A Game-Theoretic Resource Manager for RT Applications

Martina Maggio, Enrico Bini, Georgios Chasparis, Karl-Erik Årzén
Lund University, Sweden

***Abstract*—The management of resources among competing QoS-aware applications is often solved by a resource manager (RM) that assigns both the resources and the application service levels. However, this approach requires all applications to inform the RM of the available service levels. Then, the RM has to maximize the "overall quality" by comparing service levels of different applications which are not necessarily comparable.**

In this paper we describe a Linux implementation of a game-theoretic framework that decouples the two distinct problems of resource assignment and quality setting, solving them where in the domain they naturally belong to. By this approach the RM has linear time complexity in the number of the applications. Our RM is built over the `SCHED_DEADLINE` Linux scheduling class.

I. Introduction

The problem of managing the computing resources among different competing applications is as old as multitasking operating systems [1], [2]. If applications have real-time constraints, such management must also account for them.

Together with resource assignment, an orthogonal dimension along which the application performance can be tuned is the selection of the *service levels* of adaptive applications. Service levels depend on the possible configurable features of applications and determine the delivered quality/accuracy. Examples are adjustable video resolutions or the amount of data sent through a socket channel to render a web page. Hence, a proper solution should include both the resource assignment and the service level setting.

The typical solution to this problem is the implementation of a *resource manager* (RM), which is in charge of:

- assigning the resources to each application;
- monitoring the resource usage and possibly adjusting the assignment based on the actual measurements;
- assigning the service levels to each application, so that an overall delivered quality is maximized.

A scheme of this approach is illustrated in Figure 1. In the figure, app_i is the i-th application, s_i is its service level, and v_i is a set of parameters that determines the amount of resources allocated to app_i. According to this view, the RM monitors the behavior of the applications by measuring some *sensor* f_i. Then, using all the values read, the RM sets both the service level s_i and the amount of resource v_i assigned to the applications. This choice is often made through a *centralized* optimization procedure [3], [4], [5], [6].

However, these centralized optimization-based RMs have the following weaknesses:

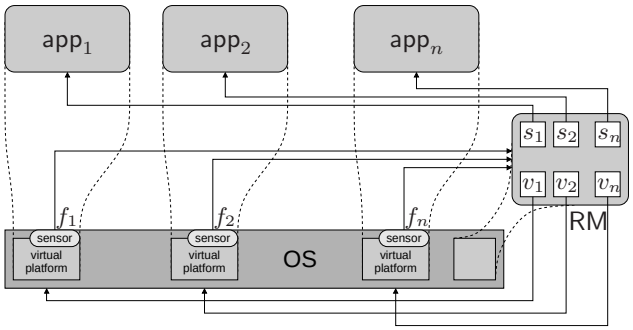

Figure 1. Centralized RM

- The complexity of the solvers used to implement the RM (such as ILP solvers) grows significantly with the number of applications. Hence, it is evidently impractical to have a RM that optimally assigns resources at the price of a high resource consumption by the RM itself.
- To enable the choice of a service level by the RM, the application must inform the RM about its available service levels and the expected consumed resource at each service level. However, this could infringe some Intellectual Property agreements or it could simply be too difficult to achieve.
- To enable a meaningful formulation of a cost function in the optimization problem, the RM must compare the quality delivered by different applications. This comparison is unnatural because the concept of quality is extremely application dependent.

Our proposal starts from the observation that the resource allocation and the service level assignment problems naturally belong to two different domains. The resource assignment has certainly to be made in a centralized way by the RM, since the same resource is shared by all applications. However, since the quality is application dependent, the service level assignment can be better made at application level, in a decentralized way. In fact, the application developer is the one who knows how to tune the application parameters in an opportune way. However, when should the application decide to change service level? What is the event that triggers such a decision at application level? Borrowing from game theory, we introduce an "application payoff" that is communicated by the RM to each application. Each application then should adjust its own service level to maximize the "revenue". This interaction leads to an equilibrium between the service levels of the applications, and their assigned re-

source. A final, not negligible, advantage of the proposed solution is that the RM has linear time complexity in the number of applications.

A. Related work

The problem of resource management was addressed by a multitude of research groups in the past. Without pretending to have made an exhaustive comparison, in this section we propose our selection of the most notable related results.

Several mechanisms for abstracting resources have been proposed in the past. Linux/RK [7] provided reserves of different types of resources (CPU cycles, memory pages, disk bandwidth, network bandwidth, etc.). However, to best of our knowledge, it is no longer maintained over current versions of the Linux kernel. LITMUS$^{\mathrm{RT}}$ [8] provides an experimental platform for testing and prototyping of multiprocessor real-time scheduling and synchronization algorithms. However, it does not aim to be included into the mainline Linux kernel. Hence, we decided to isolate the CPU resource by using the `SCHED_DEADLINE` scheduling class [9], because it specifically targets the inclusion into the Linux kernel. Supported by the positive comments received by the kernel maintainers [10] we believe that developing our framework on top of `SCHED_DEADLINE` may ease the portability of our framework in the future.

Many resource managers were proposed in the past. Some of them also applied game theory. Subrata et al. [11] applied game theory to balance the load in grid computing. Job arrivals are modeled by Poisson processes. The players are the machines that try to maximize their profit by accepting jobs. Wei et al. [12] proposed a game-theoretic method to allocate resources to incoming tasks. Tasks are assumed to be fully parallelizable. Similarly, Grosu and Chronopoulos [13] formulated the load balancing problem among different players as a non-cooperative game and then studied the Nash equilibrium. However, none of these papers considered the adjustment of application parameters.

Much of the research on resource management is based on the concept of feedback. Initial usage of feedback loops to control the allocated resources were independently developed by Lu et al. [14], Steere et al. [15], Eker et al. [16]. In these approaches quality adjustment was not considered.

Rajkumar et al. [3] proposed the QoS-based Resource Allocation Model (Q-RAM) to manage resources spanning over multiple dimensions. The resources are allocated so that the *total utility* is maximized with minimal QoS constraints. A similar approach was proposed by Sojka et al. [4].

In the ACTORS project, applications provide a table to the RM describing the required amount of CPU resources and the achieved QoS at each supported service level [5], [6]. In the multicore case, the amount of resources must be given for each individual partition. Then, the RM decides the service level of all the applications and how the partitions should be mapped to physical cores using a combination of ILP and first-fit bin-packing.

These approaches [3], [4], [5], [6] rely on the solution of an optimization problem that determines the amount of assigned resource and sets the service levels of all applications. Instead, we believe that the assignment of the service level can be better performed by the application itself, rather than at a centralized resource management level.

