A general control-theoretical methodology for runtime resource allocation in computing systems

Alberto Leva, Alessandro Vittorio Papadopoulos, and Martina Maggio

Abstract— **Control theory is emerging as a source of solutions for many problems in the computing systems domain, and in particular for resource allocation. However, the techniques proposed to date in the literature, do not successfully address the difficulty of devising allocation schemes that are general enough, provide stability guarantees, and can be parametrised and managed by system administrators, or computing systems' practitioners at large. This paper proposes a two-level time-varying control scheme for the allocation of a generic resource, and proves its stability. In addition, the paper discusses how the scheme can be made acceptable by computing systems administrators, and how to tune the involved parameters. A simulation campaign is reported, validating the aforementioned claims.**

I. Introduction

Control theory has recently provided a wide variety of contributions to the design and implementation of computing systems [2]–[4], [14], [18]. In fact, the computing system community has recognised the need for adaptation, and the capabilities of feedback control [5] to realise it. Examples of the so obtained solutions are complementing an operating system scheduler with admission policy to enforce deadlines [10], [20], [21] or assigning reservation periods [1], [15], [19]. To appreciate the generality of the problem, the interested reader can refer, e.g., to works like [14].

The idea of using control-theoretical design methodologies is particularly promising in resource allocation. In fact, the literature contains several attempts to tackle the problem by means of feedback, spanning from fuzzy logic [13] to reinforcement learning [4]. However, an unsolved challenge is to find a general scheme that can be applied to any type of resource. Also, the found solution should be easy to configure. If parameters are involved, practitioners who are not supposed to have expertise in the decision-making domain of choice, should be able to select the correct parameters to obtain a prescribed behaviour of the system.

In the area of resource allocation, a lot of attention was devoted to the CPU, that is the main computing resource. The presented work builds on previous contributions. In [11] we proposed an analysis of why controlling computing systems is different from applying control theory to any other domain. We applied the principles to the specific problem of "feedback scheduling", and developed a two-level control scheme that allocates the CPU to the running tasks in a computing system [7]. The implementation of the scheme in an embedded device with a real kernel was subject of studies [12] and benchmarking proved that the so-developed control system is useful for practical applications. However, there still were some open issues in the stability proof for the mentioned scheme, that are the main contribution of this paper.

The paper is organised as follows. Section II presents the methodology for the control system development, while Section III delves into the stability proof. Section IV discusses an important matter for practical application, which is how to translate requirements that are domain specific into "entities" of the devised control solution, comprehensible for system administrators. Results are presented in Section V, while Section VI concludes the paper.

II. Methodology

This section reviews the aspects of the theory on feedback resource allocation that are relevant to understand the content of this paper and its contribution. The term feedback resource allocation is related to the term "feedback scheduling" and refers to the application of feedback-based techniques to the assignment of computing resources to running processes. In fact, it is possible to see any computing system as a set of finite resources that tasks consume in order to get their jobs done. In the following, the two terms application and task are interchangeable: a task is some computation that require resources to terminate.

Among these resources, the most common are CPU, memory and disk. However, in principle it is possible to formulate the problem of allocating energy and power to the running applications. Another resource that in principle could be considered is a generic form of instructions that are executed on the machine. In this paper, we will allocate a generic resource. The approach that we present works under the following assumptions:

1) the running applications are competing for the resource that we are allocating, some of them can require more and some other may need less of this resource;
2) the number of tasks is time-varying, however the task pool changes sporadically with respect to the allocation time scale;
3) there is a physical limitation to the amount of resource that is assignable to the competing tasks, for example because the amount of CPU is limited and two applications can not consume the same CPU simultaneously;
4) there is no a priori assumption on how the resource will be used by the applications.

