Task Scheduling: A Control-Theoretical Viewpoint for a General and Flexible Solution

MARTINA MAGGIO, Lund University
FEDERICO TERRANEO and ALBERTO LEVA, Politecnico di Milano

This article presents a new approach to the design of task scheduling algorithms, where system-theoretical methodologies are used throughout. The proposal implies a significant perspective shift with respect to mainstream design practices, but yields large payoffs in terms of simplicity, flexibility, solution uniformity for different problems, and possibility to formally assess the results also in the presence of unpredictable run-time situations. A complete implementation example is illustrated, together with various comparative tests, and a methodological treatise of the matter.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Process control systems; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms: Design, Performance

Additional Key Words and Phrases: Task scheduling, feedback control, control-based system design, formal assessment, discrete-time dynamic systems

1. INTRODUCTION

Control-theoretical methods are nowadays applied to a variety of computing system problems, in a view to provide adaptation capabilities to withstand varying and unpredicted environmental conditions. However, in some sense, controllers are "sitting at the door" of the computing domain, as they are almost invariantly applied to systems that are fully functional also in their absence. For example, coming to the scope of this work, many controllers have been proposed to adapt the tunable parameters of scheduling algorithms, but those algorithms would work—albeit without adaptation—also if the controller were not present.

Recent advances have shown that this is a partial view on the matter, since in many cases controllers can in fact entirely replace computing system components—or, better, said components can be totally designed as controllers. For example, sticking again to the subject of this work, a feedback control structure can *be* a task scheduler.

Introducing control in computing systems by "opening the door" and accepting to use it right from the design phase, has two main advantages. First, we argue that the resulting adaptive component is normally simpler than if a "non-control-theoretic" one

Received October 2011; revised February 2012, April 2012; accepted May 2012

Authors' addresses: M. Maggio, Department of Automatic Control, Lund University; email: maggio.martina@gmail.com; F. Terraneo and A. Leva, Department of Electronics and Information, Politecnico di Milano.

was initially created, and then a control layer was added on top of it to make it self-adaptive or self-aware. The component designed following control-theoretical principles could be more complex than the original nonadaptive one that one may start with, but in general, as a second and most important point, an entirely control-based design can be analysed and assessed in the system-theoretic sense, yielding methodologically grounded performance and robustness guarantees, also in the presence of discrepancies between the design and the actual operating conditions. As a consequence, in general "fully control-theoretical" components are not only simpler, but also more effective, safe, and flexible.

This work applies and demonstrates the above envisaged ideas in the context of task scheduling. Its contributions are summarised below.

—A unifying paradigm is introduced that allows to both represent existing scheduling policies, and design new ones. Said paradigm is fully grounded on system-theoretical concepts. As a consequence, it allows to study and assess a policy's behaviour *analytically*.
—Along the proposed paradigm, a novel general-purpose scheduling algorithm, named I+PI for reasons explained later on, is designed and evaluated. Thanks essentially to the underlying paradigm, I+PI allows to seamlessly handle tasks with and without deadlines, periodic and nonperiodic, avoiding the necessity of coordinating multiple scheduling policies. In addition, the I+PI approach is naturally keen to implementations that keep the task activation part confined and isolated, therefore working both with processes and threads. Finally, I+PI makes the system natively adaptive to discrepancies between the assumed and the actual CPU use on the part of tasks, thus inherently robust in the presence of unpredicted runtime phenomena. The scheduling algorithm is a natural fit for soft-real time systems.
—A microcontroller kernel named Miosix with compile-time pluggable scheduling was developed, and its correctness was checked with the MiBench benchmark [Guthaus et al. 2001]. Since Miosix is targeted to microcontrollers, it only implement threads (specifically pthreads), therefore in this work, the word task is used referring to the theoretical approach and the word thread is used when referring to the implementation.
—The I+PI algorithm was realised in Miosix and compared to Round Robin (RR) and Earliest Deadline First (EDF), implemented for the same kernel, with the standard Hartstone benchmark [Weiderman and Kamenoff 1992].
—Additional comparisons are presented with an extended version of Hartstone, to address issues that are not considered in typical (and somehow problem-specific) benchmarks, but are relevant for real-world schedulers' operation.

The article is organised as follows. Related work is briefly reviewed in Section 2. Section 3 then goes through the underlying theory, to illustrate how those achievements are grounded and guaranteed. Section 4 provides implementation details. Section 5 shows some test results, to evidence the achieved goals. Section 6 finally draws some conclusions, and sketches out future research.

2. RELATED WORK

Generally speaking, "closing a loop" in the scheduling algorithm is not a new concept. Probably, the first scheduling algorithm based on the concept of "feedback" is the *Multi-level Feedback Queue*, presented in Kleinrock and Muntz [1972] and described, together with others in a comparative way in Brucker [2007]. According to this algorithm there are different task queues in the system and whenever a new task is introduced, it is destinated to a specific queue based on its priority. The running task is picked from

Table I. Summary of Control Functionalities Introduced in Related Work

	Time computation	Task selection	Activation	Task admission
I+PI	×	×	×	
[Abeni et al. 2002]	×			
[Batcher and Walker 2008]		×		
[Brinkschulte and Pacher 2008]	×			×
[Cervin and Alriksson 2006]			×	
[Cucinotta et al. 2010]	×			
[Cucinotta et al. 2010]	×			
[Kihl et al. 2008]				×
[Kjaer et al. 2009]				×
[Kotecha and Shah 2008]	×	×		
[Lawrence et al. 2001]	×			
[Lohn et al. 2011]	×			×
[Lu et al. 1999]				×
[Lu et al. 2002]				×
[Palopoli and Abeni 2009]	×			
[Xia et al. 2007]			×	

the highest priority nonempty queue. Whenever a task has spent an excessive amount of time in the system without being scheduled, its priority could be increased and its destination queue could change. This ancestor of the feedback scheduling concepts introduced some re-design as well as the key idea of using a feedback signal to determine the scheduler behaviour. In fact, the feedback nature of this scheduler in the system-theoretical sense resides precisely in acting (moving a task from a queue to another) based on measurements (starvation), not in the feedback nature of the queues. Any two languages have some false friends and for the computer science and the control engineer ones, "feedback" is a notable example indeed.

Intuitively, introducing a feedback signal (from now on in the system-theoretical sense) is a key point in the adoption of a control-based attitude, but the design of the control system is, at least, as important as the chosen feedback signal. The use of a control methodology to design and implement computing systems leads to flexibility, adaptivity, performance control and robustness to variations. The ARTIST2 project was aimed at defining a roadmap on control of real-time computing systems [Årzén et al. 2006]. In the vast majority of cases, the controlled item is the allocation of computing and communication resources. In feedback scheduling the allocation of CPU resources is based on a comparison of the actual resource consumption with the desired one. One of the main output of this research is recognizing the absence of a macroscopic physical level to be used for modelling the system in a suitable way for control purposes. In this work we addressed this limitation at least for the specific problem of CPU allocation.

As another example of introducing adaptivity at the operating system level, [Xu et al. 2006] controls CPU utilization of a virtual server running a web server with various model predictive control strategies, and compare them. Differently from the quoted work, the purpose of this article is not to complement an existing operating system but to re-design part of it with a control-oriented attitude.

Focusing on the specific topic of scheduling, in the case of single processor (possibly real-time) operating systems addressed herein, EDF has been proved to be theoretically the optimal scheduling algorithm whenever the device is underloaded [Pinedo 2008]. This means that EDF is able to schedule every task pool that is schedulable. Such off-line considerations are not suited to address online behaviour. In fact, the performance of EDF decreases exponentially when the device becomes slightly overloaded. In this work we compared our results to the classical EDF algorithm. In the literature, a different approach to overcome the limitations of the classical EDF was to combine it with a different scheduling algorithm, based on an Ant Colony Optimization design [Kotecha and Shah 2008].

Apparently, Lu et al. [2002] is very close to our work; however, some differences can be underlined. Lu et al. in fact propose a conceptual framework for introducing Feedback Control in real-time operating system scheduling. A remarkable contribution of their work is the introduction of the distinction between open- and closed-loop policies, the latter category corresponding in system-theoretical terms to feedback. An open loop policy is a scheduling algorithm that does not receive any measurement from the running tasks and the operating system, while a closed-loop one does take advantage of such measurements. In their work, the authors proposed to use the estimated future utilization as a control signal and to derive from the desired utilization and miss ratio an admission controller that allows tasks to enter the system. The amount of control and the place where this control is introduced is apparently different from our proposal. Notice that admission control in general is very popular in the context of web servers, where the tasks could be rejected to preserve utilization [Kihl et al. 2008; Kjaer et al. 2009]. Some of the proposed ideas could be useful also in a different context, like the one presented here, but the main difference between the referenced papers and this proposal is the application domain, in one case it being a server and in the other an embedded device.

