A PI-based control structure as an operating system scheduler

Martina Maggio, Federico Terraneo, Alessandro V. Papadopoulos, Alberto Leva

Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza Leonardo Da Vinci, 32 - 20133 Milano, Italy
{*maggio,terraneo,papadopoulos,leva*}*@elet.polimi.it*

Abstract: Many functions of operating systems are keen to be realised as feedback controllers. Doing so has a non negligible design impact, but also a significant payoff in terms of simplicity and generality. This paper presents a complete operating system scheduler, at present implemented in a microcontroller kernel, entirely composed of a PI-based control structure. The proposed scheduler is experimented with in several load conditions. In all of them, it performs in a comparable manner with respect to the classical (i.e., not control-based) policy optimised for that condition, as long as design assumptions such as schedulability are met. In addition, if some off-design situation is encountered, the proposed control-based scheduler definitely outperforms those not conceived as controllers.

Keywords: PI-based control structures; scheduling; operating systems.

1. INTRODUCTION

The complexity of operating systems is nowadays abruptly increasing, and even the architecture of some core functionalities are being affected. To give just one example, consider the Linux scheduler. In the Kernel version 2.4.37.10 (September 2010) all of its code was contained in a single file of 1397 lines. In version 2.6.39.4 (August 2011) the scheduler code is spread among 13 files for a total of 17598 lines. Other examples could be given, but are omitted for space limitations. Indeed, when such "explosions" are experienced, it is at least advisable to somehow reconsider the overall design approach.

Observing the matter from a system- and control-theoretical standpoint, and by the way not necessarily limiting the scope to the scheduler case treated herein, it can be noticed that hardly any operating system functionality has been conceived and developed based on a dynamic model of some physical phenomenon to be controlled. In the scheduler case, the phenomenon is how the CPU is distributed among the running tasks, depending on control actions (the allotted timeslices) and exogenous disturbances (task blockings, resource contentions, and so on); see Pinedo (2008); Brucker (2007) for details. The situation just sketched has quite clear historical reasons. Suffice to say that, while in any other context controlled objects can be modelled based on physical (first) principles, this is not the case for computing systems, because there the "physics" is created by the system designer him/herself. In the absence of a modelling framework, system design is carried out directly in an algorithmic setting, leaving the engineer without any means to assess its behaviour in the sense that term is given in the system and control theory domain.

While such a *scenario* could to date be tolerated, given the mentioned complexity rate increase, it cannot be assured that said tolerability will carry over to the future. In fact, as "more physics" is created, the absence of a rigorous dynamic description of it may sooner or later pose intractable problems as for its governance. As a consequence, rigorous – and possibly simple – modelling frameworks to ground system design upon are needed. The main message this paper wants to convey, is that if one accepts to re-design part of said system, such a framework can be found by (usefully) limiting the model scope to describing the real physical phenomenon on which the addressed aspects of the system behaviour depend. If this is done, surprisingly simple formalisms can be used—a noticeable example indeed of process/control co-design.

This paper shows that, by adopting the attitude just sketched, the scheduling problem can be handled with a single, PI-based control structure – named I+PI –, unifying in a single framework the treatment of cases that are generally addressed by devising *ad hoc* individual solutions. In this paper the focus is set on a single problem, albeit very relevant and emerging with various flavours, as such a specialisation allows to describe the entire design process, from control synthesis through formal assessment and simulation, up to real code for a kernel targeted to microcontrollers, named Miosix, and released as free software [1] . However, in the authors' opinion, many other computing system problems can be addressed in a way similar to this, revealing further effective applications for simple controllers such as PIDs.

2. THE CONTROL SCHEME

In this work a "scheduler" is realised as a feedback controller. This implies a perspective shift from the idea of complementing the original scheduler with a control mechanism able to tune its parameters, which is the classical adaptation approach, for example seen in Batcher and Walker (2008); Lu et al. (2002); Xu et al. (2006); Palopoli and Abeni (2009); Cucinotta et al. (2010); Abeni et al. (2002); Xia et al. (2007).