The first three assumptions simply describes the normal operation of a computing system, where the distributed resources are limited. The fourth assumption is a call for generality. In fact, even though with some resources it could

A. Leva, A.V. Papadopoulos are with Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy. {leva,papadopoulos}@elet.polimi.it

M. Maggio is with Lund University, Department of Automatic Control, Lund, Sweden. martina.maggio@control.lth.se

be in principle possible to assume some kind of usage pattern, this is not feasible for every possible resource. Therefore, in our system, we will not assume anything about the resource usage. In control-theoretical terms, we will model every possible difference between the nominal resource usage and the effective one as a disturbance, and try to reject these disturbances to obtain the desired behaviour. In other words, we address here "resource-resource" problems, i.e., those in which the distribution itself of the resource is also the ultimate objective, as opposite to "resource-work" ones, where the goal is to guarantee an effective *use* of the resource (see the discussion on the matter in [8, Chap. 8]).

The general approach to feedback resource allocation is to take an existing controller (scheduler, memory controller or disk controller) and realise a form of adaptivity by changing some of its parameters (the amount of memory allocated to each task or the number of active memory areas per application). Some contributions devise control strategies for resource allocation starting from a fully functional system and tweaking it to make it more robust to changes in the operating conditions [1], [6], [9], [15]. This is for example the case of adaptive web servers, where the percentage of requests to be served is dynamically chosen based on the load [17]. If one removes the admission controller, that is allowing a certain percentage of requests to enter the system and denying the access to some other requests, the system will work exactly in the same way, but have a higher probability of be overloaded and fail in serving the requests.

However, the main difference between this research and the *corpus* of literature works is that here we design a resource allocator exactly in the form of a controller. There is no fully functional object to be augmented with control strategies. If one removes the controller, the system simply does not work. Our main objective is to achieve a certain distribution of the resource, based on the applications' needs also in the presence of unmodeled dynamics. For example, in the case of the CPU, this means that we want to maintain the CPU distribution of three different tasks also when the operating system is partially using the CPU for its own purposes.

A. A model for resource consumption

The first step to build a resource allocator is to devise a model of the tasks response to the amount of resource that they receive. Since we want to abstract from the specific characteristics of each task, the model is trivial, but captures the phenomena of interest. At each controller intervention k, a task i is given a *burst* of resource, denoted by $b_i(k)$. For the CPU this value is often called bandwidth while for memory it could be called pages. When its share of the resource is consumed by the task, the resource is relieved and becomes available for the other tasks that are waiting for it. Moreover, if the task does not need the resource anymore, it can release it voluntarily. If some other operation, for example the code of a critical section, prevents the task from releasing the resource, prolonging its consumption time.

As anticipated before, every behaviour that is unmodeled is seen as a disturbance, $\delta b_i(k)$. Denoting by $\tau_{t,i}(k)$ the amount of resource that is allocated to the task at the resource allocator intervention k, the model for the i-th task can be written as

$$\tau_{t,i}(k) = b_i(k-1) + \delta b_i(k-1), \quad (1)$$

where i varies in the interval $1,\ldots,N$ and N is the number of tasks. It is necessary to clarify the role of $\tau_{t,i}(k)$, which is the amount of time that the task had for its use of the resource between the $(k-1)$-th and the k-th intervention of the allocator — the $(k-1)$-th round of allocation. This means that our control variable is the time allocated to the task and we can measure the effective time that the application has spent using the resource between two distinct control actions. This time is not necessarily equal to the control variable due to disturbances. We want the resource distribution to follow some determined division among the running applications.

To extend the model to the entire task pool, we should consider that every task receives a burst at the beginning of the allocation round, leading to:

$$\mathbf{P} : \begin{cases} \boldsymbol{\tau}_t(k) = & \mathbf{b}(k-1) + \delta\mathbf{b}(k-1) \\ \tau_r(k) = & \sum_{i=1}^{N} \tau_{t,i}(k) \\ \tau(k) = & \tau(k-1) + \tau_r(k-1) \end{cases} \quad (2)$$

where τ_t is the vector of allocations, τ_r is the time between two subsequent scheduler interventions, or *round* and τ is the system time. We also want to constrain the time between two subsequent allocator intervention to a specific value.

