasks are periodic, where each task T_i is defined by a period (and deadline) p_i and a worst-case execution time e_i, with $p_i \geq e_i > 0$ and $p_i, e_i \in \mathbb{N}$. Interface Γ is a periodic resource model (described below).

Periodic resource model. A periodic resource model (PRM) is defined by $\Gamma = (\Pi, \Theta)$, where Π is the resource period and Θ is the execution budget guaranteed by Γ in every period. The *bandwidth* of Γ is defined by Θ/Π. A PRM is *(bandwidth) optimal* for W iff it has the smallest bandwidth among all PRMs that can feasibly schedule W. A workload W is *harmonic* iff the periods of its tasks (subcomponents' interfaces) are pairwise divisible.

The minimum resource guaranteed by a PRM Γ is captured by a *supply bound function* (SBF) [1], written as $\mathsf{sbf}_\Gamma(t)$, which gives the minimum number of execution units provided by Γ over any time interval of length t, for all $t \geq 0$. The SBF of $\Gamma = (\Pi, \Theta)$ for a workload W is thus given by [1], [5]:

$$\mathsf{sbf}_\Gamma(t) = \begin{cases} y\Theta + \max\big(0,\, t - x - y\Pi\big), & \text{if } t \geq \Pi - \Theta \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where

- $x = (\Pi - \Theta)$ and $y = \lfloor \frac{t}{\Pi} \rfloor$, if W is harmonic; and
- $x = 2(\Pi - \Theta)$ and $y = \lfloor \frac{t-(\Pi-\Theta)}{\Pi} \rfloor$, otherwise.

Schedulability condition. Given $C = (W, \Gamma, RM)$ with $W = \{T_1, T_2, \cdots, T_n\}$, $T_i = (p_i, e_i)$, and $p_1 \leq p_2 \leq \cdots \leq p_n$. Here, T_i is a periodic task or a PRM interface of a subcomponent of C. Resource demands of C are characterized by the request bound functions (RBFs) of W, given by $\mathsf{rbf}_{W,i}(t) = \sum_{k \leq i} \left(\left\lceil \frac{t}{p_k} \right\rceil e_k \right)$ for all $1 \leq i \leq n$ [6]. Lemma 1 states its schedulability condition based on $\mathsf{rbf}_{W,i}$ and sbf_Γ [1].

Lemma 1: Given a component $C = \big(W, \Gamma, RM\big)$ with $W = \{T_1, T_2, \cdots, T_n\}$ and $T_i = (p_i, e_i)$ for all $1 \leq i \leq n$. Then, C is schedulable (Γ can feasibly schedule W) iff

$$\forall\ 1 \leq i \leq n,\ \exists t \in [0, p_i]\ s.t.\ \ \mathsf{sbf}_\Gamma(t) \geq \mathsf{rbf}_{W,i}(t). \quad (2)$$

From Lemma 1, a necessary schedulability condition for C is $U_\Gamma \geq U_W$, where $U_\Gamma = \Theta/\Pi$ and $U_W = \sum_{i=1}^{n} e_i/p_i$. The difference, $U_\Gamma - U_W$, is called the *interface overhead* of C. Thus, Γ is optimal for W iff it has the smallest interface overhead compared to all interfaces that can feasibly schedule W. It can be implied from Lemma 1 and Eq. (1) that the interface computed assuming a harmonic workload has a smaller (possibly zero) interface overhead than that of an interface computed assuming a general workload.

PRMs as periodic servers. Each PRM interface $\Gamma = (\Pi, \Theta)$ is implemented as a periodic server [3] with period Π and execution budget Θ, i.e., the server is ready for execution periodically every Π time units and its execution time is at most Θ time units. Thus, interfaces of components can be scheduled in the same manner as periodic tasks are. Further, a component is schedulable iff its interface (i.e., periodic server) is feasibly scheduled by its parent component.

B. Overview of Xen

Xen [7], the most widely used open source virtual machine monitor (VMM), allows a set of guest operating systems (OS), called *domains*, to run concurrently. To guarantee that every guest OS receives an appropriate amount of CPU time, Xen provides a scheduling framework within which developers can implement different scheduling policies. In this framework, every core in a guest OS is instantiated as a Virtual CPU (VCPU), and a guest OS can have as many VCPUs as there are underlying physical cores. Xen schedules VCPUs in the same manner as a traditional OS schedules processes, except that its pluggable scheduling framework allows different scheduling policies to be used. A special VCPU, called IDLE VCPU, is also created for each physical core to represent an idle task in a traditional OS. When the IDLE VCPU is scheduled, the specific physical core becomes idle. Xen by default uses the "credit" scheduler, which schedules VCPUs in a proportional fair share manner. This scheduler has been shown to be unsuitable for real-time applications [8].

In our earlier work [8], we have developed RT-Xen, a real-time virtual machine manager that supports hierarchical real-time scheduling in Xen. The compositional scheduling architecture (CSA) presented in this paper builds on and complements RT-Xen with a compositional scheduling capability. It differs from RT-Xen in four important aspects: (1) while RT-Xen instantiates hierarchical real-time scheduling in Xen, it was not designed to support the CSF model where the resource demand of a component is encapsulated by its interface; (2) RT-Xen focuses on the implementation and evaluation of different existing server algorithms, including Polling Server, Deferrable Server, Sporadic Server, as well as the classical Periodic Server that is used as a baseline in this work - in contrast, this work proposes two new work-conserving Periodic Server algorithms to improve soft real-time performance; (3) this work presents a new method to select the optimal interface parameters for a given scheduling quantum for RM scheduling, an issue not addressed by RT-

Xen or the earlier work; (4) this work introduces an integrated scheduler architecture that allows different periodic servers to be instantiated through component reuse and enables the schedulers to be swapped online.

C. Challenges

Despite the availability of considerable *theoretical* results on CSF for real-time systems, those results have yet to be implemented in a virtualization platform such as Xen. The gap between theory and systems results in two significant problems. First, real-time system integrators cannot take advantage of the body of CSF theory in practice due to a lack of system support. We have addressed that issue by developing a novel *Compositional Scheduling Architecture (CSA)* within the Xen virtual machine monitor (VMM). This unified scheduling architecture supports different scheduling policies at the VMM level, while preserving the modularity and extensibility of the scheduler implementation.

Moreover, without implementation and experimentation on a real system, it is not possible to explore crucial system design tradeoffs and practical issues involved in realizing a particular CSF in a given virtualization platform, such as the following important practical issues we face in realizing the PRM-based CSF in Xen.