Elsewhere, Brinkschulte et al. [2002] developed a scheduling algorithm called Guaranteed Percentage Scheduling, based on the idea that each task should execute for a percentage of a round time (although in their approach, this period is a fixed value, hardcoded in the policy itself). The scheduler they developed is not based on any feedback signal in principle, and it maintains hard real-time guarantees with a task admission control policy. However, the percentages of CPU sharing that each task needs were adapted with a feedback loop, based on measurements of the instructions per cycle rate [Brinkschulte and Pacher 2008; Lohn et al. 2011]. Also in this case, a fully functional system that was initially not meant to be adaptive, was closed a feedback loop around, resulting in a “sophisticated, probabilistic processor model.”

Some papers introduce controllers to adjusts the “reservation period”, that is, the times assigned to the tasks, with the purpose of keeping the system utilisation below a specified upper bound [Abeni et al. 2002; Cucinotta et al. 2010a, 2010b; Lawrence et al. 2001; Palopoli and Abeni 2009]. In these works the burst duration is adjusted according to the results of the execution of a controller, built to optimize different cost function. No control-based selection of the next task is envisioned, while this is a native feature of I+PI, emerging when some bursts are set to zero. Moreover, the scheduler of the cited works calculates the next burst every time a task is to be activated, while I+PI views the task pool as a single entity, computing bursts just one per round.

Elsewhere, the Batcher and Walker [2008] reorder the list of tasks to be scheduled with a round robin algorithm in an embedded device, with the aim of reducing cache misses. Also in this case, control is introduced to meet a system requirement by acting on a parameter of a fully functional scheduler, not necessarily designed with that requirement in mind.

3. THEORY

Control-theoretical contributions preliminary to this work were presented in Leva and Maggio [2010], where however the scheduler behaviour was only simulated, and the focus was set on the controller synthesis. A brief overview on that matter is given here, adapting the viewpoint to the scope of this work. The implementation and benchmarking of the proposed scheduler on real hardware raised some issues that are here extensively discussed. Of course, providing a full treatise of the used theory is impossible here; the interested reader can, for instance, refer to Franklin et al. [2010] for background material and deeper theoretical discussions.

As anticipated, however, control-based concepts are used here in a way that is definitely novel in the computing system domain. To understand such a new approach, it is then necessary to abandon for a moment any traditional view on scheduling, review the principles of a control problem and its treatment, discuss their (full) application to computing systems so that the required perspective shift be naturally induced, and finally go back to scheduling. The following principles' introduction is carried out by presenting general ideas while at the same time having in mind an example, that *deliberately* refrains from referring to scheduling. From one point of view, this should help the reader concentrate on principles without being distracted by envisaging their use in the addressed context. From another standpoint, it should help understand the generality of the theory, since as different problems as temperature control and scheduling can be treated more or less the same, thus foreseeing the advantages of using said theory—when possible—in the way proposed here.

3.1. Control Principles (The Required Theory in a Nutshell)

In virtually any control system—think for example to temperature control in a room—some *physical object* (the room) is the site where some *phenomena* (heat generation, storage, and exchange) occur. *Objectives* are to be attained, and quite frequently these take the form of a desired (or *reference*) evolution of those phenomena in time, as appreciated by conveniently chosen measurements (for example, the room temperature has to follow a reference daily profile). A certain *control precision* is required in the presence of acceptable *system variability* (humidity can alter the air heat capacity) and unpredicted exogenous actions, collectively called *disturbances* (a door may be opened, or the external temperature may vary within a range and with a rate for which upper and lower bounds are normally known). Precision is expressed by means of

—*static requirements*: when the system is at rest the room temperature must be within $1^\circ C$ from the desired value;
—*dynamic requirements*: if the reference is subject to a step variation, the room temperature must reach the new value within $1^\circ C$ in ten minutes or less, and without oscillating;
—*disturbance rejection* requirements: if the door is opened for at most 30 seconds, or the external temperature varies by at most $10^\circ C$ with a maximum absolute rate of $4^\circ C/min$, the room temperature must never drift from the reference by more than $2^\circ C$, and recover the reference within $1^\circ C$ in five minutes or less;
—*robustness requirements*: all the above must hold true also if any or even all of the system parameters, such as the air or walls heat capacity or some thermal exchange coefficient, differ from the nominal value used to solve the problem by $\mp 10\%$.

There are in general also *asymptotic stability requirements*, meaning that if perturbed the system must tend to recover its previous state, but for the purpose of this treatise their quantification may be taken as implicitly given by precision and robustness ones.

To be controlled, the physical object needs *instrumenting*, that is, endowing with *sensors* to appreciate the behaviour of the relevant phenomena over time (a room temperature sensor), and with *actuators* to influence the system (a heater, supposing for simplicity that cooling is not required). At this point an *oriented system* is defined, having as inputs the actuator's action or *control variables* (the heater command) and the disturbances (the openings' areas and the external temperature), and as outputs the controlled variables (the room temperature). This system is called the *plant*.

If up to now everything is well done and consistent, it is possible to mathematically represent the plant with a *model*. In any case of interest such a model is *dynamic*, which in the control jargon means that knowing the evolution of the inputs in time is not enough to know the evolution of the outputs: for that purpose, it is also necessary to know the initial values of other quantities, called the *state variables*. In the room case, for example, knowing the behaviour of heater command, openings' area and external temperature from a certain (initial) instant does not provide the behaviour of the room temperature, since it is also required to know what the temperature was at the initial instant.

Dynamic models can be of many types. Limiting the scope to what is relevant here, they can take the form of (time) difference equations: for example, neglecting for simplicity any heat storage except that of air, the room temperature at time t_k equals that at time t_{k-1}, k being an integer index, plus the net energy (positive or negative) entering the room from t_{k-1} to t_k (owing to the control action of the heater and/or to the disturbances introduced by openings and eternal temperature) divided by the air thermal capacity. Such models are called *discrete-time dynamic systems*, and take the general form

$$\mathcal{P}: \begin{cases} x(k) &= f(x(k-1), u(k-1), d(k-1), \theta) \\ y(k) &= g(x(k), \theta), \end{cases} \tag{1}$$

where vectors u, d, y, and x are respectively the control inputs, the disturbances, the outputs (to be controlled) and the states, while θ contains the model physical parameters. In (1), function f provides the "next" state given the "previous" one and the inputs, while g gives the outputs. The dynamic character of f, evidenced by the presence of two time index values, provides (1) with memory of the past, and gives a quantitative meaning—via the idea of *state* to the intuitive fact that the same inputs can result in different outputs depending on the system's condition. Knowing θ, x at a given time—say zero for simplicity—and $u(k)$, $d(k)$ for $k \geq 0$, Equation (1) permits to compute $x(k)$, $y(k)$ for $k \geq 0$, that is, to *simulate* the model of the plant, denoted in the following by $\mathcal{P}$. The absence of the inputs u and d in the second equation of (1) is a technical hypothesis inessential to discuss here, suffice to justify it in this context by saying that no input-output action can be instantaneous–a simple matter of causality.

To attain the objectives, a *(feedback) controller* is typically used. Such a controller measures y, knows the reference r for it, and computes u so that y follows r within the specifications despite any expectable d. Mathematically this means creating a new dynamic model $\mathcal{C}$ representing the controller, in the form

$$\mathcal{C}: \begin{cases} \xi(k) &= \varphi(\xi(k-1), r(k-1), y(k-1), \psi) \\ u(k) &= \gamma(\xi(k), r(k), y(k), \psi), \end{cases} \tag{2}$$

with state ξ and parameter vector ψ, φ and γ playing the same role as f and g in (1). Note that in the second equation of (2) the inputs are present, as it is common in control theory to assume that a controller *is* able to react "instantaneously" to errors and/or disturbances, since this just amounts to neglect (hopefully very small) computation delays. Combining (1) and (2), the overall *closed-loop* system (termed also the *control system*) turns out to be ruled by

$$\begin{cases} x(k) &= f(x(k-1), \gamma(\xi(k-1), r(k-1), g(x(k-1), \theta), \psi), d(k-1), \theta) \\ \xi(k) &= \varphi(\xi(k-1), r(k-1), g(x(k-1), \theta), \psi) \\ y(k) &= g(x(k), \psi), \end{cases} \tag{3}$$

which has $X := [x\,\xi]'$ as the overall state vector, r, d as inputs, and y as output, were the prime sign stands for the transpose operator. Denoting by $\Theta := [\theta\,\psi]'$ the overall parameter vector, (3) can thus be written as

$$\begin{cases} X(k) &= \Phi(X(k-1), r(k-1), \Theta) \\ y(k) &= \Gamma(X(k), \Theta). \end{cases} \tag{4}$$

The control theory provides a wealth of methods to translate the control specifications as introduced above into a desired closed-loop system, that is, desired characteristics for functions Φ and Γ (that, again, play respectively the role of f, ψ and g, γ). Once said translation is done, the control problem is therefore turned into that of finding two functions φ and γ for the controller, that combined with functions f and g from the plant, reproduce the desired Φ and Γ with sufficient precision; methods are also available to quantify that precision, and the consequent discrepancies of the desired behaviour from the reference one. Finally, when a suitable (φ, γ) couple is found, (2) produces the control algorithm in a straightforward manner.