B. A control structure for resource consumption

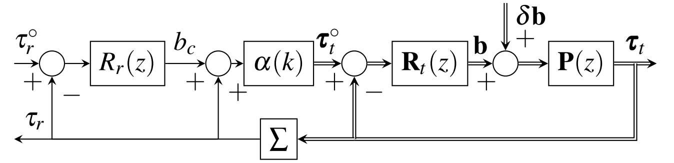

Fig. 1: Block diagram representing the proposed approach for resource allocation.

We recall that our control system' purpose is to select $\mathbf{b}$ to enforce a specific distribution of the resource among the tasks. We present one possible solution for the resource allocation, that was devised in the context of task scheduling but is actually more general and applies to other resources as well. The control structure that we chose is shown in Figure 1. Block $\mathbf{P}(z)$ represents the plant to be controlled, or the task pool. Block $\mathbf{R}_t(z)$ is devoted to computing the vector of task bursts $\mathbf{b}$ in such a way that the vector $\boldsymbol{\tau}_t$ of tasks' time usages for the resource follow a set point $\boldsymbol{\tau}_t^\circ$. This set point is obtained by partitioning the measured round duration τ_r according to the (possibly time-varying) vector $\alpha(k)$, the elements of which sum to one. An idle task can be introduced to manage the case in which the total resource requested is less than one.

The bursts are additively corrected by the *burst correction* b_c output by block $R_r(z)$, so that τ_r follow its set point τ_r°. This is important because if for example some task gets blocked and stop consuming the resource, $\mathbf{R}_t(z)$ can still keep the resource time usage for the others, but the round

duration set point is lost [7]. Note that the proposed discrete-time control is single-rate, and the rate is dictated by the outer level. The task allocator, therefore, performs its computations once per round, which is somehow a peculiarity with respect to the existing literature, where the allocator intervenes more frequently and often unnecessarily. The proposed control structure can effectively manage task scheduling when the control blocks are very simple. We chose a diagonal integral one for $\mathbf{R}_t(z)$ and a single input single output PI for $R_r(z)$. Simulations and executions on real hardware were discussed extensively to testify the validity of this approach in the CPU case [12].

However, one main issue has yet to be solved with respect to the state of the art [7]. The scheme of Figure 1 is time-varying, owing to the natural use of $\alpha(k)$ as a means to govern the resource distribution, and a stability analysis is in order. The primary contribution of this paper is to prove the stability of the proposed scheme. Section III is devoted to this matter. Another contribution of this paper is to simplify the use of the proposed control strategy by system administrators, that usually have little (if any) knowledge on control theory. Even if the matter has not been stressed before, this is really important in order for the contribution to get accepted by the specific community that it addresses. This is treated in Section IV.

III. Stability analysis

This section analyses the proposed allocation scheme as for stability, accounting for its time-varying nature. Such an analysis constitutes the main contribution of this paper. It is also worth noticing that most of the addressed cases (think to the CPU scheduling one as a representative example) exhibit no or negligible parameter uncertainty, which allows – from a practical standpoint – to avoid delving in complex robustness discussions.

The general control structure to be analysed is shown in Figure 2, where the controllers within the internal and the external loop are represented by the MIMO block $\mathbf{R}_u(z)$ and the SISO one $R_s(z)$, respectively. The disturbance is not shown since it is not relevant for the present analysis.

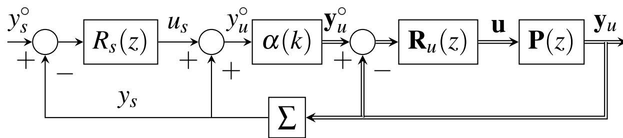

Fig. 2: Block diagram of the proposed control structure in general, evidencing the controllers in the internal and external loop.

Let us consider the $N \times N$ MIMO system with input $\mathbf{y}_u^\circ$ and output $\mathbf{y}_u$ of Figure 2, and suppose that the process $\mathbf{P}(z)$ and the regulator $\mathbf{R}_u(z)$ are diagonal, with identical elements. Let then $(A_p, B_p, C_p, 0)$ and (A_r, B_r, C_r, D_r) be realisations of the (SISO) diagonal elements of the process and the regulator respectively. Hence, the closed loop matrices $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ of the system with input $\mathbf{y}_u^\circ$ and output $\mathbf{y}_u$ are block diagonal with elements

$$A = \begin{bmatrix} A_p - B_p D_r C_p & B_p C_R \\ -B_r C_p & A_r \end{bmatrix}, \quad B = \begin{bmatrix} B_p D_r \\ B_r \end{bmatrix},$$
$$C = \begin{bmatrix} C_p & 0 \end{bmatrix}, \qquad D = 0,$$

therefore $\mathbf{A} = diag(A, A \dots A)$. Also, suppose that each A is Schur.