Non-work-conserving scheduling. The periodic server policy was proposed as an effective mechanism for implementing scheduler interfaces in CSF. However, the classical periodic server algorithm [3], referred to as a *Purely Time-driven Periodic Server (PTPS)* in this paper, adopts a non-work-conserving policy. Specifically, when a higher-priority component has no work to do, it simply idles away its budget while lower-priority components are not allowed to run. RT-Xen [8] emulates this feature by scheduling the IDLE VCPU to run while a high-priority domain idles away its budget. This scenario arises when a high-priority domain underutilizes its budget, e.g., due to an interface overhead or an over-estimation of tasks' execution times when configuring the domains' budgets. While the non-work-conserving approach does not affect the *worst-case* guarantees provided by PTPS, it wastes CPU cycles while increasing the response times of low-priority domains. This is particularly undesirable for soft real-time systems, as well as many hard real-time systems where short response times are also beneficial.

Scheduling quantum. While previous interface calculation techniques assume continuous values for interface budgets, a real system such as Xen must deal with quantized scheduling. For example, experimental results with RT-Xen showed that 1ms is a suitable scheduling quantum within Xen [8] in order to balance scheduling overhead and temporal granularity of scheduling. To deal with quantized scheduling, new techniques are needed to compute the bandwidth optimal interface for a guest OS and the maximum value of the optimal period when using the RM scheduling algorithm.

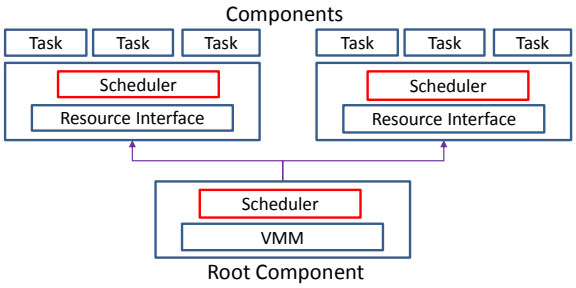


Fig. 1: Compositional Scheduling Architecture

III. Solution Approach

Real-time guarantees in Xen can be achieved via compositional schedulability analysis in our *Compositional Scheduling Architecture (CSA)*. As is shown in Figure 1, the Xen VMM corresponds to a root component, and each Xen domain corresponds to a subcomponent of the root component in the CSA. The Xen VMM's scheduler (extended from the original RT-Xen interfaces) schedules domains based on their PRM interfaces, which are implemented as periodic servers (described in Section III-A). Each server's period and budget are computed using our quantum-based extension of compositional scheduling theory (described in Section III-B) to ensure schedulability of tasks in the underlying domain. The system is hence schedulable iff all servers are feasibly scheduled by the VMM's scheduler.

A. Periodic Server Design

In this section, we present two enhanced variations of the *purely time-driven periodic server* to optimize runtime performance and resource-use efficiency, namely the *work-conserving periodic server* and the *capacity-reclaiming periodic server*. These variations differ in how a server budget changes when the server has remaining budget but is idle (i.e., has no unfinished jobs), or when it is non-idle but has no budget left. Recall that in the classical purely time-driven periodic server, a server's budget is replenished to full capacity every period. The server is eligible for execution only when it has non-empty budget, and its budget is always consumed at the rate of one execution unit per time unit, even if the server is idle. In the work-conserving periodic server variant, whenever the currently scheduled server is idle, the VMM's scheduler lets another lower-priority non-idle server run; thus, the system is never left idle if there are unfinished jobs in a lower-priority domain. Finally, the capacity-reclaiming periodic server variant further utilizes the unused resource budget of an idle server to execute jobs of any other non-idle servers, effectively adding extra budget to the non-idle servers. In what follows, "the scheduler" refers to the VMM's scheduler, unless explicitly mentioned otherwise.

Purely Time-driven Periodic Server (PTPS). As is mentioned above, the budget of a PTPS is replenished at every period and its budget is always consumed whenever it is executed. As Xen is an event-triggered virtual platform,

we introduce a mechanism to allow this time-driven budget replenishment and scheduling approach in CSA. Note that the PTPS approach is not work-conserving since the system resource is always left unused if the currently scheduled server (Xen domain) is idle.

Work-Conserving Periodic Server (WCPS). The budget of a WCPS is replenished in the same fashion as that of a PTPS. However, if the currently scheduled server (C_H) is idle, the scheduler picks a lower-priority non-idle server to execute, according to the following work conserving rules:

(1) Choose a lower-priority server, C_L, with the highest priority among all non-idle lower-priority servers.

(2) Start executing C_L and consuming the budgets of both C_L and C_H, each at the rate of one unit per time unit.

(3) Continue running C_L until one of the following occurs: (a) C_L has no more jobs to execute; (b) C_L has no more budget; (c) Some jobs in C_H become ready and C_H has remaining budget; or (d) C_H has no more budget. In the case of (a) or (b), the scheduler goes back to Step 1 where it selects another lower-priority non-idle server. In the case of (c), C_L immediately stops its execution and budget consumption, whereas C_H resumes its execution. In the case of (d), C_L immediately stops its execution and budget consumption; a new server will be chosen for execution by the scheduler.

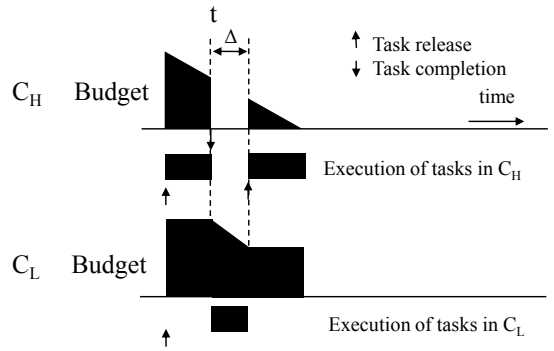


Fig. 2: Execution of Servers in the WCPS Approach.

Figure 2 illustrates a general scenario under the work conserving rule. In this scenario, C_H becomes idle at time t and thus, a lower-priority server C_L is selected for execution. At time $t+\Delta$, some jobs in C_H become ready (i.e., case (c) in Step 3); therefore, C_H preempts C_L and resumes its execution. By allowing C_L to run (if C_H is idle) and maintaining the same execution for C_H, the WCPS achieves shorter overall response times of tasks compared to PTPS while preserving conservative CSF schedulability.

Lemma 2: A CSF system is schedulable under the WCPS approach if it is schedulable under the PTPS approach.