A key feature of the feedback approach is that any unforeseen action on the controlled variables is detected by observing measurements of those variables over time. Feedback control requires neither very detailed models of the controlled objects, nor complex prediction procedures for possible unexpected exogenous actions: it only needs reliable measurements.

Coming back to models, and considering (4), some of its equations (denote them as set $\mathcal{E}_s$) come from the physics of the controlled object and cannot be altered, while the others (set $\mathcal{E}_c$) come from the controller, and the connection between the latter and the object where the controlled phenomenon occurs. A key rule of the game is to select $\mathcal{E}_s$ properly, that is, to include in it all that is necessary to appreciate the object's behaviour in a view to attain the control desires, but nothing else. Another rule is to identify all external actions that influence the phenomenon, and classify them into (possible) control signals and disturbances. The former can be acted upon, the latter cannot, and the role of the controller is to reduce their effect on the controlled variables as much as possible.

3.2. Back to Scheduling (The New Control-Based Design Perspective)

The triggering remark of the presented research is that, in the computing system domain and particularly in scheduling, the two rules just mentioned are hardly ever considered as a design paradigm. Specialising that remark to scheduling, in the light of the general ides above, provides the material to be discussed here.

Consider a pool of tasks that need to share a CPU, and attempt to isolate the sole phenomenon that cannot be omitted in any description of it, whatever the scheduling policy is. Not surprisingly, this phenomenon is simple: at the beginning of the time span between a scheduler intervention and the subsequent one, some tasks are allotted a CPU burst (sometimes also called "reservation period" or "scheduling bandwidth"); at the end of the same time span, those processes have used a certain amount of CPU time, not necessarily equal to their bursts. For each task this is simply translated in the difference equation

$$\tau_t(k) = b(k-1) + \delta b(k-1) \tag{5}$$

where k counts the scheduler interventions, τ_t is the actually used CPU time, and b the burst. The disturbance δb accounts for any action on the phenomenon other than that of allotting b, such as for example anticipated CPU yields, delays in returning the CPU whatever the cause is, and so forth.

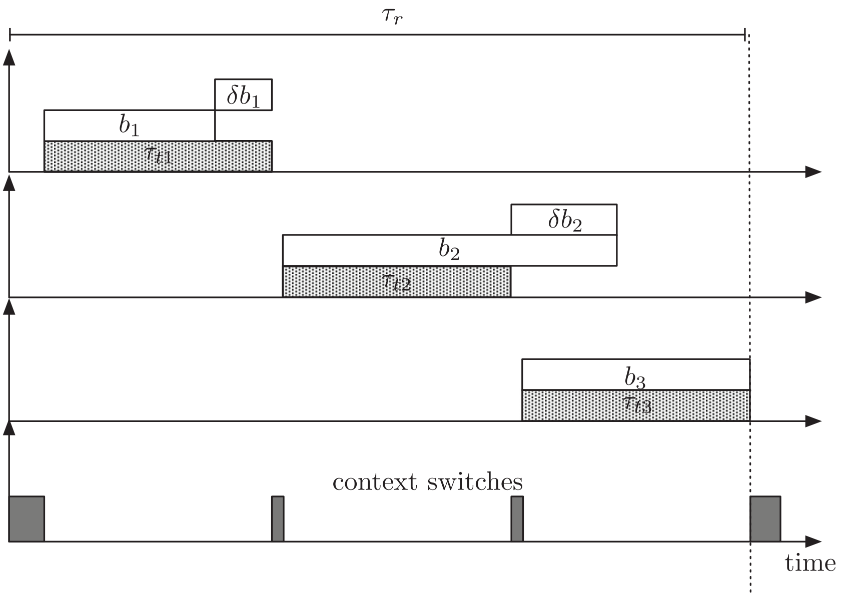


Fig. 1. Meaning and behaviour over time of the involved quantities.

Extending (5) to the entire pool, one obtains the model

$$\begin{cases} \tau_t(k) &= b(k-1) + \delta b(k-1) \\ t(k) &= t(k-1) + \sum b(k-1) + \sum \delta b(k-1) \\ \tau_r(k) &= \sum \tau_t(k), \end{cases} \tag{6}$$

where summations are over the pool, τ_r is the time between two subsequent scheduler interventions (no matter how many tasks were allotted a nonzero burst and in which order), and t is the system time. Model (6) respects the rules, in that it is entirely physical, accounts for all the entities acting on the phenomenon, and exposes both the actually used CPU times and the time between two subsequent instants when the scheduler regains control. Based on those quantities, desires on fairness, responsiveness and so on (i.e., typical metrics used to evaluate schedulers behaviour) can be reformulated in control theoretical terms. Figure 1 gives a visual representation of the meaning of the involved quantities and their behaviour in time, introducing also the presence of the scheduler operations and context switches, τ_t values represents the effectively elapsed times, in the figure, the time spent executing the first task, τ_{t1} is greater than the one assigned by the scheduler (corresponding to b_1), for example because the task was not correctly preempted when its quantum expired. In this the value of the corresponding disturbance δb_1 is greater than zero. On the contrary, the second task executed for less time than planned, for example because it executed a yield or stops waiting for interrupts. In this case δb_2 is a negative value that accounts for the difference. The third tasks just used its burst. The fourth line represents scheduling time and context switches duration. Also, the round time τ_r is depicted.

3.3. Control Synthesis

The I+PI scheduler and the controlled task pool are completely represented by the block diagram of Figure 2, which was used to synthesise and assess the policy, and also for verification-oriented simulations prior to the actual implementation. In that

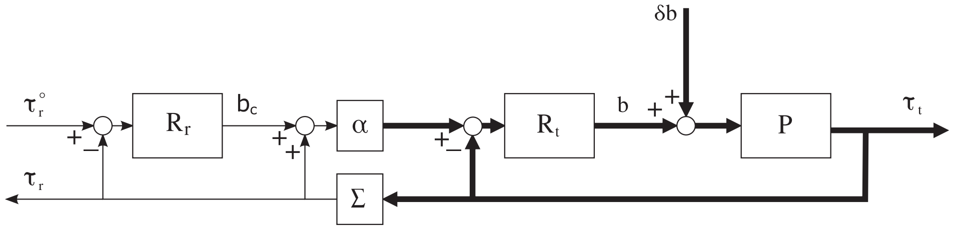


Fig. 2. The proposed scheduler as a feedback control block diagram.

scheme P—plus the input summation node—is the "controlled plant," that is, the first equation in (6) with control input b, disturbance input δb, and output τ_t; the block denoted by Σ realises the third equation in (6), producing the round duration τ_r; the other blocks compose the controller, and will be dealt with in the next sections.

The scheme is of the so called *cascade* type, as two nested loops can be recognised. The inner loop is devoted to ensuring the prescribed distribution of the CPU time, acting on the bursts. The outer loop introduces an additive burst correction so as to keep the round time to the desired value.

3.3.1. Inner Loop. The inner loop is composed of the task pool and a diagonal integral (I) regulator, realised by block R_t, whence the first part of "I+PI". Since also model (6) is diagonal as for the $b \mapsto \tau_t$ relationship, the result is a diagonal (or "decoupled") closed-loop system that can be studied by simply considering one of its scalar elements. Also, the choice of the (diagonal) I structure stems from the pure delay nature of said elements—evidenced by the first equation in (6)—as a typical control design procedure; see Franklin et al. [2010]. In view of this, if for each burst b_i an integral discrete-time controller with gain k_I is adopted, that is,

$$b_i(k) = b_i(k-1) + k_I \left(\tau_{t,i}^\circ(k-1) - \tau_{t,i}(k)\right), \tag{7}$$

where $\tau_{t,i}^\circ$ is the *set point* (the control-theoretical term for "desired value") for the i-th component $\tau_{t,i}$ of τ_t. The inner closed loop is thus represented in state space form by

$$\begin{bmatrix} \tau_{t,i}(k) \\ b_i(k) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -k_I & 1 \end{bmatrix} \begin{bmatrix} \tau_{t,i}(k-1) \\ b_i(k-1) \end{bmatrix} + \begin{bmatrix} 0 \\ k_I \end{bmatrix} \tau_{t,i}^\circ(k-1) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \delta b_i(k-1). \tag{8}$$

Observing system (8) with inputs $\tau_{t,i}^\circ$, δb_i and output $\tau_{t,i}$, it can be concluded that the disturbance is asymptotically rejected and the set point followed with a response time (in rounds) dictated by k_I provided that the eigenvalue magnitude is less than the unity, that is, $|1 \pm \sqrt{1-4k_I}| < 2$. A good default choice is to have two coincident eigenvalues in 0.5, hence $k_I = 0.25$. Higher values of k_I make the controller respond "more strongly" to the difference between the desired and achieved $\tau_{t,i}$, thus making the system faster at rejecting disturbances (owing to a prompt action) but easily producing oscillatory responses to set point variations (owing to a possibly transiently excessive action); lower values of k_I, intuitively, cause the reverse to happen. Figure 3 illustrates the matter, and shows why 0.25 could be used as default for a scheduler with no real-time requirements, and 0.5 could be used for a soft real-time one. More detailed computations on the relationship between k_I and the obtained responses are omitted for brevity.