[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.

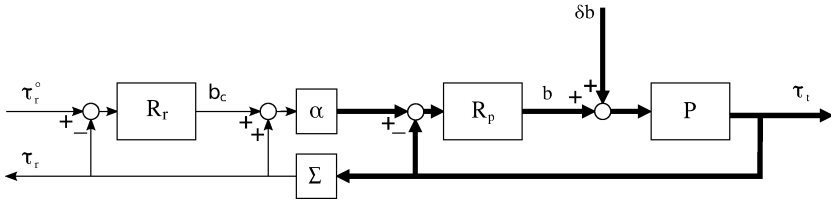


Fig. 1. The proposed "scheduler as controller" scheme.

2.1 Synthesis

The starting point for this work is Leva and Maggio (2010), where a single-processor multitasking system with a preemptive scheduler is considered, and the N processes to be scheduled are activated circularly, assigning them a certain amount (possibly zero) of CPU time. Feedback is introduced in that said CPU time amounts come from the computation performed by a controller. To briefly review the matter, define the "scheduling round" as the time between two subsequent scheduler interventions, and let $\tau_p(k) \in \Re^N$, $\tau_r(k) \in \Re$, $b(k) \in \Re^N$, $\delta b(k) \in \Re^N$ represent, respectively,

- the CPU times *actually* allocated to the processes in the k-th scheduling round,
- the *actual* duration of the k-th round,
- the CPU times or *bursts* assigned to the processes at the k-th round,
- the disturbances possibly acting on the scheduling action during the k-th round, that appear as a difference between the burst and the CPU time actually consumed by the process,

Denoting by t the total time *actually* elapsed from the system initialisation, the scheduling problem can be addressed by describing the controlled system (the process pool) with the simple model

$$\begin{cases} \tau_p(k) = b(k-1) \\ \tau_r(k) = r_1 \tau_p(k-1) \\ t(k) = t(k-1) + r_1 \tau_p(k-1) \end{cases} \quad (1)$$

where r_1 is an all-ones row vector of length N. The scheduler is then synthesised as a cascade controller, where the internal loop manages the CPU time distribution among the processes within the round, and the external one controls the time between two subsequent scheduler interventions (i.e., the desired "round duration"). In the cascade structure, the internal loop is controlled by a diagonal I regulator, while a PI SISO one is used in the external loop; this motivates the name I+PI.

Indicating with τ_r° the required round scheduling duration, and with

$$\alpha \in \Re^N, \quad \alpha_i \geq 0, \quad \sum_{i=1}^{N} \alpha_i = 1 \quad (2)$$

the vector containing the required CPU time fractions to be allocated to each process, it is quite intuitive to see that virtually any specification on fairness, tardiness, and so forth, can be expressed in terms of the two references τ_r° and α above. The "scheduler as controller" scheme is thus represented by Figure 1, where an appropriate choice of the R_r and R_p regulators allows to attain the round duration set point, and the desired CPU percentages.

For details on the synthesis the reader can refer to Leva and Maggio (2010). Suffice to say here that the single-input, single-output system with input b_c and output τ_r seen by R_r in figure 1 has the transfer function

$$\frac{T_r(z)}{B_c(z)} = \frac{k_{pi}}{z(z-1)} \quad (3)$$

while the closed-loop transfer function from τ_r° to τ_r is

$$\frac{T_r(z)}{T_r^\circ(z)} = \frac{k_{pi}k_{rr}(z-z_{rr})}{z^3 - 2z^2 + (1+k_{pi}k_{rr})z - k_{pi}k_{rr}z_{rr}} \quad (4)$$

and the choice of the parameters k_{pi}, k_{rr}, and z_{rr} is easily related to the desired closed-loop behaviour thanks to the simplicity of (1). In the example, $k_{pi} = 0.5$, $k_{rr} = 0.9$, and $z_{rr} = 0.88$.