Proof: The lemma is derived from two key observations: (1) at the VMM level, the scheduler can still schedule the domains because their corresponding servers (domains' interfaces) are unchanged; and (2) each server can schedule its own workload because the total resource given to each server is unchanged and only the idle time of C_H is utilized by C_L. The complete proof can be found in [9]. ■

Capacity Reclaiming Periodic Server (CRPS). Like the WCPS, the CRPS is also work conserving and the budget of a server is replenished to full capacity every period. However, the CRPS improves tasks' response times by allowing the idle time of the currently running server to be utilized by *any* other server (including higher-priority ones). Specifically, we define the *residual capacity* of a server to be the time interval during which the server consumes its budget but is idle (e.g., C_H has a residual capacity of $[t, t+\Delta]$ in Figure 2). At run time, the server budget is modified using the following capacity-reclaiming rule: during a residual capacity interval of a server C_H, the resource budget of C_H is re-assigned to any other non-idle server C_L and only this budget is consumed (e.g., the budget of C_L remains intact).

Similarly, we can show that the CRPS also preserves conservative CSF schedulability. Since each CRPS server gets not only its own resource budget but also the extra budgets of idle servers, it can potentially finish its jobs earlier than a corresponding WCPS or PTPS can. This results in an overall improvement in tasks' response times compared to the WCPS and PTPS approaches, as is also validated in our evaluation (see Section IV). Note that due to the capacity reclaiming capability, the CRPS is the most difficult to implement among the three server variants.

B. Interface Computation for Quantum-based Platforms

In the existing CSF theory [4], the optimal PRM interface of a component is computed by iterating the resource period from 1 to a manually chosen value, while assuming rational values for the resource budget. For this approach to be implementable, given a particular time granularity of a Xen platform, the resource budget needs to be scaled to a multiple of the time unit. As is illustrated in our technical report [9], this scaling may lead to a sub-optimal resulting interface. Further, a naive choice of the period's bound can also result in sub-optimality. To address these shortcomings, in this section we introduce an algorithm for computing the optimal PRM interface for quantum-based platforms under RM scheduling.

Upper bound on the optimal interface period. Theorem 1 gives an upper bound on the resource period of the optimal interface of a given workload $W = \{(p_i, e_i) \mid 1 \leq i \leq n\}$ under RM. Intuitively, a PRM interface Γ is schedulable *only if* its upper supply bound function (USBF) (i.e., the minimum sloped upper linear curve of the interface's SBF) meets each $\mathsf{rbf}_{W,i}$ at a step-point of $\mathsf{rbf}_{W,i}$ and is below $\mathsf{rbf}_{W,i}$ at all other points in $[0, p_i]$. We call these meeting points *critical points*, with $\mathsf{CrT}_{W,i}$ denoting the set of time-coordinates of the critical points of $\mathsf{rbf}_{W,i}$. Thus, the optimal resource bandwidth is lower bounded by the minimum slope of all linear curves f_i^t that are equal to $\mathsf{rbf}_{W,i}$ at time $t \in \mathsf{CrT}_{W,i}$ and smaller than $\mathsf{rbf}_{W,i}$ at all other times. As a result, the optimal resource period is upper bounded by the minimum of all P_i ($1 \leq i \leq n$), where P_i is the maximum of the periods P_i^t of the PRMs with USBFs f_i^t for all $t \in \mathsf{CrT}_{W,i}$. Theorem 1 computes this upper bound based on an initial feasible PRM Γ_c for W. A detailed proof

of the theorem is available in [9].

Theorem 1: Suppose $\Gamma_c = (\Pi_c, \Theta_c)$ is the minimum bandwidth PRM among all PRMs that can feasibly schedule a workload W and whose period is at most Π_c. Then, the optimal PRM $\Gamma_{\mathsf{opt}} = (\Pi_{\mathsf{opt}}, \Theta_{\mathsf{opt}})$ for W satisfies $\Pi_c \leq \Pi_{\mathsf{opt}} \leq \mathsf{MaxResPeriod}(\kappa, W)$ where $\kappa = \frac{\Theta_c}{\Pi_c}$ and

$$\mathsf{MaxResPeriod}(\kappa, W) \stackrel{\mathsf{def}}{=} \min_{1 \leq i \leq n} \left(\max_{t \in \mathsf{CrT}_{W,i}} \frac{\kappa \cdot t - \mathsf{rbf}_{W,i}(t)}{\kappa(1-\kappa)} \right).$$

Algorithm 1 Optimal integer-valued interface computation.

Input: A workload W
Output: The optimal integer-valued PRM Γ_{opt} for W
1: $\Theta' = \mathsf{MinExec}(p_n, W)$
2: $\kappa = \frac{\Theta'}{p_n}$
3: $\Gamma_{\mathsf{opt}} = (p_n, \Theta')$
4: $\Pi_{\max} = \mathsf{MaxResPeriod}(\kappa, W)$
5: **for** $\Pi = 1$ to $\Pi_{\max}$ **do**
6: $\quad \Theta = \mathsf{MinExec}(\Pi, W)$
7: $\quad$ **if** $\frac{\Theta}{\Pi} < \kappa$ **then**
8: $\quad\quad \kappa = \frac{\Theta}{\Pi}$
9: $\quad\quad \Gamma_{\mathsf{opt}} = (\Pi, \Theta)$
10: $\quad\quad \Pi_{\max} = \min(\Pi_{\max}, \mathsf{MaxResPeriod}(\kappa, W))$
11: $\quad$ **end if**
12: **end for**
13: **return** Γ_{opt}

Optimal integer-valued PRM period computation. Algorithm 1 computes the optimal integer-valued PRM of a given workload W by incorporating the above upper bound of the resource period $\mathsf{MaxResPeriod}(\kappa, W)$. In Lines 1-2, $\mathsf{MinExec}(p_n, W)$ gives the minimum budget for the period p_n such that the resulting PRM can feasibly schedule W (i.e., satisfies Lemma 1), and κ denotes the corresponding bandwidth. The initial bound on the resource period is given by $\Pi_{\max}$ in Line 4. The function MaxResPeriod(κ,W) in Lines 4 and 10 computes the upper bound on the optimal PRM as defined in Theorem 1. Finally, the minimum bandwidth acquired during the algorithm execution is stored in κ, and it is used to re-evaluate $\Pi_{\max}$ (Lines 7–11).

C. System Architecture

This section presents details about CSA and the implementation of the PTPS, WCPS, and CRPS in Xen.

Compositional Scheduling Architecture. In CSA, at the most general level, an existing Xen scheduling framework provides interfaces to a specific scheduler. Each scheduler has its own data structure but must implement several common functions including *wake*, *do_schedule*, *sleep*, and *pick_cpu*. Since the three CSA schedulers mainly differ in how the budget is consumed, we provide a real-time sub-framework which abstracts common functions and data structures among the CSA schedulers. The scheduling-related functions such as *do_schedule* are implemented as pointers to functions in sub-schedulers. Under the real-time sub-framework, we implement PTPS, WCPS, and CRPS separately. Figure 3 shows a high-level view of CSA.