Trivial computations lead to write from (8) the iterative law describing the closed inner loop's behaviour in time, which takes the form

$$\begin{cases} \tau_t(k) &= \tau_t(k-1) - k_I I \tau_t(k-2) + k_I I \tau_t^\circ(k-2) + \delta b(k-1) - \delta b(k-2) \\ b(k) &= b(k-1) + k_I I \tau_t^\circ(k-1) - k_I I \tau_t(k-1) \end{cases} \tag{9}$$

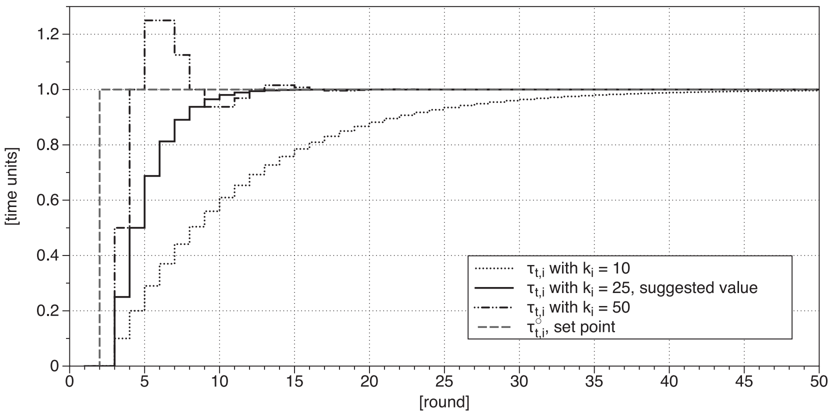


Fig. 3. Inner loop set point responses for different values of k_i.

where I is the identity matrix of dimension equal to the number of tasks. This realises R_t, as the second equation of (9) is the required control algorithm for the inner loop.

3.3.2. Outer Loop. Once the inner loop is closed, the convergence of the actual CPU times to the required ones is ensured since choosing eigenvalues with magnitude lower than the unity ensures asymptotic stability, and the regulator contains an integral action [Franklin et al. 2010]. To determine the set point τ°_t, an outer loop is used that provides an additive correction (b_c in Figure 2) so as to maintain the round duration τ_r to a prescribed value τ°_r; the computation of b_c is accomplished by block R_r. It can be verified that choosing a single k_I for the inner loop results in a $b_c \mapsto \tau_r$ relationship independent of α. A suitable controller structure for the outer loop, along considerations analogous to those that led to the I one for the inner loop, is then the Proportional plus Integral one (PI), whence the rest of "I+PI." Reasoning in the same way as for (8) this leads to determine the closed outer loop's behaviour in time as ruled by

$$\begin{cases} \tau_r(k) = 2\tau_t(k-1) - (1+k_Ik_R)\tau_r(k-2) + k_Ik_Rz_R\tau_r(k-3) \\ \qquad\quad + k_Ik_R\tau^\circ_r(k-2) - k_Ik_Rz_R\tau^\circ_r(k-3) \\ x_R(k) = x_R(k-1) + k_R(1-z_R)\left(\tau^\circ_r(k-1) - \tau_r(k-1)\right) \\ b_c(k) = x_R(k) + k_R\left(\tau^\circ_r(k) - \tau_r(k)\right), \end{cases} \tag{10}$$

where again, the second and third equations provide the control algorithm for R_r (x_R is the PI state variable), while the role of block α should now be self-evident. The PI parameters k_R and z_R can be set in various ways and are both connected to the response speed. Stability is ensured if the roots of the characteristic equation

$$z^3 - 2z^2 + (1+k_Ik_R)z - k_Ik_Rz_R = 0 \tag{11}$$

in the unknown z, have magnitude less than the unity, which provides easy parameter bounds, while disturbance rejection is still guaranteed by the contained closed inner loop.

As a result of the synthesis process just sketched, the I+PI algorithm is unambiguously defined as follows.

3.3.3. Simulation Example. Equations (9) and (10) also allow to simulate the control system. An example is shown in Figure 4 to illustrate the set point following and disturbance rejection characteristics. A pool of three tasks is considered, and both the required CPU distribution (vector α) and the desired round duration (τ°_r) are varied.

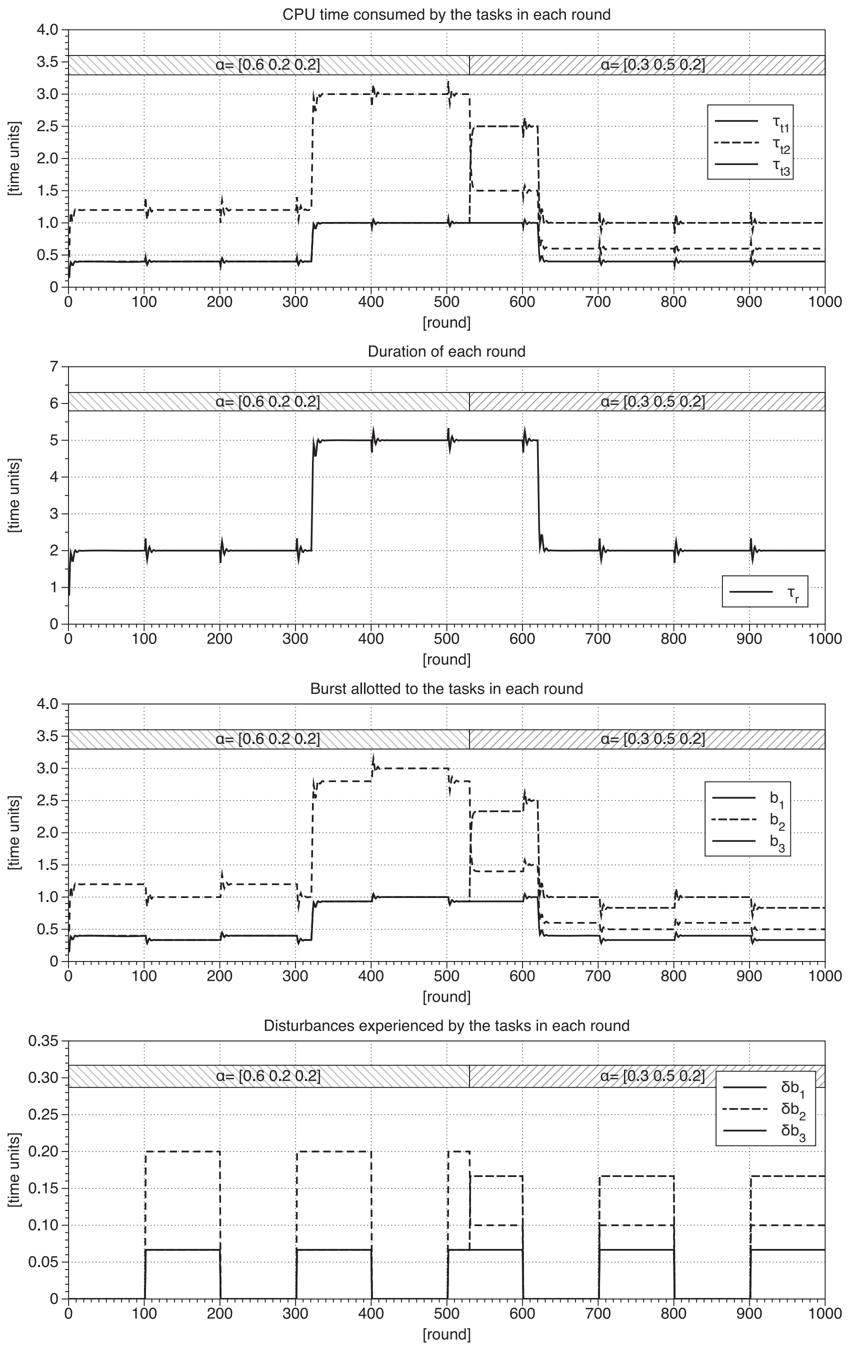


Fig. 4. Simulation results to demonstrate set point following and disturbance rejection.