Let $\alpha(k)$ a time-varying vector of length N such that $\sum_i \alpha_i(k) = 1, \forall k$. Thus the SISO system with input y_u° and output y_s with state space realisation $(\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \mathfrak{D})$ is obtained as

$$\mathfrak{A} = \mathbf{A}, \quad \mathfrak{B} = \mathbf{B}\alpha(k), \quad \mathfrak{C} = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \mathbf{C}, \quad \mathfrak{D} = 0.$$

Finally, let consider the system with input u_s and output y_s, thus introducing a new positive feedback, yielding to the state space realisation $(\mathfrak{A}_s, \mathfrak{B}_s, \mathfrak{C}_s, 0)$

$$\mathfrak{A}_s = \mathfrak{A} + \mathfrak{B}\mathfrak{C}, \quad \mathfrak{B}_s = \mathfrak{B}, \quad \mathfrak{C}_s = \mathfrak{C}.$$

Recalling that the state space vector of the system has the form

$$\mathbf{x} = \begin{bmatrix} x_{p,1} & x_{r,1} & \cdots & x_{p,N} & x_{r,N} \end{bmatrix}'$$

where, clearly, $x_{p,i}$ and $x_{r,i}$ are the states of the i-th process and of its controller, we can introduce a state space transformation T such that the new state space is

$$\xi = T\mathbf{x} = \begin{bmatrix} x_{p,1} - \sum_{i=2}^{N} x_{p,i} & x_{r,1} - \sum_{i=2}^{N} x_{r,i} & x_{p,2} & x_{r,2} & \cdots & x_{p,N} & x_{r,N} \end{bmatrix}'$$

and the resulting state space matrices are obtained as $\tilde{\mathfrak{A}}_s = T^{-1}\mathfrak{A}_s T$, $\tilde{\mathfrak{B}}_s = T^{-1}\mathfrak{B}_s$, $\tilde{\mathfrak{C}}_s = \mathfrak{C}_s T$. In particular, it is worth evidencing that

$$\tilde{\mathfrak{A}}_s = \left[\begin{array}{c|cccc} A + BC & 0 & \cdots & \cdots & 0 \\ \hline BC\alpha_2(k) & A & 0 & \cdots & 0 \\ BC\alpha_3(k) & 0 & A & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ BC\alpha_N(k) & 0 & \cdots & 0 & A \end{array}\right],$$

where the eigenvalues of $\tilde{\mathfrak{A}}_s$ do not depend on $\alpha(k)$, and therefore are fixed in time.

As a consequence, if $R_s(z)$ stabilises $A + BC$, and recalling that A is Schur, then the time variance induced by $\alpha(k)$ does nothing but altering the inputs of asymptotically stable systems that do not take part into the external loop.

Summarising, it was proven that the proposed control structure can be safely used for resource allocation. Vector α is used to govern the resource distribution via the internal controller, and the set point y_s° to decide – via the external controller – "how frequently" the allocator has to regain control. No action on those two inputs can jeopardise the control system stability.

IV. THE SYSTEM ADMINISTRATOR'S INTERFACE

Proving the stability of the closed-loop system ensures that even a non-expert cannot alter the correct behaviour of the controller, when choosing the parameters and the set points of the resource allocator. This opens the possibility of providing to system administrators a usable interface to the scheme, without requiring them to deal with control concepts.