Contributions of the paper: In this paper we propose a game-theoretic resource management scheme for real-time applications. We designed the scheme and implemented it over the `SCHED_DEADLINE` scheduling class [9] on the Linux kernel. The paper presents both the theoretical analysis and the implementation of the framework. More details on the theoretical analysis and the proofs can be found in [17]. Experiments with single and multicore platforms are shown to validate the approach.

II. Game-theoretic model

The overall framework is illustrated in Figure 2. A set $\mathcal{I}$ of n *applications* are competing among each other for resources. Since we allow applications to dynamically join or leave, the number n is not constant over time. The resource is managed and allocated by a *resource manager* (RM) making sure that the overall allocated resource does not exceed the available one. Naturally, the set of applications along with the resource manager constitutes a finite set of decision makers, usually called *players/agents*. Both the set of applications and the resource manager are assumed to act independently. In game-theoretic terminology, the players can take *actions* that are described in details below.

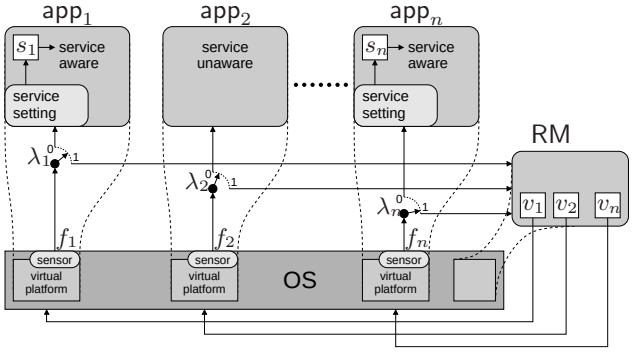

Figure 2. Game-theoretic resource management framework.

Each application $\mathsf{app}_i \in \mathcal{I}$ may change its *service level* $s_i \in \mathcal{S}_i$ (examples of service levels are: the accuracy of an iterative optimization routine, the details of an MPEG player, etc.). The service level s_i is an internal state of application app_i. Hence, it is written/read by app_i only. The set of service levels $\mathcal{S}_i$ is equipped with a partial ordering "$\geq$" that translates the concept of quality: if $s_i \geq s'_i$ then the perceived quality delivered to the user by application app_i when running at level s_i is not smaller than the quality when running at level s'_i. The partial order structure is the one that

best responds to multidimensional quality settings [3]. We denote by $s = (s_1, ..., s_n)$ the profile of service level of all applications evolving within the Cartesian product $\mathcal{S} \triangleq \mathcal{S}_1 \times ... \times \mathcal{S}_n$. The framework also allows applications that do not adjust their service level (app_2 in Figure 2).

The resource manager RM manages the available resource by allocating it to the applications. This allocation is made by reserving a dedicated *virtual platform* $v_i \in \mathcal{V}$ to the application app_i. A virtual platform is an abstraction for the use of a specific resource, where the word "virtual" is used with respect to computational capabilities. The set of virtual platforms (VP) is also equipped with a partial ordering "$\geq$", meaning that if $v_i \geq v_i'$ the quality delivered by any application app_i executing over v_i cannot be lower that the quality delivered when running over v_i' (for example, if a virtual platform is abstracted by its bandwidth, then this ordering will simply be inherited from the bandwidth ordering). All possible allocations of virtual platforms is $\mathcal{V}^n = \mathcal{V} \times \cdots \times \mathcal{V}$ (n times). However, not all allocations are feasible on the available physical platform and we denote by $\overline{\mathcal{V}} \subseteq \mathcal{V}^n$, the set of *feasible* virtual platforms. Hence the action of the RM will be the selection of $v = (v_1, ..., v_n)$ in $\overline{\mathcal{V}}$.

The goal of the proposed resource allocation framework is to find, for all application $\mathsf{app}_i \in \mathcal{I}$, a matching between the service level s_i set by application app_i and the virtual platform v_i assigned by the RM to the application app_i. The quality of this matching is defined by the following function.

Definition 2.1: The quality of the matching between a service level s_i and a virtual platform v_i of the application app_i is defined by the *matching function* $f_i : \mathcal{S}_i \times \mathcal{V} \to \mathbb{R}$ with the following properties:

- if $|f_i(s_i, v_i)| \leq \delta$, then the matching is *perfect*;
- if $f_i(s_i, v_i) < -\delta$, then the matching is *scarce*;
- if $f_i(s_i, v_i) > \delta$, then the matching is *abundant*;

with δ being a system parameter (in the experiments we use $\delta = 0$).

The perfect matching between s_i and v_i describes a situation in which the application has exactly the right amount of resource v_i when it runs at service level s_i. A scarce (resp., abundant) matching describes the situation when either increasing v_i or decreasing s_i (decreasing v_i or increasing s_i) is needed to move toward the perfect matching.

We stress that, differently than in the centralized management (Figure 1), the service levels are internal states of the applications, while the virtual platforms $\{v_1, \ldots, v_n\}$ belong to the RM space (as illustrated in Figure 2). Hence, no player can have a complete knowledge of the matching function f_i. In fact, the matching function is only measured during run-time. In Section IV we will describe our implementation of this measure-based function. The only properties that we require from any implementation of f_i are that

1) $\forall s_i,\ f_i(s_i, 0) < -\delta$, that is, the matching must certainly be scarce if no resource is assigned;
2) $s_i \geq s_i' \Rightarrow f_i(s_i, v_i) \leq f_i(s_i', v_i)$, if an application lowers its service level (it requires less computation), then the matching function should increase;
3) $v_i \geq v_i' \Rightarrow f_i(s_i, v_i) \geq f_i(s_i, v_i')$, if the virtual platform is increased (in the sense of the ordering in $\mathcal{V}_i$), then the matching function should increase.

When $|f_i| > \delta$ some adjustment is needed to the service level s_i or to the virtual platform v_i. The *weight* $\lambda_i \in [0, 1]$, also depicted in Figure 2, determines the amount of correction made by each player:

- if $\lambda_i = 1$, then the correction is entirely made by the RM with a calibration of the virtual platform v_i;
- if $\lambda_i = 0$, then the correction *should be* entirely made by app_i with an adjustment of the service level s_i;
- intermediate values of λ_i correspond to a combined correction made by both the application and the RM.

We remark that in our framework, the correction of the service level is left entirely to the application programmer. The RM simply informs the applications about the correction that is needed in order to satisfy the timing constraints of the applications.

Although we are interested in the general problem of managing all types of resources here only the CPU time is managed. Not surprisingly, however, in our experiments in Section V we are able to show that also memory-intensive applications can be controlled by managing the amount of allocated computations. Hence, from now on, we assume that the virtual platform v_i is simply abstracted by the speed of a virtual processor dedicated to app_i, and the set of feasible virtual platforms then is

$$\overline{\mathcal{V}} = \left\{ v \in [0, 1]^n : \sum_{i=1}^{n} v_i \leq m \right\} \tag{1}$$

with m being the number of available cores. If $v_i = 0$ then the virtual processor is halted, if $v_i = 1$ it provides the full speed of a single core.