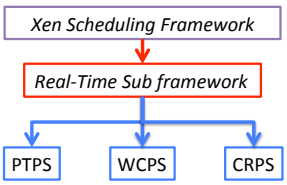


Fig. 3: The CSA Schedulers Architecture

Data structure. Each VCPU has three parameters: *budget*, *period*, and *priority*. In CSA, the priority is determined based on the VCPU's period according to the RM scheduling policy. Each VCPU is implemented as a periodic server, where its *budget* is set to full every *period* units of time, and the VCPU consumes its budget whenever it is executed. Note that *budget* is a fixed amount of resource given to a VCPU periodically, computed so as to provide real-time guarantees for all tasks in the system. This is different from *credit* in Xen, which is given to each VCPU dynamically and is computed such that all VCPUs have their fair shares of resource.

Each physical core has two queues: a *Run Queue* (RunQ) and a *Ready Queue* (RdyQ). Both queues are used to store active VCPUs. The IDLE VCPU, which has the lowest priority, is always located at the end of the RunQ. These queues operate as follows.

The RunQ stores VCPUs that have jobs to run (regardless of their budgets), sorted by priority. Whenever the *do_schedule* function is triggered, it first returns the current VCPU to the RunQ or the RdyQ. It then selects an appropriate VCPU from the RunQ based on the scheduling decision, and runs the selected VCPU for one millisecond (which has been shown to be a suitable scheduling quantum in [8]).

The RdyQ holds the VCPUs that have no jobs to run, sorted by their priorities. Because one VCPU can consume other VCPUs' *budget* (e.g., via capacity reclaiming in the CRPS, or scheduling the IDLE VCPU while consuming other VCPUs' *budget* in PTPS), all active VCPUs' information is needed to enable work conserving and capacity reclaiming.

Implementation of the *do_schedule* function. The *do_schedule* function is responsible for updating VCPUs' information and making scheduling decisions. We first present the PTPS algorithm, as is shown in Algorithm 2, and then explain the WCPS and CRPS extensions.

In this algorithm, the function queuePick returns the highest priority VCPU with positive *budget* in the RunQ or in the RdyQ. Whenever a higher priority VCPU has a positive budget, we consume its budget by either scheduling it (if it has jobs to execute) or scheduling the IDLE VCPU (otherwise). Lines 1–6 demonstrate how we consume the highest priority VCPU's budget. Lines 12–17 show how the next VCPU is selected.

For the WCPS, if a higher-priority VCPU has budget but is idle, instead of scheduling the IDLE VCPU, we schedule in advance the next highest priority VCPU among all the non-idle lower-priority ones (so as to improve the responsiveness of jobs belonging to that VCPU); in this case, we consume

Algorithm 2 *do_schedule* function for PTPS

```
Input: currentVCPU, RunQ, RdyQ
Output: nextVCPU to run next
 1: rdyVCPU = queuePick(RdyQ)
 2: if currentVCPU = IDLE_VCPU then
 3:     consume budget of rdyVCPU
 4: else
 5:     consume budget of currentVCPU
 6: end if
 7: if currentVCPU has jobs to run then
 8:     insert currentVCPU into RunQ
 9: else
10:     insert currentVCPU into RdyQ
11: end if
12: nextVCPU = queuePick(RunQ)
13: if priority(rdyVCPU) > priority(nextVCPU) then
14:     nextVCPU = IDLE_VCPU
15: else
16:     remove nextVCPU from RunQ
17: end if
18: return nextVCPU
```

both *budgets* in parallel. As a result, in Lines 12–17, the algorithm always returns the VCPU from the RunQ, denoted by queuePick(RunQ). Further, in Lines 1–6, if queuePick(RdyQ) has a higher priority than that of queuePick(RunQ), their *budgets* will be both consumed.

For CRPS, only one budget is consumed at a time and "Capacity Reclaiming" is enabled between active VCPUs. In Lines 1–6, the CRPS always consumes the highest *priority* VCPU's budget among currentVCPU, queuePick(RunQ), and queuePick(RdyQ). In Lines 12–17, if the function queuePick(RunQ) returns a VCPU that is different from the IDLE VCPU, that VCPU will be scheduled. Otherwise, the IDLE VCPU is returned. There are two cases for this: either the RunQ is empty, or all active VCPUs on RunQ have no budget left. In the former case, the IDLE VCPU will be scheduled. In the latter case, if queuePick(RdyQ) returns a valid VCPU (i.e., other VCPUs have *budget*), the returned VCPU will be executed; otherwise, all active VCPUs have no budget left and thus, the IDLE VCPU will be scheduled (even if the active VCPUs still have jobs to execute). In other words, we do not allow budget to be *stolen* from the IDLE VCPU. The implementations of all the above algorithms, along with the hot-swap tool and the periodic tasks, are open source and can be found in [10].

IV. Evaluation

This section presents our evaluation of the PTPS, WCPS, and CRPS approaches that are implemented in our CSA. We focus on the run-time performance of real-time tasks, considering the following two evaluation criteria: (1) *responsiveness*, which is the ratio of a job's response time to its relative deadline; and (2) *deadline miss ratio*. Our evaluation consists of two types of workloads: synthetic workloads (Section IV-B) and ARINC workloads obtained from an avionics system (Section IV-C).

A. Experiment Setup

We implemented CSA in Xen version 4.0. Fedora 13 with para-virtualized kernel 2.6.32 is used for all domains. We pinned *Domain 0* to core 0 with 1 GB memory, and pinned all the guest operating systems to core 1 with 256 MB memory each. This configuration is used to provide minimal interference as in an ideal single core environment[1]. During the experiments, we shut down the network service as well as other inessential applications to avoid other sources of potential interference. The experiments for synthetic workloads were done on a Dell Q9400 quad-core processor while the experiments for ARINC workloads were performed on a Dell Vostro 430 quad-core processor, neither with hyper-threading. During the experiments, SpeedStep was disabled and all cores constantly ran at 2.66 GHz.

We assume all tasks are CPU intensive and independent of each other. Every task is characterized by three parameters: *worst case execution time (WCET)*, *period* (equals *deadline*), and *execution time factor (ETF)*. Here, the *ETF* represents the variance of each job's actual execution time (uniformly distributed in the interval $(WCET*ETF, WCET)$). An *ETF* of 100% indicates that every job of the task takes exactly *WCET* units of time to finish. The task model fits typical soft real-time applications (e.g., multimedia decoding applications where frames' processing times are varied but are always below an upper limit).