ALGORITHM 1: I+PI algorithm (the complete C implementation is about 40 lines long)

```
Initialize the I and the PI state variables
for each scheduling round $k$ do
  Read the measured CPU times used by the $N_t$ tasks in the previous round into vector $\tau_t(k-1)$
  Compute the measured duration of the last round as $\tau_r(k-1) = \sum_{i=1}^{N_t} \tau_{t,i}(k-1)$
  Read the required round duration $\tau_r^\circ(k-1)$
  if the task pool cardinality or parameters have changed then
    Reinitialize $b_i(k)$ to the default values
  else
    Compute the burst correction $b_c(k)$ for this round by the PI algorithm:
      $b_c(k) = b_c(k-1) + k_{rr}(\tau_r^\circ(k-1) - \tau_r(k-1)) - k_{rr}z_{rr}(\tau_r^\circ(k-2) - \tau_r(k-2))$
    Apply saturations to $b_c(k)$
    Compute the vector $\alpha(k)$ of required CPU time fractions
    for each task $i$ do
      // Compute the burst vector $b(k)$ for this round the by the I algorithm:
        $\tau_{t,i}^\circ(k) = \alpha_i(k)\tau_r^\circ(k)$
        $b_i(k) = b_i(k-1) + k_{it}(\tau_{t,i}^\circ(k) - \tau_{t,i}(k-1))$
        Apply saturations to $b_i(k)$
    end for
  end if
  Activate the $N_t$ tasks in sequence, preempting each of them when its burst is elapsed
end for
```

When the set point is modified, the behaviour of the system changes accordingly, therefore resulting in different curves. Also, steplike disturbances are periodically introduced in the actual CPU time consumed by the tasks. In the same conditions, the system react to disturbances exactly in the same way. When the set point changes, the reaction is different. However, as can be seen, both characteristics (CPU distribution, that is, weighed fairness, and round time, i.e., responsiveness) can be prescribed as set points, and the system achieves them also in the presence of disturbances. Note, in this respect, how fast the CPU times and the round duration converge to their set points, and how quickly the effect of disturbances vanishes. Also, note the smooth behaviour of the allotted bursts.

Once the control system is assesses as dynamic system and simulated to get further insight on the aspect of the obtained transients, the code-abstracted part of the design is completed. From now on, in other words, it is *proven* that the solution matches the problem, and the only issue is to check that the code, that comes from the scheduler related equations, matches the solution.

4. IMPLEMENTATION

The full scheduler implementation is outlined in Figure 5. The scheduler can be divided into two parts:

—The *I+PI* controller algorithm, described in Section 3.3 and Algorithm 1, which computes the bursts values. It should be stressed that I+PI runs once per round, not once per task. At the beginning of each round I+PI computes the values for all the units present in the task pool. Tasks can be then run one after the other without any further scheduler intervention, except than very simple context switches.

—The *set point generator*, that needs running only when changes occur in the task pool, the required CPU distribution, the required round duration or any combination thereof. Its aim is to compute the reference signals for the I+PI algorithm. Set

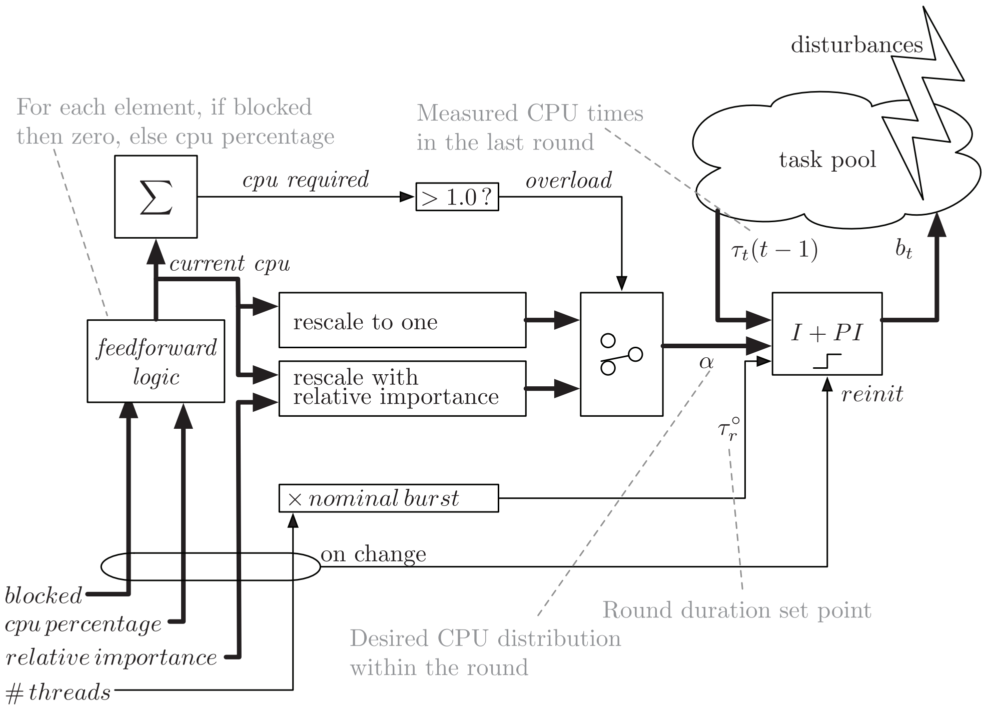


Fig. 5. Implementation scheme, containing the I+PI regulator and the set point generator.

point generation can be further divided into “overload detection and rescaling” and “reinitialisation and feedforward”.

It is worth evidencing that the correct behaviour of the scheduler in terms of stability and performance depends inherently on the correct realisation of the I+PI algorithm only and thus can be checked formally. All the rest (the second item above) is, in control-theoretical terms, “outside the loop,” and cannot alter the system stability as assessed on the model. This is very important to streamline the design process, as once the core algorithm is written, parametrised and checked, all the rest of the code *structurally cannot* have unexpected or disruptive impacts on the system. Needless to say, also the code structuring and modularisation takes profit from the concepts just recalled.

Since I+PI is a closed-loop scheduler, it requires measurements of the actual CPU times consumed by the individual tasks in the previous scheduling period. To achieve that, the Miosix implementation makes use of a hardware timer that can also be configured for scheduling preemption. The timer is started at the end of the context switch code and its value is read at the beginning of the next switch. This allows to measure execution time with a fine-grained resolution.

Set point generation is ruled by two input parameters. One is an estimate of the CPU percentage that each task requires, for example not to miss deadlines. The second is the relative importance of each task, which is used to handle CPU overload situations. These parameters can be set by the task itself through an API provided by the Miosix kernel, and can be dynamically changed during the lifetime of the task to reflect changes in its behaviour. In addition, the scheduler needs to know which tasks are blocked, for example sleeping or waiting for I/O operations.

4.1. Overload Detection and Rescaling

From time to time, especially in soft real-time systems, the CPU utilisation may exceed the unity. This means that the sum of the required CPU percentages (for all nonblocked tasks) exceeds one. This situation is used in the proposed solution to detect a CPU overload situation, signalling that the task pool is not schedulable. This overload indicator is used to select the rescaling policy to be used.

If the task pool is schedulable the "rescale to one" policy is used to produce vector α, by rescaling the required CPU percentage vector so that its sum equal one. For example, consider a task pool with four tasks, of which three require a 20% CPU share, and the fourth one is blocked and therefore requires zero CPU share. The policy will result in an α array of $\{0.33, 0.33, 0.33, 0\}$. This policy will, by design, give a CPU share greater or equal than the one requested by the tasks, ensuring that the tasks have enough CPU to carry out their job successfully. It is particularly significant that this policy ensures good real-time performance—the following benchmarks should evidence it—even without the scheduler having any knowledge of deadlines whatsoever. In other words deadlines are *implicitly* enforced by ensuring that the involved tasks receive enough CPU share on time.

In the presence of CPU overload, conversely, this policy is not adequate. Consider an example with three tasks, all of which require a 50% CPU share. The rescaling would give a CPU share of 33% to all three tasks, so if deadlines are present all of them will eventually start missing. In this case the "rescale with relative importance" policy is thus used. This policy first weighs the CPU share using the relative importance parameter and then rescales the resulting α vector to have unitary sum as before. As a result, two tasks that require the same CPU share will receive a burst proportional to their relative importance parameter. This significantly differs from classical approaches to tackle similar issues, that are typically based on *task priorities*, in that the proposed policy allows to *predict* the CPU share that will be received by all tasks even in the case of overload. Also, and again differing from priority-based techniques, the relative importance parameter is only taken into account when CPU overload occurs, therefore having no influence when the pool is schedulable. Notice that the relative importance parameter can be set based on the task importance but is a parameter of the control system and does not influence its formal assessment.

Once again, recall that both these techniques for rescaling simply produce the set points for the same regulator and control algorithm, that stays untouched.