Below, we introduce a learning procedure under which the applications and the RM adapt to possible changes in their "environment" (variations in the actions of the other players or variations in the number of applications) trying to maximize their own utility.

A. Adaptation of the virtual processors by the RM

To simplify the implementation, the RM updates the bandwidth $\tilde{v}_i = {}^{v_i}/_{m}$ normalized with the number of cores m. The corresponding vector of normalized resources is denoted by $\tilde{v} \triangleq [\tilde{v}_i]_i$ and lives in the *unit simplex of dimension* n, denoted $\widetilde{\mathcal{V}}$, i.e.,

$$\widetilde{\mathcal{V}} \triangleq \left\{ \tilde{v} \in [0, 1]^n : \sum_{i=1}^{n} \tilde{v}_i = 1 \right\}.$$

Note that the value of the actual bandwidth $v_i(t)$ may exceed its allowable values $[0, 1]$. Formally, in order for $v_i(t)$ to always be within $[0, 1]$, the updated value of normalized resources, $\tilde{v}_i(t+1)$, in (2) needs to be projected within $[0, {}^1/_m]$.

For the sake of clarity, we skip this projection and we briefly present the analysis for those cases where the resulting allocation of resources satisfies $v_i(t) \in [0,1]$ at all times t. The analysis for the more general (projected) dynamics can be found in [17].

When $t = 0$, the RM simply divides the amount of assignable bandwidth equally among the applications. At each time $t = 1, 2, ...$ and for each application $i \in \mathcal{I}$, the RM assigns resources according to the following rule:

1) it measures the performance[1] $f_i(t)$;
2) it updates the virtual platform $\tilde{v}_i$ as follows:

$$\tilde{v}_i(t+1) = \tilde{v}_i(t) + \epsilon_{\mathsf{RM}}(t)\Big(-\lambda_i f_i(t) + \sum_{j=1}^{n} \lambda_j f_j(t)\tilde{v}_i(t)\Big), \quad (2)$$

where $\epsilon_{\mathsf{RM}}(t)$ is a step-size sequence;
3) it computes the original value of bandwidth by

$$v_i(t+1) = m\tilde{v}_i(t+1),$$

4) it updates the time $t \leftarrow t+1$ and repeats.

In the forthcoming analysis, we will consider sequences $\epsilon_{\mathsf{RM}}(t)$ that diminish similarly to Robbins-Monro type of stochastic approximations [18]. For example, a candidate step-size sequence is $\epsilon_{\mathsf{RM}}(t) = 1/t+1$, which establishes an (implicit) averaging over time and recursively seeks for a root of the *observation* term, i.e., the multiplier of $\epsilon_{\mathsf{RM}}(t)$. As we shall see, these roots, which also correspond to rest points of the recursion (2), exhibit desirable properties. Notice that in the implementation we reset $\epsilon_{\mathsf{RM}}(t)$ every time the set of applications to be managed changes.

Note that according to recursion (2), we expect that v_i increases when app_i performs poorly compared to the others (when $\lambda_i f_i(t)$ is small compared to $\sum_{j=1}^{n} \lambda_j f_j(t)\tilde{v}_i(t)$). Consequently, at a rest point of (2), the performance of each application will be close to the weighted average performance, establishing a form of *fairness*.

B. Adaptation of the service level by the application

The RM provides information to all applications to guide their selection of a proper service level. To explain how any application app_i adjusts its service level, we introduce the following simple example.

Let us assume that the set of service levels of any app_i is $\mathcal{S}_i = [\underline{s}_i, \infty)$, where $\underline{s}_i > 0$ is the lowest possible service level of the application. At a given service level s_i, the application needs to execute sporadic jobs, each one with execution time $C_i = \alpha_i s_i$, with α_i being an application dependent constant. Since the application runs over a v_i-speed virtual processor, its *job response time* R_i, that is the time elapsing from the start time to the finishing time of a job, simply is

$$R_i = \frac{C_i}{v_i} = \frac{\alpha_i s_i}{v_i} \quad (3)$$

[1]As stated in its definition, f_i is a function of the service level s_i and the virtual platform v_i. However, here we intentionally hide this dependency and report only the dependency on time t, since the RM only measures a value over time.

Assuming that each job of the application has a *soft deadline* D_i, that denotes the expected job response time, then an example of a matching function f_i is:

$$f_i = \frac{D_i}{R_i} - 1 \quad (4)$$

which, given (3), can also be written as:

$$f_i = \frac{D_i v_i}{\alpha_i s_i} - 1 = \beta_i \frac{v_i}{s_i} - 1, \quad (5)$$

with $\beta_i = \frac{D_i}{\alpha_i}$ being an application dependent constant. This expression satisfies the required properties of f_i.

Let us assume that at time t the RM measures the matching function $f_i(t)$ of Eq. (4) and it discovers it is not zero. In response to this deviation from the equilibrium the RM sets the virtual platforms according to Eq. (2). *How should then application* app_i *act in order to bring the next value of the matching function,* $f_i(t+1)$*, equal to zero?* By setting $f_i(t+1)$ equal to zero in (5), we get

$$s_i(t+1) = \beta_i v_i(t+1).$$

However, setting the next service level $s_i(t+1)$ according to this rule is an open-loop technique that relies on a careful estimation of β_i, which may be unavailable. Moreover, it requires the knowledge of v_i within the application. From the (possibly non-zero) measurement $f_i(t)$ at time t we can actually estimate β_i, that is

$$\beta_i = (1 + f_i(t))\frac{s_i(t)}{v_i(t)}$$

so that the service level update rule becomes

$$s_i(t+1) = (1 + f_i(t))\frac{v_i(t+1)}{v_i(t)} s_i(t). \quad (6)$$

The above recursion may exhibit large incremental differences, $s_i(t+1) - s_i(t)$, which may lead to instability. Hence, we introduce a smoother update rule for the service level s_i that exhibits the same stationary points as those of (6), by setting:

$$s_i(t+1) = s_i(t) + \epsilon_i(t)\Big(\underbrace{(1 + f_i(t))\frac{v_i(t+1)}{v_i(t)}}_{\text{RM space}} s_i(t) - s_i(t)\Big) + \epsilon_i(t) z_i(s_i(t)), \quad (7)$$

with $\epsilon_i(t)$ governing the adaptation rate of app_i. The correction term $\epsilon_i(t) z_i(s_i(t))$ corresponds to the projection necessary to bring the RHS of (7) within $\mathcal{S}_i$. In words, we should expect that s_i decreases when $f_i < 0$ and $v_i(t+1)/v_i(t) < 1$, i.e., when the application is doing poorly and the assigned resources have been decreased. The term $v_i(t+1)/v_i(t)$ provides a look-ahead information to the application about the expectation over future available resources.