In fact, system administrators are used to categorise the tasks in batch or interactive, period or sporadic, preemptable or non-preemptable and so on. In the computer science domain, the problems on different task sets are often solved independently and the solution that works for one type of tasks does not cover the others. The presented framework treats all task types the same way, i.e., with the same control law. The distinction becomes evident only from how the resources are assigned to those tasks, i.e., from parameters' and set point profiles' selection. Therefore, it is necessary to ease the logic for said selection, to make it accessible also to somebody that does not have a background in control engineering. In the following, we suppose that α can be varied freely, thus the analysis of Section III is a prerequisite.

To address the problem, we will here split it into three different parts. First, one should select the controllers' structuring. Second, a convenient set point and α generation mechanism needs devising. Last, the parametrisation of the two entities just mentioned has to be addressed. In the authors' opinion, the aforementioned subproblems are naturally attributed to the *three* involved professionals: the control engineer, the computer engineer, and the system administrator.

A. Control structuring

Intuitively, given the necessity of minimising the resource allocation overhead, simple controllers should be selected for the internal and external loops. Based on previous works, there are some typical choices. The internal controller can either be an integral or a deadbeat regulator, while the external controller can be a PI, a pure proportional, or a deadbeat one. Using integral and proportional blocks results in the lowest orders, while deadbeat blocks may require more stages, but allow to shape the set point responses. Attempts were made also with model predictive controllers, but this introduces a computational burden and should be used only when strictly necessary [16].

B. Set point and α generation

With the introduced control structure, the choice of τ_r° and α can enforce any requisite of fairness, timeliness and so forth [11]. The computer engineer can in some sense "pre-configure" a system for its administrator, reasoning on four basic task types.

1) Tasks with periodic deadlines: Let T_i be the period of a task, and W_i its workload in the period (i.e., every T_i time units the task must receive the resource for W_i time units). The *accumulated* resource time in the period needs thus following the trapezoidal profile of Figure 3, the start of which is triggered by a periodic interrupt, while the end can be required by the task once its per-period workload is accomplished — all the major operating systems provide primitives for such a signalling.

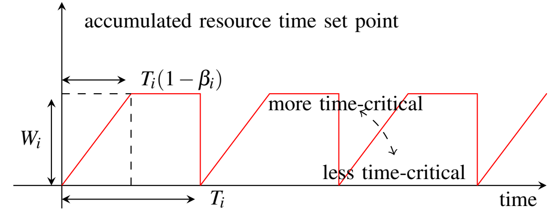

Fig. 3: Accumulated resource time set point for periodic tasks.

This means that at the beginning of each period the task will be allotted a *tentative* α component, denoted by $\widehat{\alpha}_i$, given by

$$\widehat{\alpha}_i(k) = \frac{W_i}{(1-\beta_i)T_i\tau_r^\circ(k)}, \quad 0 \leq \beta_i < 1, \tag{3}$$

while the meaning of "tentative" has to do with preserving the unity sum of α, and is explained later on. The same $\widehat{\alpha}_i$ will be reset to zero by the task itself, to which – correctly, for the scheduler's generality – any decision is devoted on what to do if T_i expires before W_i is accomplished. Note that $\widehat{\alpha}_i$ remains constant unless τ_r° is changed while $\widehat{\alpha}_i \neq 0$, which is correct. Note also that a β_i close to one requests the resource time "as soon as possible", which diminishes in general the probability of missing a deadline. Building on such an idea, one can probably hope in the future to have some tuning knob to pass with continuity to non real-time through soft up to hard real-time constraints.

2) Tasks with a single deadline: Such tasks are managed in the same way as those with periodic deadlines, except that only one period, of length equal to the desired task duration, is triggered at its arrival. Also the meaning of W and β are the same.

3) Tasks without deadlines: This is the typical situation for interactive tasks, such as desktop applications. However, one can define *for tasks without deadlines* a *real* – i.e., not integer nor quantised – priority range, say from zero (lowest) to one (highest), and obtain the corresponding tentative α elements as

$$\widehat{\alpha}_i(k) = \alpha_{\min,i} + p_i(\alpha_{\max,i} - \alpha_{\min,i}), \quad 0 \leq p_i \leq 1, \tag{4}$$

where p_i is the mentioned "priority", the quotes indicating that its effect on the actually allotted resource time is definitely more direct and interpretable than it would be if the term was given the traditional meaning.