The availability of the round duration and CPU distribution set points allows to formulate a variety of scheduling objectives, since the former is related to responsiveness and the latter to possibly weighted fairness Maggio and Leva (2010). Also, the extreme simplicity of (1) allows for an almost model-free synthesis of R_r and R_p regulators, and to efficiently simulate the control system so as to verify that the requirement specifications are met prior to implementation.

2.2 Implementation

The full scheduler implementation is outlined in Figure 2. The scheduler can be divided in two parts:

- The *I+PI* controller algorithm, described in Leva and Maggio (2010) and summarised in 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 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 can be checked formally. 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

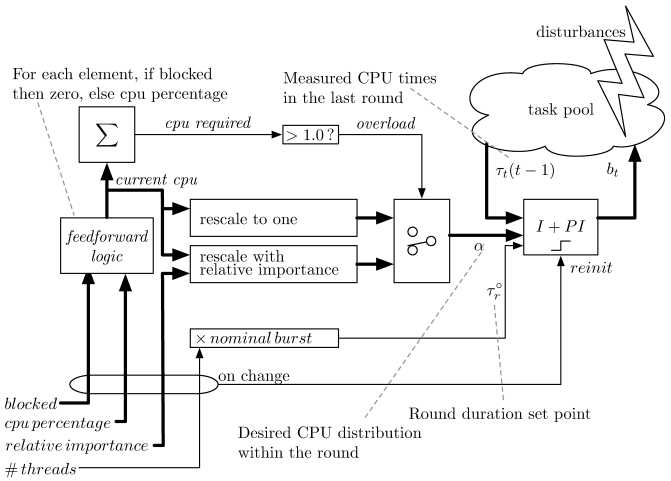


Fig. 2. Implementation scheme, containing the I+PI regulator and the set point generator.

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.

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

2.3 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.

2.4 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, 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 to the saturation value. 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 3.

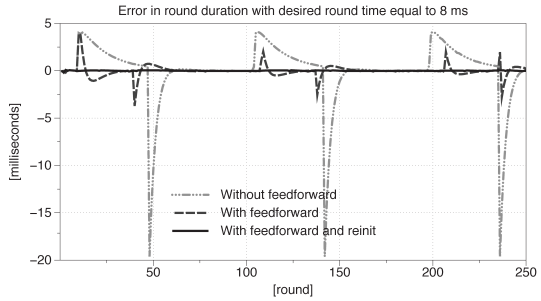


Fig. 3. Effects of task blockings on the round duration control (data from hardware implementation).

3. RESULTS

The I+PI algorithm, together with the classical RR (Round Robin) and EDF (Earliest Deadline First) scheduling policies, has been implemented in Miosix. 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. Two test sets are presented. The first 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. The second one considers the apparently off-design condition of a task pool not schedulable owing to CPU over-utilisation, where the approach behind I+PI makes it inherently superior to non control-theoretic ones.

3.1 Test set 1 (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 re-assessing 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.

Table 1. The Hartstone baseline task set.

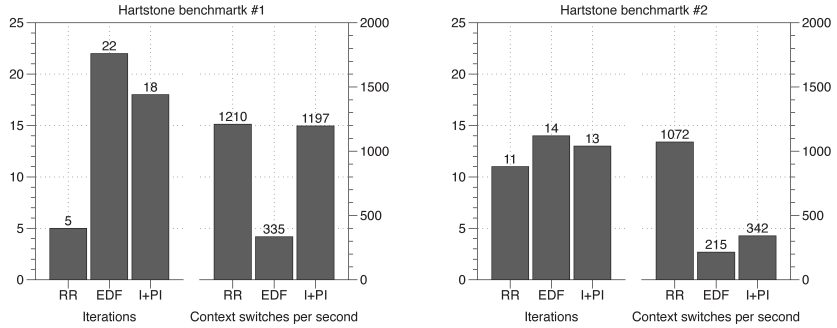
task	frequency	workload	workload rate (workload/period)
1	2 Hertz	32 Kilo-Whets	64 KWIPS
2	4 Hertz	16 Kilo-Whets	64 KWIPS
3	8 Hertz	8 Kilo-Whets	64 KWIPS
4	16 Hertz	4 Kilo-Whets	64 KWIPS
5	32 Hertz	2 Kilo-Whets	64 KWIPS

Table 1 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 non balanced 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.