Also, in (7) it can be observed that the only quantity that needs to be communicated by the RM to app_i is the factor highlighted by the underbrace. After app_i receives such a

quantity, it can adjust its service level without knowing all the information the RM used to compute it.

The interesting property of adjusting virtual platform and service levels according to (2) and (7) is that we can prove their convergence, as demonstrated in the next section.

III. Convergence Analysis

Below, we summarize the convergence analysis of the framework. The details of these results can be found in [17].

Before proceeding with the convergence analysis, we first rewrite recursion (2) for each $i = 1, \ldots, n$, as follows:

$$\tilde{v}_i(t+1) = \tilde{v}_i(t) + \epsilon_{\mathsf{RM}}(t) g_{\mathsf{RM},i}(s(t), \tilde{v}(t)) \tag{8}$$

where

$$g_{\mathsf{RM},i}(s, \tilde{v}) \triangleq -\lambda_i f_i(s_i, v_i) + \sum_{j=1}^{n} \lambda_j f_j(s_j, v_j) \tilde{v}_i,$$

where $v_i = m\tilde{v}_i$, $i \in \mathcal{I}$. Also, we write recursion (7) as follows:

$$\begin{aligned} s_i(t+1) = s_i(t) + \\ \epsilon_i(t) g_{\mathsf{app},i}(s_i(t), \tilde{v}_i(t), y_i(t)) + \epsilon_i(t) z_i(s_i(t)), \end{aligned} \tag{9}$$

where

$$g_{\mathsf{app},i}(s_i, \tilde{v}_i, y_i) \triangleq (1 + f_i(s_i, v_i)) \frac{\tilde{v}_i}{y_i} s_i - s_i,$$

and $y_i(t)$ is a delayed version of $v_i(t)$ updated according to

$$y_i(t+1) = y_i(t) + \epsilon_i(t) \left(\tilde{v}_i(t) - y_i(t) \right). \tag{10}$$

The reason for augmenting the state with $y \triangleq [y_i]_i$ is to deal with the different time indices in (7).

The asymptotic behavior of the overall recursion (8), (9) and (10) can be characterized as follows:

Proposition 3.1: If $\epsilon_i(t)$ is proportional to $1/t+1$ then, the overall recursion (8), (9) and (10) is such that the sequence $\{(s(t), \tilde{v}(t), y(t))\}$ converges[2] to some limit set of the Ordinary Differential Equation (ODE):

$$\begin{pmatrix} \dot{s} \\ \dot{\tilde{v}} \\ \dot{y} \end{pmatrix} = g(s, \tilde{v}, y) + \begin{pmatrix} 0 \\ z(s) \\ 0 \end{pmatrix}, \tag{11}$$

where $g \triangleq ([g_{\mathsf{app},i}]_i, [g_{\mathsf{RM},i}]_i, \tilde{v} - y)$ and $z \triangleq [z_i]_i$ is the minimum force required to drive s back in $\mathcal{S}$. Finally, if $E \subset \mathcal{S} \times \widetilde{\mathcal{V}} \times \widetilde{\mathcal{V}}$ is a locally asymptotically stable set in the sense of Lyapunov[3] for (11) and $(s(t), \tilde{v}(t), y(t))$ is in some compact set in the domain of attraction of E, then $(s(t), \tilde{v}(t), y(t)) \to E$.

Proof: Proof is based on Theorem 2.1 in [18]. ∎

A similar result can also be stated for constant step-size sequences which are sufficiently small.

The above proposition relates the asymptotic behavior of the overall discrete-time recursion with the limit sets of the ODE (11). Since the *stationary points*[4] of the vector field g are invariant sets of the ODE (11), then they are also candidate attractors for the recursion. In the following sections, we analyze the convergence properties of the recursion with respect to the stationary points of the ODE (11).

A. Stationary points

Lemma 3.1 (Stationary Points): Any stationary points of the vector field g, say (s^*, v^*, y^*), satisfies all the following conditions:

(C1) $\sum_i \lambda_i f_i(s_i^*, v_i^*) \tilde{v}_j^* = \lambda_j f_j(s_j^*, v_j^*)$,
(C2) $(f_j(s_j^*, v_j^*) = 0) \vee ((s_j^* = \underline{s}_j) \wedge (f_j(s_j^*, v_j^*) \leq 0))$,
(C3) $y_j^* = \tilde{v}_j^*$.

for all $j = 1, \ldots, n$. Furthermore, the set of stationary points is non-empty.

Proof sketch: Existence of stationary points follows from Brower's fixed point theorem [20, Corollary 6.6]. The coordinates of the stationary points follow from the definition of $g_{\mathsf{RM},i}(\tilde{v}, s)$. More details can be found in [17]. ∎

The above proposition states that at a stationary point, app_i is either performing sufficiently good (i.e., $f_j(s_j^*, v_j^*) = 0$), or it performs poorly but its service level s_i cannot be decreased any further (i.e., $f_j(s_j^*, v_j^*) \leq 0$, $s_j^* = \underline{s}_j$).

Note that any pair (s, v) in the domain for which $f_i(s_i, v_i) = 0$ is a stationary point of the vector field g. This multiplicity of stationary points complicates the convergence analysis of the dynamics, however, in several cases, uniqueness of the stationary point can be shown. The following subsections identify a few such special cases.

1) Fixed Service Levels: The following result characterizes the set of stationary points when applications do not update their service levels.

Proposition 3.2: Consider the matching function defined by (4). For some given $s_i \geq \underline{s}_i$ for all i, and under either one of the following hypotheses:[5]

(H1) *β_i / s_i is sufficiently small for all i;*
(H2) *$\lambda_i \frac{\beta_i}{s_i} \approx \lambda \frac{\beta}{s}$ for all i and for some constants $\lambda \in (0, 1)$, $\beta > 0$ and $s > 0$;*

then the vector field g has a unique stationary point which satisfies:

$$\tilde{v}_j^* \approx \frac{\lambda_j}{\sum_i \lambda_i}, \tag{12}$$

for all $j = 1, \ldots, n$.

Proof sketch: It follows from Proposition 4.2 in [17]. More details can be found in [17]. ∎

In other words, Proposition 3.2 states that, if the service level s_i of each application i is sufficiently large (compared to the available resource v_i), (H1), or applications have similar characteristics, (H2), then the unique stationary point of the dynamics assigns resources to applications proportionally to their normalized weights.

[2] By $x(t) \to A$ for a set A, we mean $\lim_{t \to \infty} \operatorname{dist}(x(t), A) = 0$.