4) Event-triggered tasks: This is the case of many services, think for example of the mouse driver with the CPU resource. The idea is that when awakened the task gets a certain tentative α element, which then decays to zero at a given rate, i.e., as

$$\widehat{\alpha}_i(k) = \widehat{\alpha}_i(k-1) \cdot a_i^{-(k-k_{0,i})}, \quad 0 < a_i < 1, \tag{5}$$

and is reset to the initial value ($\widehat{\alpha}_{i0}$ in the following) when a new awakening event is triggered, typically via an interrupt, that simply has to reset $k_{0,i}$ to the current time index. Note that this is the only case in which $\widehat{\alpha}_i$ can in general undergo variations over each round. If this is not acceptable for any

reason, one could for example allot a "small" fixed resource share to the task, and consider it blocked when not awakened.

Once all the tentative components $\widehat{\alpha}_i$ are available, α is simply obtained by rescaling them (uniformly, as "relative task importances" are already substantiated by the choices of the sections above) so that the sum be one. In the case of particularly critical periodic tasks a flag can be introduced to prevent their components from being rescaled. The computer engineer in charge of pre-configuring a system has essentially to set limits on the admissible values for parameters $\alpha_{\min,i}$ and $\alpha_{\max,i}$ and possibly narrow those for β_i, p_i and a_i, either system-wide or for some classes of tasks. Also, she has to decide how many tasks can be excluded from rescaling, and so forth. The idea is to set "safe" limits for the choices of the system administrator, when necessary. The interesting feature is that said limits can be clearly interpreted in terms of the used control scheme.

C. Parameter setting

Parameter setting is the role of the system administrator, and once confined as per the choices above, there should not be the possibility of provoking undesired behaviours, while a graceful degradation is expected (thanks to rescaling) in the case of overloading. To further ease the administrator's work, the computer engineer could also prepare some pre-defined parameter settings suitable for certain task types, and by means of profiling, those settings could be refined on the field. Here, the advantages of a control-theoretical design should become evident.

V. Experimental evaluation

In this section, we simulate the resource allocation to show how the allocator acts. The presented results are obtained from a PC-based simulator of how the resource is distributed in a microcontroller kernel, Miosix[1]. We chose to use the simulator because it allows us to emulate the behavior of a generic resource and is not limited to the CPU. Notice that these simulations are very realistic, given the particular nature of the problem and the almost absence of uncertainty sources.

The internal and external controllers used in the example are respectively of the I and PI type, expressed as

$$\mathbf{R}_u(z) = \text{diag}\left\{\left\{\frac{0.5}{z-1}\right\}_i\right\}, \quad R_s(z) = 2.5\frac{z-0.5}{z-1}, \qquad (6)$$

and are tuned as described in [12].

We test the system with a task pool, the characteristics of the tasks are summarised in Table I. The workload parameters represent how much resource time the applications would need. There are tasks of all the four defined types: for the two periodic ones the period, the workload and the value of β are given; the two batch ones are specified in the same way, except for the replacement of "period" with "duration" and the presence of an arrival time; the three prioritised ones are specified by the corresponding values of p; finally, the two event-based ones have an initial resource share and a decay rate as parameters, and are triggered at prespecified times. During the simulation, some priority modifications are

[1] `http://home.dei.polimi.it/leva/Miosix.html`.

ID	Periodic	period	workload	β	
1	`Tpe1`	50	0.5	0.5	
2	`Tpe2`	180	0.8	0.2	
	Batch	arrival	workload	duration	β
3	`Tba1`	100	60	300	0.1
4	`Tba2`	150	70	400	0.1
	Priority	priority			
5	`Tpr1`	0.1			
6	`Tpr2`	if($\tau < 250 \wedge \tau > 375$) 0.4, else 1.0			
7	`Tpr3`	0.2			
	Ev-trigg	in. share	dec. rate	trig. times	
8	`Teb1`	0.02	0.6	10, 20, 100, 280, 300	
9	`Teb2`	0.03	0.7	5, 15, 50, 80, 150, 400	

TABLE I: Task pool for the simulation example.

impressed to show the system's response. No other actions on the pool are shown to avoid too confusing a presentation, and since the purpose here is to show the effects of the proposed parametrisation scheme, care is taken in this example to avoid over-utilisation.