4.2. Reinitialisation and Feedforward

Regulator reinitialisation and feedforward have been introduced to improve the scheduling dynamic performance in the presence of task blockings. A task is said to block if it stops being able to accept the CPU for a period of time. Blocking causes include voluntarily sleeping, waiting on a locked mutex or other synchronization primitives, or waiting for an I/O operation to complete.

The I+PI algorithm is intrinsically capable of responding to task blockings due to its closed loop nature—in other words, due to past behaviour measurements—without any external intervention. However, reinitialisation and feedforward control were introduced to improve its *dynamic* performance. To show the advantages of using these two features, a simple example of what happens if reinitialisation and feedforward are not present is presented. Consider a case with two tasks, of which one repeatedly blocks. In this case the external PI regulator is able to quickly regain control of the round time duration, but in the meantime the integral regulator of the blocked task is subject to a constant error and, as such, diverges till saturation occurs. When the blocked task becomes ready again, the scheduler assigns to it a very long burst, equal

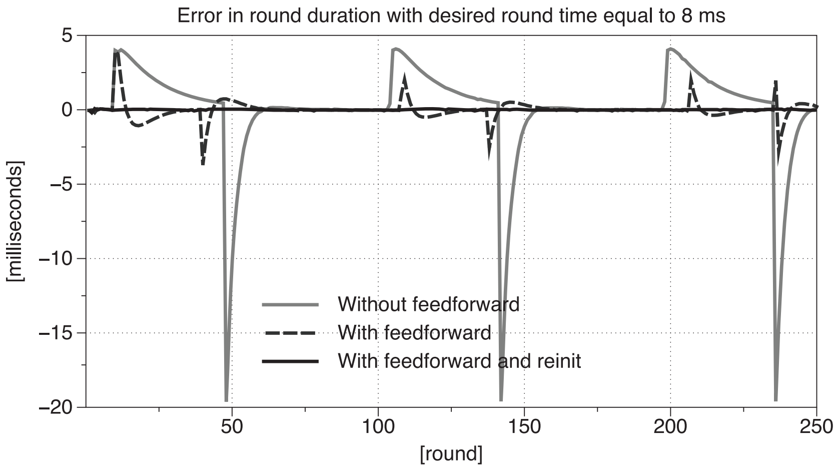


Fig. 6. Effects of task blockings on the round duration control (data from hardware implementation).

to the saturation value—a typical case of integral windup. While this situation is recovered after a short number of rounds, these spikes in the round duration may cause deadline misses. An experiment showing how this can actually happen is depicted in Figure 6.

The I+PI scheduler is a dynamic system, and as such has an internal state. The reinitialisation policy works by resetting this state to its default value whenever the task pool parameters change. This includes resetting the saturated integral regulator of a blocked task, therefore improving the dynamic response to task blockings.

The feedforward policy is instead grounded on the fact that task blocking is a measurable disturbance. A task, to sleep or wait, has to call some kernel API and as such, the scheduler is informed. This allows to further improve the dynamic response by changing the I+PI set points, namely by setting to zero the α elements corresponding to blocked tasks and distributing the round time among nonblocked tasks only.

All the benchmarks results provided in this article have been obtained with both features enabled.

5. RESULTS

The I+PI algorithm, together with RR and EDF, has been implemented in a kernel targeted to microcontrollers, named Miosix, and released as free software.[1] All the tests were made with a `stm3210e-eval` board, equipped with a 72 MHz ARM microcontroller and a 1 MB external RAM, from which the code executes.

Some test sets are presented. The first one demonstrates the ability of the implementation to effectively distribute the CPU to the running tasks and the correctness of the tasks execution, comparing their output with the reference one proposed in the benchmark suite. The second one compares I+PI to RR and EDF with a benchmark conceived for periodic tasks, limiting the scope to feasible CPU utilisations, and shows that I+PI closely approaches the EDF (optimal) results although not being tied to the concept of deadlines. Also, in this case some numbers concerning the scheduling overhead are presented. The third set considers the apparently off-design condition of a

[1]The code for Miosix and all the experiments reported here is available at http://home.dei.polimi.it/leva/Miosix.html. At the same URL the reader can find full kernel licence details, and the entire result datasets of which only a part is here reported.

Table II. Summary of Relevant MiBench [Guthaus et al. 2001] Execution Data

c[1]	benchmark	time [s]	.text [B]	.data [B]	.bss [B]	stack [B]	heap [B]
A	basicmath	61.431	75600	1288	348	1472	4684
A	bitcount	8.531	63848	1288	348	1416	4148
A	qsort	3.545	90420	1328	140604	1336	5212
A	susan[2]	8.711	97008	1328	604	370312	179588
C	jpeg[3]	14.812	231016	1336	660	3440	246228
C	tiff2bw	67.359	259200	1728	780	2408	45492
C	tiffdither	85.801	259040	1624	788	3036	41652
C	tiffmedian	113.334	253136	1600	133428	1344	160700
O	ispell	5.206	134668	1736	31436	4460	732252
O	stringsearch	0.023	64116	1280	1380	1664	6948
N	dijkstra	14.290	90384	1328	41436	1356	14308
N	patricia	10.628	57516	1328	604	1556	14308
N, S	sha	3.354	70596	1280	348	2976	5596
S	pgpsign	6.969	262980	5288	230140	14376	13588
S	pgpverify	1.088	263076	5288	230140	15964	25780
S	rijndael[4]	44.913	104128	1296	356	1416	8908
N, T	crc32	162.554	68708	1280	348	1336	7484
T	fft/ifft	94.993	71184	1288	348	1504	160652
T	adpcm[3]	52.983	69636	1280	2852	1360	8148
T	gsm[3]	12.145	135972	1576	604	1832	9300

[1]Categories are: Automotive/industrial, Consumer, Office, Network, Security, Telecommunications.
[2]Execution time includes all the three phases (edge detection, corner detection, and smoothing).
[3]Execution time includes both encoding and decoding.
[4]Source code was flawed, thus it was fixed and output correctness was compared under Linux with the fixed version instead of the reference.

task pool not schedulable owing to CPU overutilisation, where the approach behind I+PI makes it inherently superior to non-control-theoretic ones.

5.1. Test Set 1 (MiBench Benchmark)

The open source MiBench suite [Guthaus et al. 2001] is used here to assess the correctness of the Miosix pluggable scheduler implementation. The suite contains applications that span from mathematical computation to image encoding, network routing, cryptography and GSM audio encoding/decoding, thereby representing a variety of workloads. There are two tests per benchmark, a "small-" and a "large-input" one. Owing to the available hardware limitations, only the small-input one is here used (apparently, with no generality loss). Table II reports some execution time and memory occupation results. There are no significant differences between the results of Miosix and the reference ones provided with the suite.

In addition to the above tests that consider one application at a time, a further experiment was done by simultaneously running several applications, one per category, each in a thread of its own, with I+PI and RR scheduling (using EDF would imply setting fictitious deadlines, to the detriment of test significance). The used applications are basicmath, jpeg, stringsearch, dijkstra, sha, and gsm. With both schedulers each output correctly matches its reference.

In Table III, column T_{seq} and T_{sim} respectively report the sum of the *individual* applications' execution times, and the duration of their *simultaneous* execution. As can be seen, both schedulers are capable of re-assigning CPU time when one or more thread gets stuck (in this case, owing to I/O operations). However I+PI is more effective, as witnessed by both the pure parallel execution times and the differences between the

Table III. Summary of Simultaneous MiBench [Guthaus et al. 2001] Execution Times [s]

scheduler	basicmath	jpeg	stringsearch	dijkstra	sha	gsm	T_{seq}	T_{sim}
I+PI	61.641	14.812	0.023	14.290	3.354	12.145	**106.055**	**100.535**
RR	60.207	14.883	0.023	14.038	3.555	11.031	**103.737**	**101.504**

Table IV. The Hartstone [Weiderman and Kamenoff 1992] Baseline Task Set

task	frequency	workload	workload rate (workload/period)
1	2 Hertz	32 Kilo-Whets	64 KWIPS
2	4 Hertz	16 Kilo-Whets	64 KWIPS
3	8 Hertz	8 Kilo-Whets	64 KWIPS
4	16 Hertz	4 Kilo-Whets	64 KWIPS
5	32 Hertz	2 Kilo-Whets	64 KWIPS