In the rest of the paper, U_W denotes the total utilization of all tasks in the system (utilization of the workload); U_{RM} denotes the total bandwidth of interfaces (utilization of resource models); $U_{RM} - U_W$ denotes the interface overhead.

Due to space constraints, for more detailed information about task implementation and how to set up the guest OS schedulers we refer the reader to [9].

Real-time scheduling of domains. We first determined the domains' resource needs by computing an optimal PRM interface for each domain. These interfaces were implemented as PTPS, WCPS, or CRPS variants of periodic servers, which were then scheduled by the VMM. For synthetic workloads, we applied Algorithm 1 to compute the optimal integer-valued PRM interfaces for the domains. The PRM interfaces of ARINC domains were computed based on Eq. (1) using the harmonic workload case. Since the domain periods are pre-specified in the ARINC workloads, the quantum-based interface computation technique in Algorithm 1 cannot be applied. Therefore, we resorted to computing optimal rational-valued interfaces, and then rounding up the budgets to the closest integer values. Although the real-valued interfaces may have interface overheads of zero, rounding may introduce additional overheads, effectively allocating extra budget to the corresponding domains.

For each workload and corresponding interfaces obtained as above, we repeated the experiment and evaluated the respec-

[1]Delegating a separate core to handle I/O and interrupts to reduce overheads is a common practice in multi-core real-time systems research (see e.g., [11]).

tive performances of the system when setting the hypervisor scheduler to be WCPS, CRPS, and the baseline PTPS.

B. Synthetic Workloads

The purpose of this set of experiments is to compare the *soft real-time* performance of the three different periodic servers. The PTPS, WCPS, and CRPS servers differ primarily in how idle time is utilized within the system. The idle time comes from two main sources: the interface overhead due to theoretical pessimism [1]; and over-estimation of tasks' execution times (also called *slack*). Hence, we design two sets of experiments to show the effect of different idle times: (1) The range for the workload periods is varied to create different interface overheads; (2) The *ETF* for the jobs is varied so that if a job executes less than its *WCET*, it would potentially give some *slack* to other domains.

For *soft real-time* systems, we are interested not only in schedulable situations but also in overloaded situations. As a result, we ranged the U_W from 0.7 to 1.0, with a step of 0.1, to create different U_W conditions.

All the experiments were conducted as follows. We first defined a particular U_W, and then generated tasks (utilization uniformly distributed between 0.2% and 5%) until the U_W was reached. The distributions of execution times are typically application dependent; here, we used the uniform distribution, which has been commonly used in the real-time scheduling literature (see e.g., [11], [12]). Using this generation method, the generated U_W is usually larger than the desired one, but would only be 0.05 more in the worst case. After all the tasks were generated, we randomly distributed the tasks among five domains.

We ran each experiment for 5 minutes, and then calculated the $\frac{ResponseTime}{Deadline}$ for all the task sets within each domain of the experiment. For clarity of presentation, any job whose $\frac{ResponseTime}{Deadline}$ is greater than 3 is clipped at 3.

Impact of Task Period. We varied the task period range in this experiment to create different interface overheads, and evaluated the three schedulers for the generated task sets. For each different U_W (from 0.7 to 1.0), we generated three different task sets whose periods are uniformly distributed between (550ms, 650ms), (350ms, 850ms), and (100ms, 1100ms), respectively. From the calculated interfaces, the (350ms, 850ms) task period range gives the most interface overhead, followed by (100ms, 1100ms), and then (550ms, 650ms). For all the experiments, the *ETF* value was set to 100%. In other words, we let all jobs execute at their worst case execution times, so that the idle time comes only from the interface overheads. Note that when the U_W is the same, we scheduled different task sets under different task periods.

Figure 4 shows the results for all domains under $U_W = 0.9$, where DMR means Deadline Miss Ratio. This $U_W (= 0.9)$ represents a typical heavily *overloaded* situation; other cases include [9]: either guaranteed to be schedulable theoretically and only incurred negligible deadline miss ($U_W = 0.7$), not heavily *overloaded* ($U_W = 0.8$), or too *overloaded* to be schedulable ($U_W = 1.0$). Detailed results about other U_W can be found in [9].

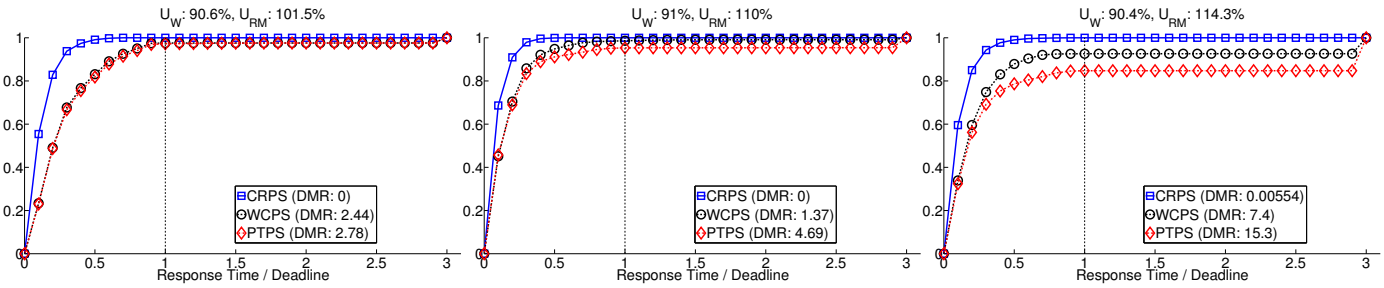
Since we are using rate monotonic scheduling, the higher priority domains have shorter periods, and thus have a larger number of jobs. The data in Figure 4 are therefore dominated by the results for higher priority domains. Lower priority domains, though having fewer jobs, suffer most from the *overloaded* situation. As such, we plot the data for the lowest priority domain (domain 5) in Figure 5 with the interface parameters given in the format of (*period*, *budget*). Figure 4 and Figure 5 clearly show that the CRPS outperforms the WCPS, which in turn outperforms the PTPS. Notably, with an interface overhead of 24% (Figure 5c), while all jobs miss their deadlines under the PTPS ($\frac{ResponseTime}{Deadline} > 1$), 60.5% and 6.2% of the jobs in domain 5 missed their deadlines under the WCPS and CRPS, respectively. These results demonstrate the effectiveness of the work-conserving and capacity-reclaiming mechanisms in exploiting the interface overhead to improve the performance of low-priority domains. The CRPS is the most effective of these approaches for implementing the interfaces in CSA.