[3] See [19, Definition 3.1].

[4] The stationary points of an ODE $\dot{x} = \bar{g}(x)$ are defined as the points in the domain D for which $g(x) = 0$.

[5] Here we abuse notation by interpreting the symbol "$\approx$" as "sufficiently close in the Euclidean distance."

The above proposition also provides an answer to how the stationary point changes with respect to the weight parameters $\{\lambda_i\}$. In particular, from (12), we conclude that if either one of the hypotheses (H1) or (H2) is valid, then the percentage of resources v_j^* of application j will increase at the stationary point if λ_j is also increased.

2) Non-fixed Service Levels: Note that the conclusions of Proposition 3.2 continue to hold when the service levels are also adjusted based on (7) as long as the corresponding hypotheses are satisfied for all $s \in \mathcal{S}$. The following proposition identifies one such case.

Proposition 3.3: Consider the matching function defined by (4). If hypothesis (H1) holds for all $s \in \mathcal{S}$, then the overall dynamics (11) exhibits a unique stationary point $(s^*, \tilde{v}^*)$ such that $s_i^* = \underline{s}_i$ and $\tilde{v}^*$ satisfies property (12).

Proof: If hypothesis (H1) holds for all $s_i \geq \underline{s}_i$ and $\tilde{v} \in \widetilde{\mathcal{V}}$, then the conclusions of Proposition 3.2 apply. Furthermore, $s_i^* = \underline{s}_i$ for all $i = 1, \ldots, n$, since $f_i(s_i, v_i) < 0$ for all $s_i \geq \underline{s}_i$ and all $v \in \overline{\mathcal{V}}$. ■

In other words, Proposition 3.3 states that there is a unique stationary point of the overall dynamics when the RM is not able to provide sufficient amount of resources to all applications. In such cases, we should expect that applications tend to decrease their service levels as much as possible in order to improve their performance. Thus, the unique stationary point will correspond to $s_i = \underline{s}_i$ for all i. Furthermore, the allocation of $\tilde{v}^*$ will satisfy property (12), since the update of the service levels does not improve significantly the performance of the applications and the matching functions remain negative at all times.

As we have already pointed out, in the more general case where hypothesis (H1) does not hold, there is a multiplicity of stationary points including any pair (s^*, v^*) for which $f_i(s_i^*, v_i^*) = 0$.

B. Local Asymptotic Stability & Convergence

The following proposition characterizes locally the stability properties of the stationary points under the hypotheses of Propositions 3.2–3.3.

Proposition 3.4 (LAS): Under the hypotheses of either Proposition 3.2 or 3.3, the unique stationary point of the dynamics (11) is a locally asymptotically stable point in the sense of Lyapunov.

Proof sketch: It is proved by defining an opportune Lyapunov function. More details can be found in [17]. ■

From Proposition 3.1, we conclude that the stationary points of the ODE (11), which satisfy the hypotheses of Proposition 3.4, are possible local attractors of the overall recursion.

IV. Implementation

The game-theoretic resource management framework illustrated in Figure 2 has been implemented in C over Linux. Below we describe in detail all its components:

- the application interface,
- the sensing infrastructure[6],
- the virtual platform implementation, and
- the resource manager.

A. Application interface

The goal of the entire resource management framework is to provide the most appropriate amount of resource to all applications. It is here assumed that applications are composed by some time sensitive portions of code, called *jobs*. For example, in a media encoder/decoder a job is the encoding/decoding of a chunk of data. Applications are requested to inform the RM about the desired duration of each job. Below we report a template of the application code. To ease the presentation, we omit some details such as variable type declarations or command line parsing.

```
int main(int argc, char* argv[]) {

  myself = app_registration();
  /* example: only one type of job to be completed in 1 ms */
  app_set_jobtypes(myself, 1, {1000000});
  ...
  while(!finished) {
    id = signal_job_start(myself, type);
    adjust[type] = get_performance(myself, type);
    /* body of the specific type job. If service aware, it should
       modify its resource requirement by adjust[type] */
    do_work(/* parameters */);
    signal_job_end(myself, id);
  }
  ...
  app_termination(myself); /* free shared memory */
}
```

When an application wants to register with the RM, it calls `app_registration`. The only purpose of this call is the initialization of a shared memory area (whose pointer is returned by the call), which is used to store information about the application run-time behavior. Notice that the management of the shared memory is completely transparent to the application programmer.

Then it communicates (through `app_set_jobtypes`) the number of job types (`NUM_JOBS` in the code above) that it can generate during its run-time and the expected response times (`job_times`) for each type of job. Depending on the application, the developer may use this value to distinguish among different application functionalities. For example, a video encoder could have several different types of job, corresponding to the encoding of the frame categories. If the number of job types or their desired response times change at run-time, it is possible to provide this new information by invoking `app_set_jobtypes` again.

As the application runs, it is asked to signal the start and the end of a job. This signaling actions are performed by invoking respectively `signal_job_start` and `signal_job_end`, providing as parameter the index `i` of the job type. Within the job, the first action is the

[6]The code for the application interface and sensing infrastructure is released and available at http://github.com/martinamaggio/jobsignal.

invocation of the function `get_performance` for this type of job. This function, which is computed by the monitoring infrastructure, returns a measurement of the service level adjustment (i.e. the underbrace expression in Eq. (7)) required to achieve a perfect matching between the service level and the virtual platform. We underline that the of the application code does not necessarily need to be open, since the body of the job can be an external function linked from some library.

B. Sensing infrastructure

To enable a prompt adjustment of the resource allocation and the service level, the progress of the applications is constantly monitored. This is performed by computing the matching function f_i. As explained in Section II, the matching function f_i should be close to zero when the resources v_i allows application app_i to run smoothly, should be negative if app_i needs more resources, and positive whenever it has more than enough.

We implemented the matching function as in (4), due to the proved convergence properties (see Section III). The job response time, R_i, of (III) is measured using the the job start times and finishing times signaled by the applications.

The application response time R_i of (4) can be computed as the maximum or the average of the job response time over a time window which can be properly specified. In the experiments we use the average over the last 10 jobs.

C. OS support

The basic support needed from the OS is the capability of implementing virtual processors. Our implementation uses the `SCHED_DEADLINE` scheduling class [9] of Linux. We chose this mechanism because of its in-depth integration within the Linux kernel[7]. The `SCHED_DEADLINE` scheduling class fully supports multicore architecture, processor affinities, and all other features of modern Linux schedulers. `SCHED_DEADLINE` combines EDF scheduling with hard CBS (Constant Bandwidth Servers) [21]. Each task is scheduled in its own server with adjustable parameters.