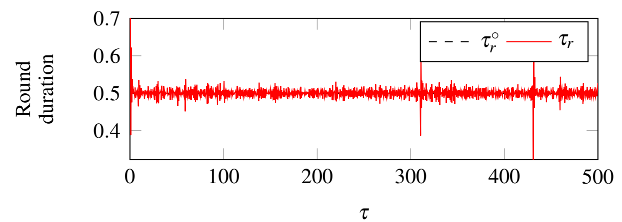

Fig. 4: Round duration control.

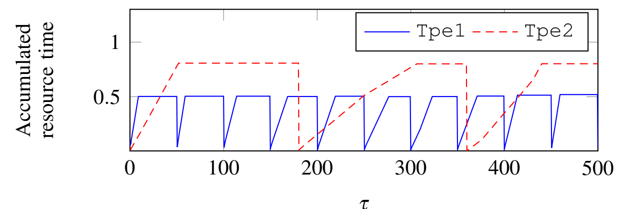

Fig. 5: Accumulated resource time for periodic tasks.

Figures 4–6, with vertical axes graduated in time units, report the results of a simulation run spanning 500 time units. In detail, Figure 4 shows the round duration, that apparently keeps its set point of 0.5 time units despite all the underlying pool-generated events and the $\delta\mathbf{b}$ disturbance, chosen so as to generate on average a 10% deviation of the actual from the allotted resource use for all the tasks.

Figure 5 shows the accumulated resource times for the periodic tasks, evidencing the effect of β. In this example no task is excluded from α rescaling, whence the observed (and desired if rescaling is not used) slope modifications. Finally, Figure 6 reports the resource shares for all the tasks in the pool, evidencing how the desired set points are met and mutual influences are dealt with thanks to the control scheme.

Summarising, the example evidenced that the proposed control structure can be parametrised by a system administrator in a comprehensible manner, as shown in Table I. Other phenomena, such as over-utilisation or other actions on the pool, are dealt with correctly by the resource allocator —

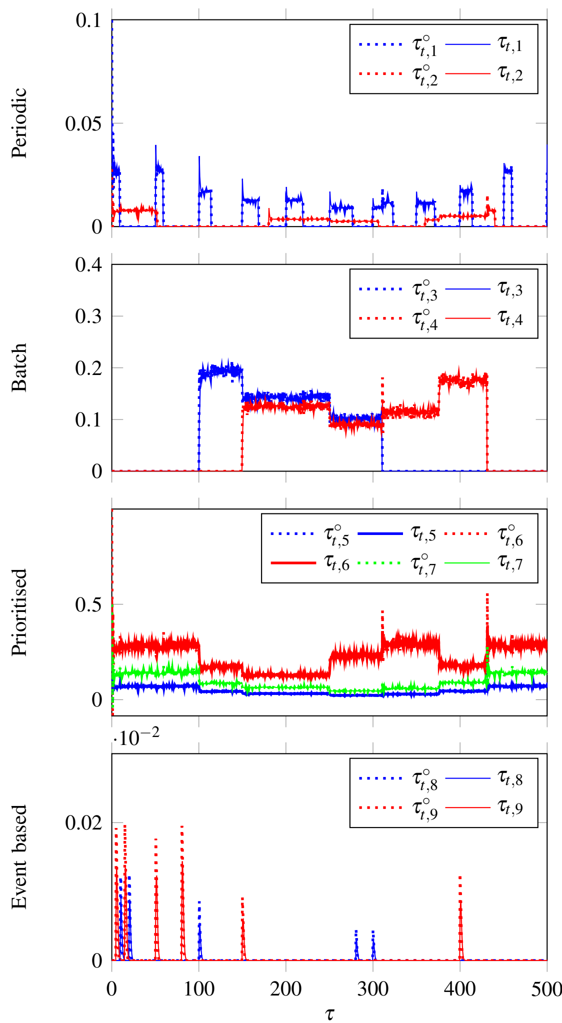

Fig. 6: Resource distribution.

thanks to the stability condition.