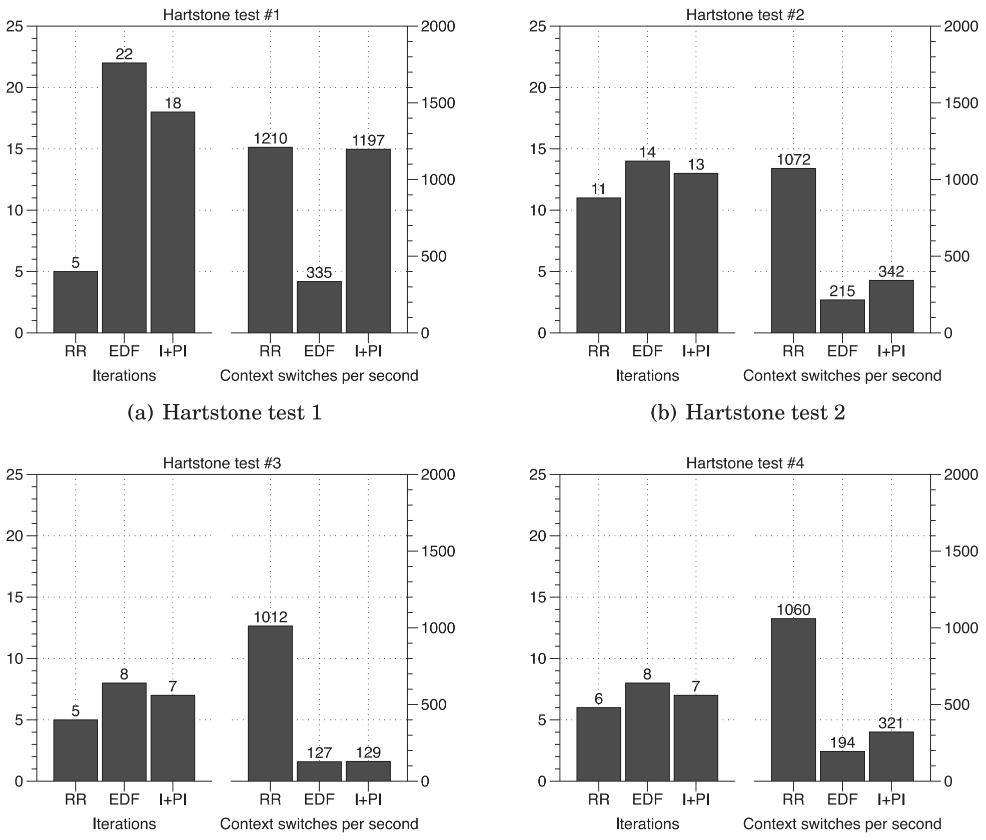
sequential and parallel times. This suggests that the I+PI scheduling overhead tends to be of modest entity with respect to improvements in CPU time management. The mentioned results find their mathematical explanation in Section 3.

5.2. Test Set 2 (Hartstone Benchmark)

The Hartstone benchmark [Weiderman and Kamenoff 1992] is used here to compare I+PI to EDF and RR. Hartstone is composed of several series of tests; each consists of starting from a baseline task system, verifying its correct behaviour, and then iteratively adding stress to that system and reassessing its behaviour until said assessment fails. The amount of added (affordable) stress allows to measure the system capabilities.

This work concentrates on the Hartstone PH (Periodic tasks, Harmonic frequencies) series, that refers to periodic tasks, and stresses the system by adding tasks and/or modifying their period and/or workload. The baseline system is composed of five periodic tasks, that execute a specific number of Wheatstones [Weiderman and Kamenoff 1992] within a period; the workload rate is thus expressed in Kilo-Whets Instruction Per Second [KWIPS]. A Kilo-Wheatstone corresponds in our architecture to a CPU occupation of $1.25ms$, maintained constant through all the tests. As per the benchmark, all the tasks are independent: their execution does not involve synchronisation, they do not communicate with one another, and are all scheduled to start at the same time. The deadline for the workload completion of each task is the beginning of its next period. The used series might thus represent a program that monitors several banks of sensors at different rates, and displays the results with no user interventions or interrupt requirements.

Table IV gives details on the baseline system. In the first test, the highest-frequency task (number 5) has the frequency increased by 8 Hertz at each iteration, until a deadline is missed. This tests the system ability to switch rapidly between tasks. In the second test, all the frequencies are scaled by 1.1, 1.2, ... at each iteration, until a deadline is missed. This means testing the system's ability to handle an increased but still balanced workload. The third test starts from the baseline set and increases the workload of each task by 1, 2, ... Kilo-Whets at each iteration. This increases the system overhead in a nonbalanced way. In the last test, at each iteration a new task is added, with a workload of 8 Kilo-Whets and a frequency of 8 Hertz (as the third task of the baseline set). This test stresses the system's ability to handle a large number of tasks.



(a) Hartstone test 1

(b) Hartstone test 2

(c) Hartstone test 3

(d) Hartstone test 4

Fig. 7. Results for the Hartstone benchmark [Weiderman and Kamenoff 1992].

Figure 7 graphically shows the results for the four tests, presenting both the number of successful iterations (higher is better) and the number of context switches per second in the last successful iteration (lower is better). In most cases the number of successful iterations and context switches per second of I+PI are similar to those of EDF, which is notoriously optimal for a schedulable set of periodic tasks. In fact, EDF significantly outperforms I+PI only in the first test, which is apparently the most extreme as for asymmetry in the task periods. This is not to diminish the relevance of the fact but, for example, if in an embedded device a critical task needs to be executed at so higher a rate than the others, one would probably consider hooking it to a timer interrupt. On the other hand, I+PI is definitely superior to RR in any sense.

An overhead analysis is in order, therefore we recorded the duration of a context switch for the baseline test with each of the implemented scheduling algorithm. The numbers are computed running the baseline task set and using an oscilloscope to read a general purpose input/output signal raised whenever the scheduler starts its execution and cleared as it finishes. For the I+PI the context switch in which the scheduler calculates the bursts is obviously longer than the other ones, as clearly shown in Table V. It is worth recalling that this longer context switch happens just one per round. In fact, whenever I+PI has to apply a previously computed control signal it results faster than RR, but intuitively slower than EDF. However, on average, the

Table V. Context Switch Duration within the Hartstone [Weiderman and Kamenoff 1992] Baseline Task Set

scheduler	average [μs]	long [μs]	short [μs]
I+PI	75.8	205.6	43.4
EDF	30.8		
RR	50.4		

Table VI. Relevant Data on the Schedulers' Execution

scheduler	.text [B]	.data [B]	.bss [B]
I+PI	1464	12	40
EDF	476	0	4
RR	600	0	20

overhead of the technique is in general heavier, one way to fasten the computation is to select an architecture with hardware support for floating point operations or to use dedicated hardware. Also, data on the schedulers execution is here reported. Table VI shows some relevant data about the schedulers' execution, retrieved with the size utility contained in the Miosix kernel.

5.3. Test Set 3 (Extended Hartstone Benchmark)

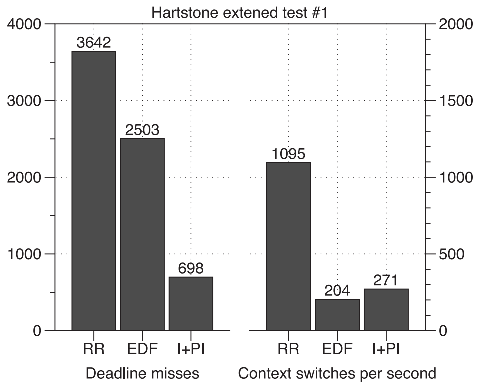
Benchmarks like Hartstone are useful to provide a simple and clear comparison test bed, but do not aim at representing "real life" workloads. For example, any scheduler regularly encounters pools of tasks where each task has its own characteristics. Also, a scheduler may be requested to recover correct operation of (soft) real time tasks after a transient CPU overutilisation, or even to withstand a long-lasting overutilisation by maintaining the timely operation of certain tasks.

As will be explained in Section 3, the general approach behind I+PI is well suited to address such issues. To witness that, I+PI is here compared to EDF and RR in an extension of the Hartstone benchmark. In the reported tests of Figure 8 the way of increasing the system load is the same of the corresponding tests of Figure 7. However, the load is not increased gradually but set so as to result in a 48% CPU utilisation from 0 to 30 seconds, then a 120% utilisation from 30 to 45 seconds and 48% again till the end of the test at 120 seconds. Figures 8(a), 8(b), 8(c) and 8(d) report respectively the total number of misses and of context switches per second in the four tests (lower is better for both). As can be seen, I+PI invariantly achieves the least miss rate, with a moderately higher number of context switches per second with respect to EDF; RR performances are definitely inferior.

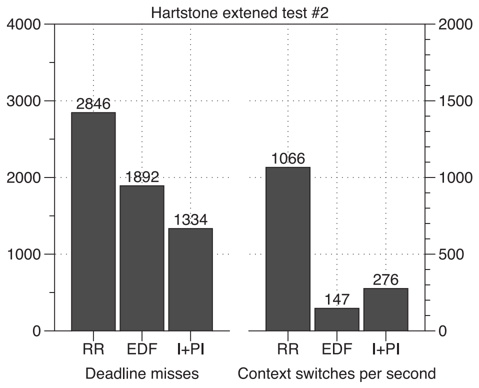
5.4. Summary of Results

From all the reported tests, it can be concluded that I+PI may in some cases be not optimal, but normally approaches optimal performance and above all does not require any assumption on the nature of the tasks (e.g., periodic or not). It is also worth noticing, as a final remark, that the I+PI implementation shown here was realised with floating point computations with an architecture that has no hardware support for them. This was done for convenience reasons inessential to explain here, but could be safely replaced by a sufficiently precise fixed point arithmetic version. Needless to say, this would move the experimental evaluation balance further toward I+PI.