The `SCHED_DEADLINE` scheduling class allows changing the reservation of a task or a group of tasks at run-time. In particular, we use the `sched_setscheduler2` function to set the following parameters of virtual processors:

- the period of the CPU reservation;
- the deadline;
- the time budget.

The period of the CPU reservation in our experiments is chosen to be equal to the period of the resource manager.

D. Resource Manager

The resource manager implements the mechanism to distribute the resource described by equation (2). Specifically, it reads the matching function for each active applications and updates the virtual platforms. The RM runs as a forever loop within a `SCHED_DEADLINE` server with period of 1 msec and budget of 10 μsec.

When assigning the virtual processors, the RM takes care of the following crucial aspects:

1) Not to exceed the maximum available computing capacity determined by the number of cores minus some safety margin which should allow for the execution of the resource manager itself and for other operating system routines.
2) When assigning the single virtual processors, not to exceed the individual processor capacity minus some safety margin to leave space to code which has to execute in the core.
3) When adjusting the virtual processors, to start by decreasing and then by increasing the bandwidth of the virtual processors, to avoid that the assigned resource exceeds the available one during some transients.

Below is shown a stub of the RM code.

```
int main(int argc, char* argv[]) {
  /* extract command line parameters and initialize */
  while(true) {
    num_applications = update_applications(apps);
    if /* reset rule */ reset = 1;
    else reset = 0;

    compute_vp(apps, num_applications,
      iterations, reset); /* equation (2) */
    rescale_vp(apps, num_applications);
    set_vp(apps, num_applications); /* call the scheduler */
    ++iterations; /* t++ in equation (2) */
  }
}
```

Since the function `compute_vp` may produce some non-feasible bandwidth assignments, the function `rescale_vp` ensures that the assigned bandwidths do not exceed the available capacity. Finally, the RM allows a reset rule, which re-initializes the adaptation of the VPs by resetting the time counter used to compute $\epsilon_{\mathsf{RM}}(t)$ of (2). In our implementation, the reset is done when the number of applications changes.

V. Experimental evaluation

We describe below experiments settings. The experimental platform is a quad-core running at 3.4GHz. In single core experiments three cores are switched off before the experiment starts, while in multicore experiments all cores are online. The RM always allocates 90% of the available cores to the set of running applications.

To simulate service-aware applications, we developed a synthetic test application, which performs some computation and uses some memory, depending on the service level s_i. Such an application has one type of job with deadline D_i, which is executed in a forever loop. Each job performs $a_i^{\mathsf{cpu}} s_i + b_i^{\mathsf{cpu}}$ mathematical operations and uses $a_i^{\mathsf{mem}} s_i +$

[7]We are using branch mainline-dl that can be found at https://github.com/jlelli/sched-deadline/ and the scheduling support added with the provided library that can be retrieved at https://github.com/gbagnoli/rt-app/.

b_i^{mem} bytes of memory to simulate computation and memory usage, respectively. Hence, applications with a large a_i^{cpu} and a_i^{mem} are more service-sensitive than applications with a_i^{cpu} and a_i^{mem} close to zero. The application service level is adapted according to (7) and the small constant value of ϵ_i determines the amount of correction in the service level s_i. As illustrated in Eq. (7), a large ϵ_i produces a quick adaptation together with significant fluctuations, while small values produce a smoother evolution at the price of a slower adaptation. All applications parameters (D_i, ϵ_i, a_i^{cpu}, b_i^{cpu}, a_i^{mem}, and b_i^{mem}), which determine the application behavior and its capacity to adapt, are read from command line. This enables, for example, the coexistence of fully service-aware applications together with service-unaware ones.

Single core, VP convergence: In the first scenario, we validate Proposition 3.2 by testing the convergence of the virtual processors v_i to the expression of Equation (12), which is proportional to the application weight. To produce the condition of hypothesis (H1) of Prop. 3.2, all applications keep a constant high service level (overload condition). All applications use no memory and perform 30K mathematical operations without adapting service level (b_i^{cpu}= 30K, a_i^{cpu}= a_i^{mem}= b_i^{mem}= 0). Also they all have a job deadline D_i = 100 msec. The convergence is tested in correspondence to application arrivals (at time 2, 2, and 4), and application termination (at time $t = 3$). The results are shown in Figure 3. The virtual processors are scheduled on a single core, we use as maximum assignable bandwidth $U^{\mathsf{ub}} = 0.9$. At time

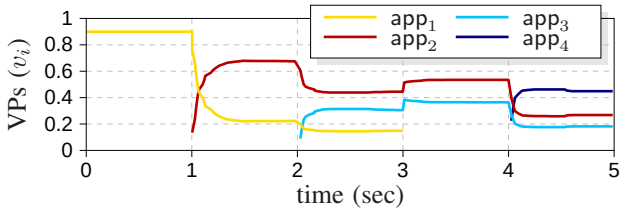

Figure 3. Convergence of the virtual processors.

$t = 0$, app$_1$ with weight $\lambda_1 = 0.1$ is started. Being the only one running the RM assigns all the available resource to it. At time 2, a second application with weight $\lambda_2 = 0.3$ starts. We observe that the the new values of the VPs tend exactly to the theoretical values $v_1 = U^{\mathsf{ub}}\lambda_1/\sum_i \lambda_i = 0.225$ and $v_2 = U^{\mathsf{ub}}\lambda_2/\sum_i \lambda_i = 0.675$. At time 3, app$_3$ with $\lambda_3 = 0.2$ enters and the VPs reach again the theoretical values dictated by (12). Finally, at time 3 app$_1$ terminates, while at time 4 a new one with weight $\lambda_4 = 0.5$ enters. We observe that in all these circumstances the VPs tend to the expected theoretical values since Proposition 3.2 is satisfied.

Single core, adaptive and non-adaptive applications: In the next experiment, we tested the coexistence between service-aware and service-unaware applications. Again, the RM assigns VPs such that $U^{\mathsf{ub}} = 0.9$. All the applications have jobs that should complete in 10 msec. The values given to the application parameters can be found in Table I.

In Figure 4 we show the results of the experiment. At time

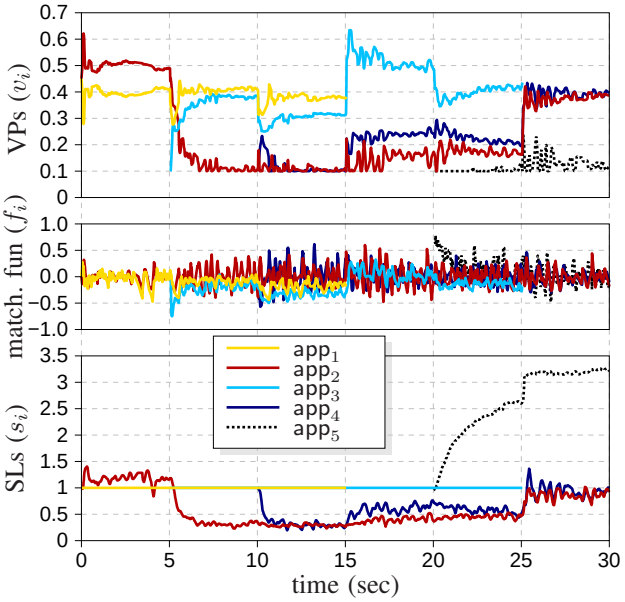

Figure 4. Adaptive and non-adaptive applications.