Impact of *Execution Time Factor (ETF)*. In *real-time* applications such as multimedia frame decoding, every frame may take a different amount of time to finish. Traditionally, the *WCET* is used to represent every task's execution time. This usually results in a relatively large interface, giving more idle time for the domain.

In this set of experiments, the same U_W ranging from 0.7 to 1.0 were used. Under each U_W, we only generated one task set. Then, for each particular task set, three *ETF* values (100%, 50%, 10%) were configured for the three highest priority domains, while leaving the two low priority ones with an *ETF* of 100%. A lower *ETF* value means a lower "actual" U_W for that domain; for example, if an *ETF* of 10% is applied, all jobs' execution time uniformly distributes between 10% and 100% of *WCET*. On average, the actual U_W is 0.55 ($\frac{100\%+10\%}{2}$). All task periods were uniformly distributed between 550 ms and 650 ms. We note that the idle time comes not only from the interface overhead but also from the over-estimation of jobs' execution times.

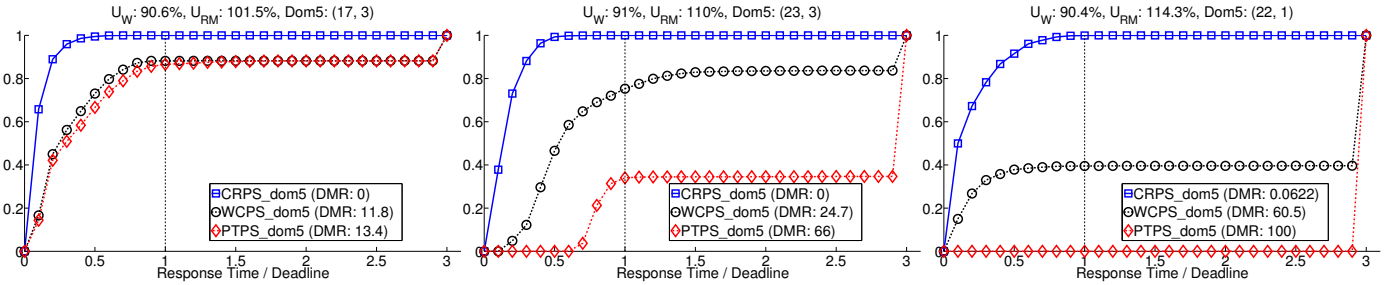
Figure 6 shows the box plot results for all U_W for the lowest priority domain. Results for all domains exhibit the same behavior. On each box, the central mark represents the median value, whereas the upper and lower box edges show the 25th and 75th percentiles separately. If the data values are larger than $q_3+1.5*(q_3-q_1)$ or smaller than $q_1-1.5*(q_3-q_1)$ (where q_3 and q_1 are the 75th and 25th percentiles, respectively), they are considered outliers and plotted via individual markers. Within one subfigure, the boxes are divided into three sets, from left to right, corresponding to the results under the *ETFs* of 100%, 50%, and 10%, respectively. The detailed CDF plots for all the results can be found in [9].

As is shown in Figure 6, the CRPS again outperforms the WCPS and PTPS. In Figure 6c, the deadline miss ratio under the PTPS stays constant when the *ETF* is varied (26.9%,



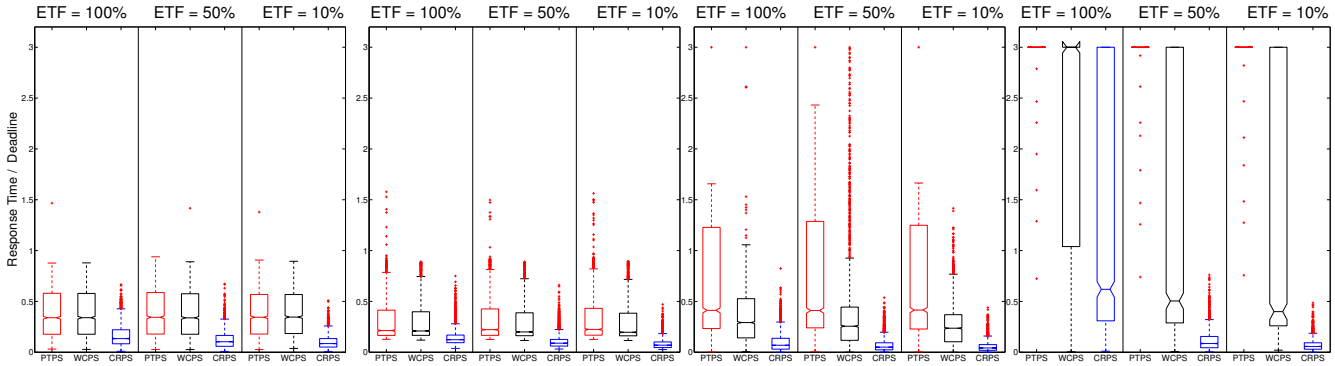
(a) Period: (550, 650), Interface Overhead: 10.9% (b) Period: (100, 1100), Interface Overhead: 19.1% (c) Period: (350, 850), Interface Overhead: 24%

Fig. 4: CDF Plot of $\frac{ResponseTime}{Deadline}$ for All Tasks in Five Domains under Designed $U_W = 0.9$ varying Task Periods



(a) Period: (550, 650), Interface Overhead: 10.9% (b) Period: (100, 1100), Interface Overhead: 19.1% (c) Period: (350, 850), Interface Overhead: 24%

Fig. 5: CDF Plot of $\frac{ResponseTime}{Deadline}$ for Tasks in the Lowest Priority Domain under Designed $U_W = 0.9$ varying Task Periods



(a) $U_W = 0.703$ (b) $U_W = 0.832$ (c) $U_W = 0.91$ (d) $U_W = 1.021$

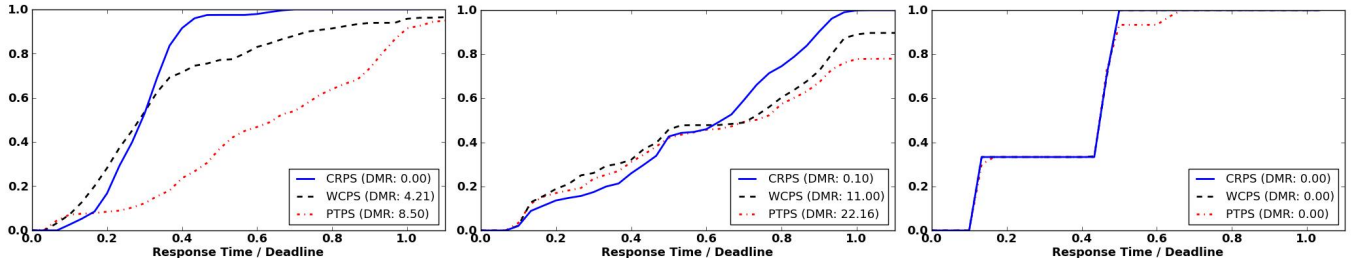
Fig. 6: Box Plot of $\frac{ResponseTime}{Deadline}$ for Tasks in the Lowest Priority Domain under Different U_W and *ETF* Values