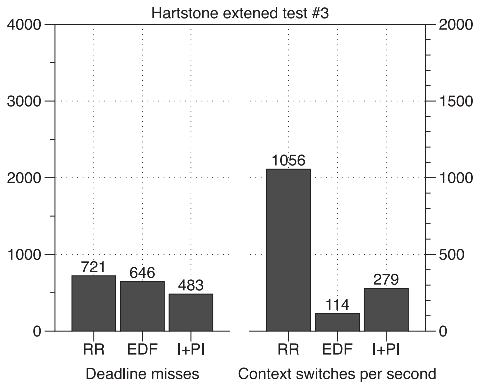
After showing the advantages of the proposed (I+PI) scheduler, it is now the time to go through the underlying theory, and show the potential of the approach that it substantiates.



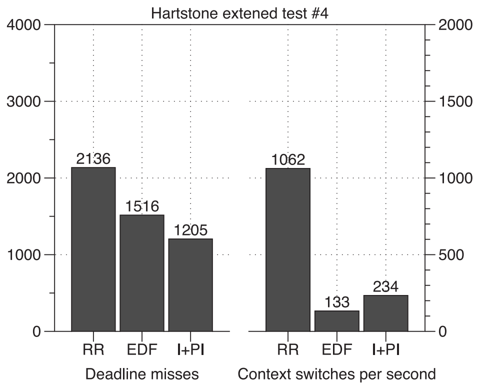
(a) Extended Hartstone test 1



(b) Extended Hartstone test 2



(c) Extended Hartstone test 3



(d) Extended Hartstone test 4

Fig. 8. Results for the extended Hartstone benchmark.

6. CONCLUSIONS AND FUTURE WORK

In this article a control-theoretical approach to task scheduling was presented, leading to a novel design process. Traditionally, in fact, one starts from desires expressed informally, and figures out an algorithm capable of attaining said desires. Modifications and refinements are then cyclically introduced by testing the algorithm in a supposedly wide enough variety of situations, and to decide which modification to introduce, experience and intuition play a crucial role. This frequently results in complex code, where the real effect to a certain modification is hard to predict and can only be appreciated after a long programming work.

In the opinion of the authors, at least in the addressed context, a major reason for that is the absence, in the design cycle, of a system model *in the particular sense the term has in the systems and control theory*. With the proposed design perspective, one first characterises informal requirements in system-theoretical terms, that is—sticking to the subject of this work—as desired behaviours of measured quantities over time. Then, a dynamic model of the controlled object is created, taking care to include in its *equations* only those phenomena that are physically inherent to that object. This allows, by means of rigorous methods vastly assessed in a number of domains, to obtain a dynamic model of the *controller*, from which the control algorithm emerges *unambiguously*. In such a cycle, the correct behaviour of the devised controllers, both in design conditions and in unforeseen ones, can be checked *a priori* and *formally* by only manipulating dynamic models. Once the result is assessed, coding is needed only once, or at most the number of times necessary to check the code against its *dynamic* model. The iterative modification cycle involves only models, thereby resulting faster and generally leading to simpler algorithms.

It was reasonable right from the start to expect such a novel cycle to result in simplicity and performance improvements. In this article a verification of that claim is shown. It was also expectable that casting the variety of scheduling problems in a unified theoretical framework yielded controllers capable of dealing with a wide range of cases with simple algorithms. It was finally conjectured that a model-driven choice of inputs and outputs, combined with a properly designed feedback structure, allowed to treat off-design situations at runtime in an effective and formally verifiable manner. These ideas too were testified, showing that a single (model-originated) feedback algorithm could handle tasks of different types (e.g., periodic and nonperiodic, with and without deadlines) simultaneously and with a code complexity lower or comparable to classical ones, also in the presence of disturbances.

In this work, a scheduler (named I+PI) was devised along the proposed approach, and implemented on real hardware within the Miosix kernel. Its behaviour was tested through the MiBench and Hartstone benchmark and compared with other commonly used scheduling algorithms. An extension of the Hartstone benchmark was also proposed, to evaluate some cases that are not considered by the standard set of tests. The I+PI implementation is outperformed by some other algorithm only in cases for which that algorithm is specifically tailored, and that strictly do not violate the nominal hypotheses under which that algorithm is designed. On the other hand, I+PI is more complex than the sole pure round robin, and treats *all* cases and their mixtures.

The authors hope that this work may foster a deeper use of dynamic models (and the systems theory at large) in the design of computing system components—a domain where said theory and its methods appear particularly promising and not yet fully exploited. Future research will thus be directed to treat other problems, such as resource allocation and QoS, with the same or a similar approach.

ACKNOWLEDGMENT

The authors want to thank Karl-Erik Årzén and the annonymous reviewers for the valuable feedback provided on earlier versions of this manuscript.

REFERENCES

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

K.-E. Årzén, A. Robertsson, D. Henriksson, M. Johansson, H. Hjalmarsson, and K. H. Johansson. 2006. Conclusions of the artist2 roadmap on control of computing systems. *SIGBED Rev.* 3, 11–20.

K. W. Batcher and R. A. Walker. 2008. Dynamic round-robin task scheduling to reduce cache misses for embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*. ACM, New York, 260–263.

U. Brinkschulte and M. Pacher. 2008. A control theory approach to improve the real-time capability of multithreaded microprocessors. In *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing*. 399–404.

U. Brinkschulte, J. Kreuzinger, M. Pfeffer, and T. Ungerer. 2002. A scheduling technique providing a strict isolation of real-time threads. In *Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02)*. 334–340.

P. Brucker. 2007. *Scheduling Algorithms*. Springer.

A. Cervin and P. Alriksson. 2006. Optimal on-line scheduling of multiple control tasks: A case study. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*. IEEE, 141–150.

T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli. 2010a. Self-tuning schedulers for legacy real-time applications. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, 55–68.

T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari. 2010b. On the integration of application level and resource level qos control for real-time applications. *IEEE Trans. Ind. Inf.* 6, 4, 479–491.

G. F. Franklin, J. D. Powell, and A. Emami-Naeini. 2010. *Feedback Control of Dynamic Systems.* 6th Ed. Pearson.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization*. IEEE, 3–14.

M. Kihl, A. Robertsson, A. Andersson, and B. Wittenmark. 2008. Control-theoretic analysis of admission control mechanisms for web server systems. *World Wide Web* 11, 1, 93–116.

M. A. Kjaer, M. Kihl, and A. Robertsson. 2009. Resource allocation and disturbance rejection in web servers using slas and virtualized servers. *IEEE Trans. Netw. Serv. Manage.* 6, 4, 226–239.

L. Kleinrock and R. R. Muntz. 1972. Processor sharing queueing models of mixed scheduling disciplines for time shared system. *J. ACM* 19, 3, 464–482.

K. Kotecha and A. Shah. 2008. Adaptive scheduling algorithm for real-time operating system. In *Proceedings of the IEEE World Congress on Computational Intelligence*. 2109–2112.

D. A. Lawrence, J. Guan, S. Mehta, and L. R. Welch. 2001. Adaptive scheduling via feedback control for dynamic real-time systems. In *Proceedings of the IEEE International Conference on Performance, Computing, and Communications*. 373–378.

A. Leva and M. Maggio. 2010. Feedback process scheduling with simple discrete-time control structures. *IET Control Theory Appl* 4, 11, 2331–2342.

D. Lohn, M. Pacher, and U. Brinkschulte. 2011. A generalized model to control the throughput in a processor for real-time applications. In *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing.* 83–88.

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

C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. 1999. Design and evaluation of a feedback control edf scheduling algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*. IEEE, 56.

L. Palopoli and L. Abeni. 2009. Legacy real-time applications in a reservation-based system. *IEEE Trans. Ind. Inf.* 5, 3, 220–228.

M. Pinedo. 2008. *Scheduling Theory, Algorithms, and Systems.* 3rd Ed. Springer.

N. Weiderman and N. Kamenoff. 1992. Hartstone uniprocessor benchmark: Definitions and experiments for real-time systems. *Real-Time Syst.* 4, 4, 353–382.

F. Xia, G. Tian, and Y. Sun. 2007. Feedback scheduling: an event-driven paradigm. *SIGPLAN Not.* 42, 12, 7–14.

W. Xu, X. Zhu, S. Singhal, and Z. Wang. 2006. Predictive control for dynamic resource allocation in enterprise data centers. In *Proceedings of the 10th IEEE Network Operations and Management Symposium*. 115–126.