$t = 0$ two applications are started. The first one is "legacy" (or non-adaptive) while the second application can adapt its service level. The matching function f_i of the non-adaptive application settles to zero and the RM therefore stops assigning more resources to app$_1$. The adaptive application, app$_2$, takes the rest of the CPU and increases its service level until the amount of CPU received does not match its service level. At time $t = 5$ another legacy application is started, with a lower weight λ_3 than the previous non-adaptive one. Initially it is assigned a VP equal to 0.1 and its f_i is negative. This implies that the RM assigning to app$_3$ some amount of the CPU that drives its matching function to zero. This is done by penalizing app$_2$, which must reduce its service level to be satisfied with the VP it receives. The relationship between the two "legacy" application is determined by their weights, in fact, the newcomer receives less resources than the previous one.

At time $t = 10$ another adaptive application enters and adjust its service level to be satisfied with its VP. The two legacy applications are terminated respectively at time $t = 15$ and at time $t = 25$ while another adaptive application is started at time $t = 20$. This last adaptive application has a lower workload with respect to the others. Due to this and to the termination of the two legacy applications, it is able to

	a_i^{cpu}	b_i^{cpu}	a_i^{mem}	b_i^{mem}	ϵ_i	λ_i
app$_1$	0	1000	0	0	-	0.5
app$_2$	1000	0	0	0	0.1	0.5
app$_3$	0	1000	0	0	-	0.2
app$_4$	1000	0	0	0	0.1	0.2
app$_5$	100	0	0	0	0.01	0.8

Table I
ADAPTIVE AND NON-ADAPTIVE APPLICATIONS: SETTINGS.

increase its service level. app_2 and app_4 react to the increase of VPs by raising the higher service level. Notice that all matching functions approach zero.

Multicore, SL and VP adaptation: In this experiment, we test the adaptation of both the service levels and the virtual processors over a multicore. We have 12 applications, each of them uses no memory and has $a_i^{\mathsf{cpu}} = 200$ and $b_i^{\mathsf{cpu}} = 30$. All the applications have jobs that should complete in 10 msec. The RM assigns VPs such that $U^{\mathsf{ub}} = 3.6 = 0.9 \times 4$, since the machine has all the four cores enabled.

Application app_1 is assigned a weight $\lambda_1 = 0.8$, app_{12} is assigned $\lambda_{12} = 0.2$, while the other ten applications are all assigned weight $\lambda_{2..11} = 0.5$. The virtual processor assignment v_i, the matching function f_i, and the service levels s_i are all shown in Figure 5 from top to bottom.

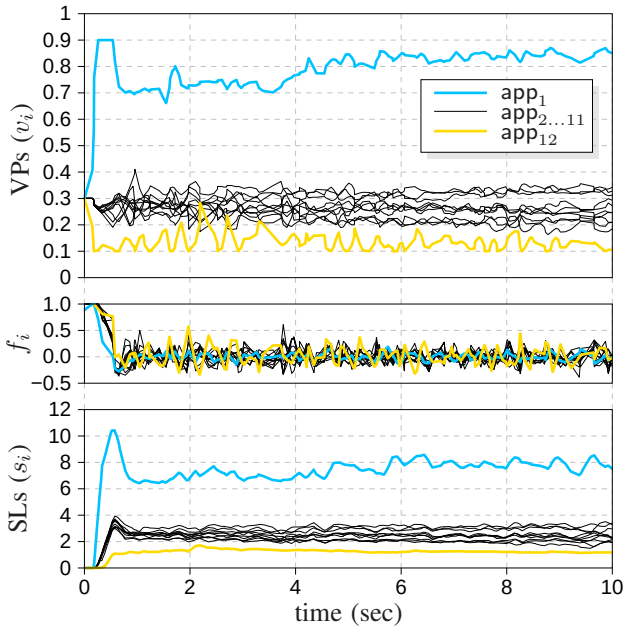

Figure 5. CPU intensive applications.

We can observe that the RM and the applications are capable to implicitly coordinate so that the assigned virtual processors are proportional to the service level of all applications, with approximately the same proportionality factor (being the applications the same). The difference in the VP allocation is only determined by the different weights λ_i.

Multicore, SL and VP adaptation with memory: In this experiment, we tested the RM with some memory-intensive applications, with parameters reported in Table II. All the applications have jobs that should complete in 10 msec.

Figure 6 shows, from top to bottom, the virtual processors, the matching functions, and the service levels. The initial service level is 1. We observe that the usage of memory does not impact the execution requirement much. The two applications which are using a large amount of memory are assigned some share of CPU and they still perform well (their matching functions are close to zero). The weights still determine the relative allocation of the resource and the values of the service levels. In fact, applications with higher weight receive more resource and, consequently can execute at a higher service level.

	a_i^{cpu}	b_i^{cpu}	a_i^{mem}	b_i^{mem}	ϵ_i	λ_i
$\mathsf{app}_{1..8}$	1000	300	1MB	10KB	0.05	0.5
app_9	1000	300	0	10KB	0.05	0.2
app_{10}	1000	300	10MB	10KB	0.05	0.2
app_{11}	1000	300	0	10KB	0.05	0.8
app_{12}	1000	300	10MB	10KB	0.05	0.8

Table II
MULTICORE WITH MEMORY-CONSUMING APPLICATIONS: SETTINGS.

RM *overhead:* The final experiment is dedicated to the evaluation of the time consumed by the resource manager. For this experiment we ran the RM 1000 times with a number of applications from 1 to 24. In Figure 7 (top) the measured run-time of the RM is shown. Notice that the resource manager itself measures its overhead by tracking its execution time, containing both the sensors measurement retrieval and its own execution. Also we draw by solid lines the maximum and minimum and by a dashed line the average of the run-time.

During the experiments, we realized (not surprisingly) that most of the time taken by the RM was spent in accessing the shared memory structure that was used for the communication between the RM and the applications. Since the shared memory is protected by a semaphore, as the number of applications grows the run-time of the RM can become much larger due to the higher risk of being blocked.