VI. Conclusion

This work discussed the problem of computing systems resource provisioning from a control-theoretical perspective. Building on previous work, a schema for resource allocation has been presented and the stability of the closed loop system has been proved. The stability analysis of the time-varying system is the most important contribution of this paper.

Another important point has been discussed: the parametrisation of the control scheme. In fact, it is often recognised that devising a control solution would solve computing systems problem. However, the lack of expertise in the control system synthesis is one of the main factors preventing these solutions from being effectively applied in real systems. Simulation studies for the CPU allocation confirmed the foreseen results. In future work, the same solution will be implemented for other resources.

Acknowledgment

This work was partially supported by the Swedish Research Council through the LCCC Linnaeus Center.

References

[1] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 71–80, 2002.

[2] F. Harada, T. Ushio, and Y. Nakamoto. Adaptive resource allocation control for fair qos management. *Computers, IEEE Transactions on*, 56(3):344–357, 2007.

[3] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004.

[4] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 39–50, Washington, DC, USA, 2008. IEEE Computer Society.

[5] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, jan 2003.

[6] D. Lawrence, G. Jianwei, S. Mehta, and L. Welch. Adaptive scheduling via feedback control for dynamic real-time systems. In *Proceedings of the IEEE International Conference on Performance, Computing, and Communications*, pages 373–378, 2001.

[7] A. Leva and M. Maggio. Feedback process scheduling with simple discrete-time control structures. *IET Control theory and applications*, 4(11):2331–2342, 2010.

[8] A. Leva, M. Maggio, A. V. Papadopoulos, and F. Terraneo. *Control-based operating system design*. Control Engineering Series. IET, Jun 2013.

[9] C. Lu, J. Stankovic, H. Sang, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Syst.*, 23(1/2):85–126, 2002.

[10] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Syst.*, 23:85–126, July 2002.

[11] M. Maggio and A. Leva. A new perspective proposal for preemptive feedback scheduling. *International Journal of Innovative Computing, Information and Control*, 6(10):4363–4377, 2010.

[12] M. Maggio, F. Terraneo, and A. Leva. Task scheduling: a control-theoretical viewpoint for a general and flexible solution. *ACM Transactions on Embedded Computing Systems*, May 2012. Accepted for publication.

[13] K. Mjelde. Fuzzy resource allocation. *Fuzzy Sets and Systems*, 19(3):239 – 250, 1986.

[14] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 289–302, New York, NY, USA, 2007. ACM.

[15] L. Palopoli, L. Abeni, and G. Lipari. On the application of hybrid control to cpu reservations. In *Proceedings of the Hybrid systems Computation and Control*, 2003.

[16] A. V. Papadopoulos, M. Maggio, S. Negro, and A. Leva. Enhancing feedback process scheduling via a predictive control approach. In *Proc. of the 18th IFAC World Congress*, volume 18, pages 13522–13527, 2011.

[17] A. Robertsson, B. Wittenmark, M. Kihl, and M. Andersson. Design and evaluation of load control in web server systems. In *Proceedings of the 2004 American Control Conference*, volume 3, pages 1980–1985, Boston, MA, June 2004. IEEE Control Systems Society.

[18] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. Mete: meeting end-to-end qos in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 13–24, New York, NY, USA, 2011. ACM.

[19] I. Song, S. Kim, and F. Karray. Stability analysis of feedback controlled reservation-based cpu scheduler. In *Proceedings of the 1st International ECRTS Workshop on Real-Time and Control*, pages 1880–1885, 2005.

[20] J. Stankovic, C. Lu, S. Son, and G. Tao. The case for feedback control real-time scheduling. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 11 –20, 1999.

[21] D. S. Swaminathan, D. R. Sahoo, S. Swaminathan, R. A-omari, M. V. Salapaka, G. Manimaran, and A. K. Somani. Feedback control for real-time scheduling. In *In Proc. American Controls Conference*, pages 1254–1259, 2002.