Figure 4 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.

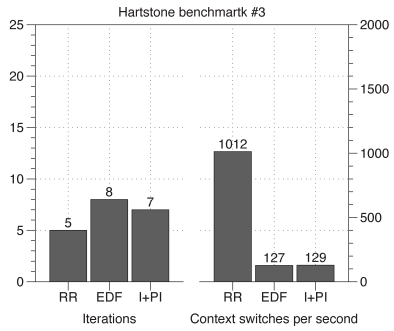
3.2 Test set 2 (extended Hartstone benchmark)

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 over-utilisation, or even to withstand a long-lasting over-utilisation by maintaining the timely operation of certain tasks. Intuitively, the general approach behind I+PI is well suited to address such issues. In a view 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 5 the way of increasing the system load is the same of the corresponding tests of Figure 4. 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 5(a), 5(b), 5(c) and 5(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.

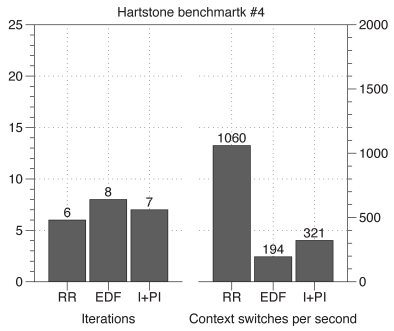


(a) Hartstone test 1

(b) Hartstone test 2

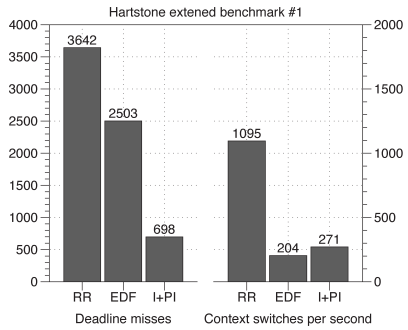


(c) Hartstone test 3

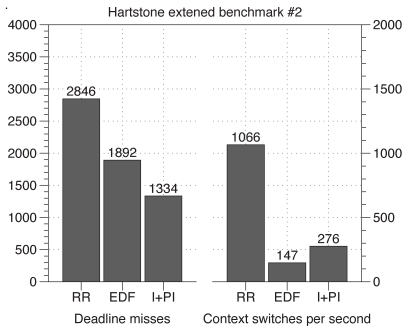


(d) Hartstone test 4

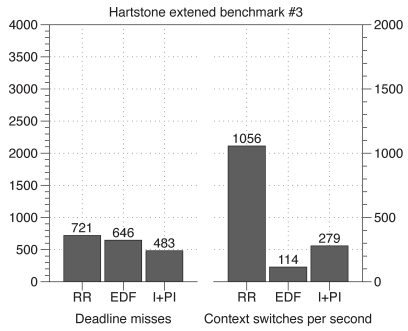
Fig. 4. Results for the Hartstone benchmark Weiderman and Kamenoff (1992).



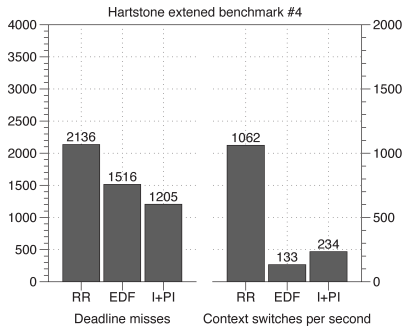
(a) Extended Hartstone test 1



(b) Extended Hartstone test 2



(c) Extended Hartstone test 3



(d) Extended Hartstone test 4

Fig. 5. Results for the extended Hartstone benchmark.