In the bottom of the figure we plot the RM overhead without the time taken to access the shared memory (still in μsec, labeled "no shared mem"). We claim that, in the future, a more efficient implementation of the RM (possibly developed in kernel space) that limits the usage of shared memory can provide an overhead that is much similar to the bottom figure. This experiment confirms one the must interesting feature of our resource management framework that is the

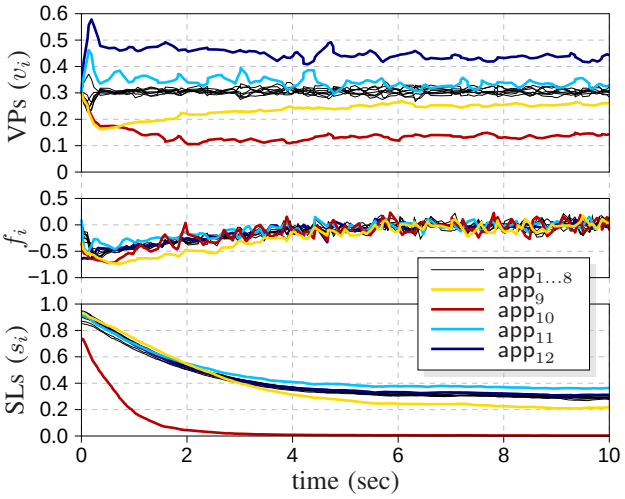

Figure 6. Applications using memory.

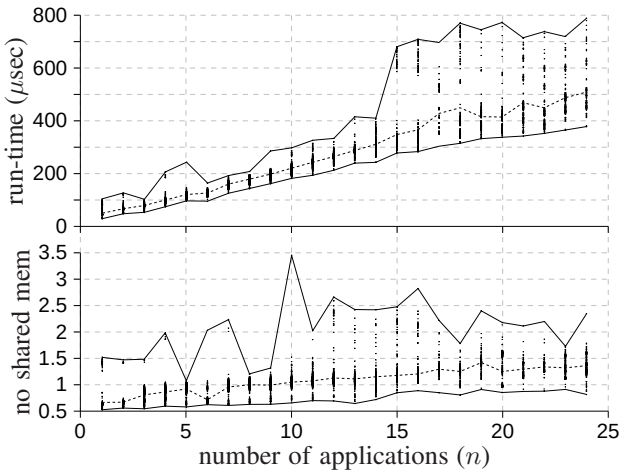

Figure 7. Overhead of the RM.

linear complexity of the resource manager.

VI. Conclusions and future works

In this paper we proposed a game-theoretic resource manager for real-time applications. The behavior of each application app_i is measured by its matching function f_i. Depending on the application-dependent weight λ_i some of the needed correction is made at the resource management level. The remaining correction is made by the application itself that adjusts its service level s_i. Thanks to the decoupling of this two operations, the RM has linear time complexity in the number of applications. Thw entire framework has been implemented in Linux, using the `SCHED_DEADLINE` scheduling class to isolate applications. Extensive experiments were performed to validate the theory.

In the future we plan to continue the tuning of the RM, to improve its implementation and to extend it with other management policies such as energy saving.

References

[1] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," in *Proceedings of the Spring Joint Computer Conference*, vol. 21, May 1962, pp. 335–344.

[2] E. G. Coffman Jr and L. Kleinrock, "Computer scheduling methods and their countermeasures," in *Proceedings of the Spring Joint Computer Conference*. New York, NY, USA: ACM, Apr. 1968, pp. 11–21.

[3] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for QoS management," in *Proceedings of the IEEE Real Time System Symposium*, 1997.

[4] M. Sojka, P. Píša, D. Faggioli, T. Cucinotta, F. Checconi, Z. Hanzálek, and G. Lipari, "Modular software architecture for flexible reservation mechanisms on heterogeneous resources," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 366–382, 2011.

[5] E. Bini, G. C. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzén, R. Vanessa, and C. Scordino, "Resource management on multicore systems: The ACTORS approach," *IEEE Micro*, vol. 31, no. 3, pp. 72–81, 2011.

[6] K.-E. Årzén, V. Romero Segovia, S. Schorr, and G. Fohler, "Adaptive resource management made real," in *Proc. 3rd Workshop on Adaptive and Reconfigurable Embedded Systems*, Chicago, IL, USA, Apr. 2011.

[7] S. Oikawa and R. Rajkumar, "Portable rk: a portable resource kernel for guaranteed and enforced timing behavior," in *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*, 1999, pp. 111–120.

[8] B. Brandenburg, A. Block, J. M. Calandrino, U. Devi, H. Leontyev, and J. H. Anderson, "LitmusRT: A status report," *Proceedings of the 9th Real-Time Linux Workshop*, pp. 107–123, 2007.

[9] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *Proceedings of the 11th Real-Time Linux Workshop (RTLWS)*, Dresden, Germany, October 2009.

[10] P. Zijlstra and S. Rostedt, "Comments on: `[RFC][PATCH 00/16] sched: SCHED_DEADLINE v4`," Available at http://thread.gmane.org/gmane.linux.kernel/1278219, Apr. 2012.

[11] R. Subrata, A. Y. Zomaya, and B. Landfeldt, "A cooperative game framework for QoS guided job allocation schemes in grids," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1413–1422, Oct. 2008.

[12] G. Wei, A. V. Vasilakos, Y. Zheng, and N. Xiong, "A game-theoretic method of fair resource allocation for cloud computing services," *The Journal of Supercomputing*, vol. 54, no. 2, pp. 252–269, Nov. 2010.

[13] D. Grosu and A. T. Chronopoulos, "Noncooperative load balancing in distributed systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1022–1034, 2005.

[14] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, "Design and evaluation of a feedback control EDF scheduling algorithm," in *Proceedings of the 20th IEEE Real Time Systems Symposium*, Phoenix (AZ), U.S.A., Dec. 1999, pp. 56–67.

[15] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A feedback-driven proportion allocator for real-rate scheduling," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, Feb. 1999.

[16] J. Eker, P. Hagander, and K.-E. Årzén, "A feedback scheduler for real-time controller tasks," *Control Engineering Practice*, vol. 8, no. 12, pp. 1369–1378, Jan. 2000.

[17] G. Chasparis, M. Maggio, K.-E. Årzén, and E. Bini, "Distributed management of CPU resources for time-sensitive applications," in *Proceedings of The 2013 American Control Conference*, 2013, available as Technical Report at http://www.control.lth.se/Publication/7625.html.

[18] H. J. Kushner and G. G. Yin, *Stochastic Approximation and Recursive Algorithms and Applications*, 2nd ed. Springer-Verlag New York, Inc., 2003.

[19] H. Khalil, *Nonlinear Systems*. Prentice-Hall, 1992.

[20] K. Border, *Fixed Point Theorems with Applications to Economics and Game Theory*. Cambridge University Press, 1985.

[21] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 1998, pp. 4–13.