27.3%, and 27.3% respectively), while performance improvement is seen under the WCPS (11.7%, 8.51%, and 0.49%) and CRPS (0.02%, 0%, and 0%). In an extremely *overloaded* situation (Figure 6d), all jobs missed their deadlines under the PTPS, whereas (75.6%, 32.7%, and 31.3%) of jobs missed their deadlines under the WCPS, and (36.1%, 0%, and 0%) of jobs missed their deadlines under the CRPS. This again demonstrates that the WCPS and the CRPS benefit from the idle time introduced by interface overheads and overestimations of jobs' execution times.

C. ARINC-653 Workloads

In this section, we evaluate the performance of our CSA implementation using ARINC-653 data sets obtained from an avionics system [13]. These data sets contain 7 harmonic workloads, each of which represents a set of domains (components) scheduled on a single processor, with each domain consisting of a set of periodic tasks. The descriptions of the workloads are available in the appendix of [5].

The evaluation goals are threefold: (1) to validate the effectiveness of the CSA implementation on real workloads; (2) to evaluate the relative performance of the PTPS, WCPS, and CRPS approaches under harmonic workloads and under different workload conditions; and (3) to quantify the impact of extra bandwidth available in the implemented interfaces (introduced by rounding up interface budgets, which are required in the implementation as the ARINC interface periods are fixed).



(a) High Interface Overhead (W. 1). (b) Low Interface Overhead (W. 2). (c) Minimum Number of Domains (W. 7).

Fig. 7: CDF Plots of $\frac{ResponseTime}{Deadline}$ For Different Types of ARINC Workloads.

Implementation of ARINC domains. Table I shows the interface overheads introduced for all 7 workloads. We ran each workload for 10 one-minute runs. The obtained results are then averaged across the 10 runs.

W. ID	1	2	3	4	5	6	7
U_{RM}	0.460	0.570	0.560	0.450	0.583	0.465	0.020
U_W	0.378	0.511	0.481	0.389	0.537	0.426	0.003
U_{RM}- U_W	0.082	0.059	0.079	0.061	0.046	0.039	0.017

TABLE I: Interface Overheads in ARINC Workloads.

Experimental results and observations. Figure 7 shows the responsiveness ($\frac{ResponseTime}{Deadline}$) distribution of the three server designs for three representative types of workloads: (a) having a high interface overhead; (b) having a low interface overhead; and (c) having a minimum number of domains. The CDF plots for all the workloads can be found in [9]. We observed the following behaviors:

(1) The WCPS and CRPS approaches consistently outperformed the PTPS approach for all three types of workloads in terms of miss rates and responsiveness, as shown in Figure 7. Note that in this experiment, we did not include all sources of system overheads in the components' parameters; as a result, the computed interfaces cannot account for all system overheads, resulting in potential deadline misses.

(2) In terms of responsiveness, the CRPS approach achieves the greatest improvement over the PTPS and WCPS approaches when the interface overhead is high, as is illustrated in Figure 7a. Conversely, when the interface overhead is low, the responsiveness improvement is less, as is shown in Figure 7b. These behaviors can be explained by the fact that a higher interface overhead potentially leads to more available residual-capacity that can be utilized by the scheduler.

(3) In terms of deadline miss rates, the CRPS approach improves significantly over the other two approaches regardless of the interface overheads, as is shown in both Figure 7a and 7b. For example, when the interface overhead is high, the CRPS approach incurs no deadline misses whereas both PTPS and WCPS experience deadline misses. Similarly, when the interface overhead is low, the PTPS (WCPS) approach incurs a miss rate of at least 200 times (100 times) more than the CRPS approach.

(4) As is illustrated in Figure 7c, the PTPS, WCPS, and CRPS approaches show similar distributions of responsiveness (with some small differences due to different system interference during different runs). This is expected because all approaches should behave identically if the workload contains a single domain, as is the case for Workload 7.

We also examined the performance of the PTPS approach with respect to the individual interface overheads of different domains within a component. Table II shows a typical example of the interface overhead versus deadline miss rate of different domains. It can be observed from the experimental results that, in general, a domain with a lower interface overhead often incurs a higher miss rate and vice versa. However, the effect of interface overhead on the domain's miss rate is less prominent when using the CRPS approach. This is expected because in the CRPS approach, the domains with lower interface overheads (smaller extra budgets) are allowed to reclaim capacity from domains with higher interface overheads (larger extra budgets).

Dom. ID	2	4	6	3	5	1	Total
Overhead	0.000	0.002	0.004	0.006	0.012	0.035	0.059
DMR	0.844	0.400	0.459	0.000	0.141	0.001	0.222

TABLE II: Relation between Interface Overhead and Deadline Miss Ratio of PTPS in Workload 2.

V. Related Work

In terms of system architecture for compositional scheduling, only a few implementations exist and none of those considers the Xen virtualization platform. For example, Behnam et al. [14] and Heuvel et al. [15] provided an implementation of a CSF on VxWorks and on μC/OS-II, respectively. However, neither approach considered virtualization. Yang et al. [16] developed a two-level CSF for virtualization using the L4/Fiasco. This work differs from ours in several aspects: (1) it builds on L4/Fiasco, which has a different system architecture from that of Xen; (2) it does not provide different work conserving enhancements to the periodic server; and (3) its interface computation is not optimal as it assumes identical periods for all domains and is based on a lower-bound of the SBF instead of the actual SBF.

Hierarchical real-time scheduling frameworks (HSFs) for closed systems also have been implemented in different OS kernels (e.g., [17]–[21]). These approaches, however, are non-compositional. Further, they implement all levels of

the scheduling hierarchy within the same operating system. HSF implementations through virtualization also have been explored lately. For instance, [22] proposed a bare VMM which uses virtualization and dedicated device techniques with a fixed cyclic scheduling policy. Cucinotta et al. [23] used KVM with a hard reservation behavior variant of the Constant Bandwidth Server (CBS). Our work is different from these in that our architecture supports compositional scheduling, which they do not. Further, ours builds on Xen, which has a different system architecture from that of KVM. (Xen is a stand-alone hypervisor that lies between guest OS and the hardware, whereas KVM is integrated into the manager domain, see e.g., [24] and [25].)