4. DISCUSSION

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 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.

Although from a control-theoretical point of view the proposal reported herein is extremely simple, it is definitely an innovation if compared to how operating system schedulers are typically designed. In fact, in that design process one typically starts with some objective expressed in common language, such as "the CPU must be distributed evenly" or "no deadlines must be missed if the pool is schedulable". From said objectives, most frequently the designer *attempts* to figure out an algorithm that solves the problem, implements it, performs the convenient checks, and if something does not go as expected, thinks and figures out another algorithm. In the process just sketched, in the authors' opinion, there are some relevant flaws. First, it is sometimes unclear how to formally represent the requirements and the designed system so that the former can be assessed prior to realising the latter. There are some analysis tools, but hardly anything capable of dealing with the online system behaviour. Second, and somehow a consequence, there is no guidance for the designer except experience and intuition. Third, interventions are made on the code directly: coupled with a certain overemphasis on the quest for reuse, this too frequently results in extremely cumbersome implementations (recall the Linux scheduler blast). In fact, all the above can be summarised in a single statement. A control problem is being solved, but in the classical procedure there is no evidence of the fundamental elements of a formalised control problem. All in all, this is how "unnecessary physics" is brought into the system, making it difficult to control unless said useless physics is preliminarily removed.

Coming to more implementation-related facts, it must be acknowledged that adopting the design paradigm shift here envisaged may not be cost-free at all. The "non formal" practices classically adopted have led to system (think again of the Linux kernel) that are extremely unsuited to being modelled in control-theoretical terms. In a control system correctly specified, replacing a control strategy with another (simplifying for brevity) is basically a matter of modifying blocks and lines in a block diagram. Interacting with modern kernels is definitely not that easy: many functions are used in many different context for different purposes, for example, so that too often a local intervention has undesired and unforeseen effects. This is why the Miosix kernel was selected, and is progressively becoming a test bed to show how control-grounded engineering can help at virtually any design level.

5. CONCLUSIONS

In this paper a control-theoretical approach to task scheduling was presented, leading to a novel design process. With the traditional *modus operandi* of the computing system community, 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.

With the proposed *control*-centric design perspective, the design cycle involves only models, thereby resulting faster and generally leading to simpler algorithms. Also, it was shown 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.

In this work, the I+PI scheduler was implemented on real hardware within the Miosix kernel. Its behaviour was tested through the 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.

REFERENCES

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

Batcher, K.W. and Walker, R.A. (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, 260–263. ACM, New York, NY, USA. doi:http://doi.acm.org/10.1145/1403375.1403438.

Brucker, P. (2007). *Scheduling algorithms*. Springer.

Cucinotta, T., Palopoli, L., Abeni, L., Faggioli, D., and Lipari, G. (2010). On the integration of application level and resource level qos control for real-time applications. *IEEE Transactions on Industrial Informatics*, 6(4), 479–491. doi:10.1109/TII.2010.2072962.

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

Lu, C., Stankovic, J.A., Son, S.H., and Tao, G. (2002). Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23, 85–126. doi:http://dx.doi.org/10.1023/A:1015398403337.

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

Palopoli, L. and Abeni, L. (2009). Legacy real-time applications in a reservation-based system. *IEEE Transactions on Industrial Informatics*, 5(3), 220–228. doi:10.1109/TII.2009.2026272.

Pinedo, M. (2008). *Scheduling Theory, Algorithms, and Systems*. Springer, third edition.

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

Xia, F., Tian, G., and Sun, Y. (2007). Feedback scheduling: an event-driven paradigm. *SIGPLAN Notice*, 42(12), 7–14. doi:http://doi.acm.org/10.1145/1341752.1341753.

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