In terms of server designs, the general idea of 'capacity reclaiming' has been explored earlier in other contexts. For instance, Lehoczky et al. [26] provided a 'slack stealing' algorithm that allows aperiodic tasks to steal slack from periodic tasks. Caccamo et al. [27] and Nogueira et al. [28] provided CBS algorithms that allow one server to 'steal' another server's budget under EDF scheduling. These approaches, however, do not support compositional scheduling. In addition, their 'reclaiming capacity' includes only idle budget due to an overestimation of tasks' execution times, whereas ours includes the idle budget due to interface overheads as well.

In terms of theoretical computation of server parameters for quantum-based platforms, the only existing technique we are aware of was developed by Yoo et al. [29]. That work, however, assumes a manually chosen bound on the server period, which cannot guarantee an optimal resource period. In this paper, we provide a method for computing the maximum optimal server period, thus avoiding such sub-optimality.

VI. Conclusion

In this paper we have presented CSA, an architecture with system support for compositional scheduling of real-time systems. CSA realizes the key concepts and important results of a PRM-based CSF within the Xen virtualization platform, bringing the benefits of existing CSF theory to practical application. We discuss several challenges faced in the development of CSA, and propose theoretical extensions and server design enhancements to address these challenges. We also present an extensive evaluation to demonstrate the utility and effectiveness of CSA in optimizing real-time performance. Our evaluation using both synthetic and avionics workloads shows significant improvements in terms of response time and interface utilization. Our implementation provides a number of scheduling policies; at the same time, it is modular and easily extensible with new server-based scheduling algorithms. CSA is released as open-source and is available at `http://sites.google.com/site/realtimexen`.

CSA currently supports only independent periodic CPU-intensive tasks running on a uniprocessor. We plan to extend it to support dependent tasks and multicore processors, which undoubtedly will present additional challenges.

References

[1] I. Shin and I. Lee, "Compositional Real-time Scheduling Framework with Periodic Model," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–39, 2008.

[2] A. Easwaran, M. Anand, and I. Lee, "Compositional Analysis Framework Using EDP Resource Models," in *RTSS*, 2007.

[3] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," in *RTSS*, 1986.

[4] A. Easwaran, I. Lee, I. Shin, and O. Sokolsky, "Compositional Schedulability Analysis of Hierarchical Real-Time Systems," in *ISORC*, 2007.

[5] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, "A Compositional Framework for Avionics (ARINC-653) Systems," *Tech Report MS-CIS-09-04*, 2009, University of Pennsylvania. [Online]. Available: http://repository.upenn.edu/cis_reports/898

[6] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *RTSS*, 1989.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *SOSP*, 2003.

[8] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Real-Time Virtualization Based on Hierarchical Scheduling," in *EMSOFT*, 2011.

[9] J.Lee, S.Xi, S.Chen, L.Phan, C.Gill, I.Lee, C.Lu, and O.Sokolsky, "Realizing Compositional Scheduling through Virtualization," *Tech Report MS-CIS-11-13*, 2011, University of Pennsylvania. [Online]. Available: http://repository.upenn.edu/cis_reports/955

[10] S. Xi, J. Lee, C. Lu, C. D. Gill, S. Chen, L. T. X. Phan, O. Sokolsky, and I. Lee, "RT-Xen: Real-Time Virtualization Based on Hierarchical Scheduling," http://sites.google.com/site/realtimexen/.

[11] B. Brandenburg and J. Anderson, "On the Implementation of Global Real-time Schedulers," in *RTSS*, 2009.

[12] T. Baker, "A Comparison of Global and Partitioned EDF Schedulability Tests for Multiprocessors," in *RTNS*, 2005.

[13] Avionics Electronic Engineering Commitee(ARINC), *ARINC Specification 653-2: Avionics Application Software Standard Interface: Part 1 - required services*. Aeronautical Radio INC, Maryland, USA, 2005.

[14] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards Hierarchical Scheduling on Top of VxWorks," in *OSPERT*, 2008.

[15] M. M. van den Heuvel, R. J. Bril, J. J. Likkien, and M. Behnam, "Extending a HSF-enabled Open-source Real-time Operating System with Resource Sharing," in *OSPERT*, 2010.

[16] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin, "Implementation of Compositional Scheduling Framework on Virtualization," *SIGBED Rev.*, vol. 8, pp. 30–37, 2011.

[17] J. Regehr and J. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *RTSS*, 2001.

[18] M. Danish, Y. Li, and R. West, "Virtual-CPU Scheduling in the Quest Operating System," in *RTAS*, 2011.

[19] Y. Wang and K. Lin, "The Implementation of Hierarchical Schedulers in the RED-Linux Scheduling Framework," in *ECRTS*, 2000.

[20] G. Parmer and R. West, "HIRES: a System for Predictable Hierarchical Resource Management," in *RTAS*, 2011.

[21] T. Aswathanarayana, D. Niehaus, V. Subramonian, and C. Gill, "Design and Performance of Configurable Endsystem Scheduling Mechanisms," in *RTAS*, 2005.

[22] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned Embedded Architecture Based on Hypervisor: the XtratuM Approach," in *EDCC*, 2010.

[23] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting Temporal Constraints in Virtualised Services," in *COMPSAC*, 2009.

[24] KVM Twiki, "Kernel-based virtual machine," https://twiki.cern.ch/twiki/bin/view/Virtualization/KVM.

[25] Xen.org Community Blog, "Xen at build a cloud day boston," http://blog.xen.org/index.php/2011/12/22/xen-build-a-cloud-day-boston/.

[26] J. Lehoczky and S. Ramos-Thuel, "An Optimal Algorithm for Scheduling Soft-aperiodic Tasks in Fixed-priority Preemptive Systems," in *RTSS*, 1992.

[27] M. Caccamo, G. Buttazzo, and D. Thomas, "Efficient Reclaiming in Reservation-based Real-time Systems with Variable Execution Times," *IEEE Transactions on Computers*, vol. 54, pp. 198–213, 2005.

[28] L. Nogueira and L. Pinho, "Capacity Sharing and Stealing in Dynamic Server-based Real-time Systems," in *IPDPS*, 2007.

[29] S. Yoo, Y.-P. Kim, and C. Yoo, "Real-time Scheduling in a Virtualized CE Device," in *ICCE*, 2010.