POLITECNICO DI MILANO

Corso di Laurea specialistica in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

Control based design of OS components

Relatore: Alberto Leva

Correlatore: Martina Maggio

Tesi di Laurea di:

Federico Terraneo, matricola 750011

Anno Accademico 2010-2011

Contents

1 Introduction **10**

1.1 Brief literature review 12

1.2 Contributions of this work 15

1.3 Thesis organization 16

2 Designing software components as controllers **17**

2.1 The scheduling problem, revisited 19

2.1.1 Task model 20

2.1.2 Task pool model 20

2.2 I+PI, theory of operation 21

2.2.1 Regulator requirements 21

2.2.2 Burst control 22

2.2.3 Round control 23

2.2.4 Saturations 24

2.2.5 I+PI overview 25

2.2.6 Simulation 26

2.2.7 Implementation 28

2.3 Set point generation 28

2.3.1 τ_r selection 29

2.3.2 Round partitioning 30

2.3.3 Dynamic performance improvements 33

3 The Miosix kernel **35**

3.1 Pluggable scheduler API 36

3.1.1 Compile time versus run time pluggable schedulers . . 37
3.1.2 The Miosix scheduler API 38
3.1.3 Implemented schedulers 39
3.2 I+PI implementation in Miosix 40
3.2.1 Sensors and actuators 40
3.2.2 Context switch implementation 41
3.3 Effort on standard compliance 43
3.3.1 Focus on C++, not just C 44
3.3.2 POSIX thread API 45
3.4 Thread safety of the C and C++ standard libraries 47
3.4.1 FILE * locking 49
3.4.2 Static constructors 49
3.4.3 C++ exception handling 50
3.4.4 Reference counting within libstdc++ 50
3.5 Additional kernel features 51
3.6 Comparison with other OSes in the same class 52

4 Benchmarks **54**
4.1 Correctness check: MiBench 55
4.2 Scheduler correctness: Parallel MiBench 58
4.3 Hartstone 59
4.4 Extended Hartstone 62

5 Conclusions **65**
5.1 Future directions 65

Bibliography **67**

List of Figures

2.1 The bank of integral regulators to control task bursts. 23
2.2 The complete structure of the I+PI scheduler. 24
2.3 Simulation results of the I+PI scheduler with a pool of three tasks. 27
2.4 The I+PI scheduler attached to its set point generator. 30
2.5 Comparison of the effect of task blocking with different countermeasures (data from Miosix implementation). 34

3.1 The architecture of the Miosix kernel 36
3.2 The pluggable scheduler API in Miosix 39

4.1 The stm3210e-eval board, used to perform the benchmarks. . . 55
4.2 Results for the Hartstone benchmark [28]. 61
4.3 Results for the extended Hartstone benchmark. 64

List of Tables

1.1 Summary of control functionalities introduced in related work. 13

3.1 Microcontroller kernel comparison 53

4.1 Summary of relevant MiBench [14] execution data. 57
4.2 Summary of simultaneous MiBench [14] execution times [s]. . . 59
4.3 The Hartstone [28] baseline task set. 60

Abstract

This thesis is part of a long-term research, the ultimate goal of which is the introduction and exploitation of control based methods and tools in the *design* of computing system components. The major claim of this work is to support –by means of experimental evidence gathered on real hardware and software– that the proposed approach is indeed a step forward from the design viewpoint. To support that claim, the typical control design procedure was applied to a software problem: preemptive task scheduling, by

- first modeling the problem in system-theoretical terms,
- then designing a solution in the so chosen modeling formalism,
- and only in the end implementing the solution as an algorithm in some programming language.

Such an approach is applied in this thesis, in all of its phases. The result is the design of a "control theoretical" scheduler named I+PI for reasons that will become clear later on.

In fact, the adoption of control theoretical methods not only generally leads to improved performance and increased robustness to unpredicted run-time situations, but also yields the ability to *formally prove* the properties of interest for the control system.

The thesis is organized as follows. Chapter one provides an introduction to the problem and a minimal literature review. Chapter two presents in detail the I+PI scheduler. In chapter three the Miosix kernel is introduced, detailing its innovative features with particular emphasis put on its

suitability for the benchmarking of schedulers. Chapter four is dedicated to the performed benchmarks, and chapter five draws conclusions regarding the presented work, and outlines future directions.

The obtained results can be summarized as the successful implementation of a control theoretical scheduling policy on a real architecture. This proves that previous results –more "theoretical" than those of this work– can be applied provided that the underlying engineering aspects are dealt with correctly. All of this allows to hope that the research path followed herein will further provide good results.

Italian abstract

Questa tesi fa parte di un lavoro di ricerca a lungo termine, il cui fine è l'introduzione dei metodi della teoria del controllo nella *progettazione* di componenti software. Il principale contenuto innovativo di questa tesi è quello di dimostrare –attraverso prove sperimentali– che l'approccio proposto è effettivamente un passo avanti rispetto alle usuali tecniche di design. A sostegno di tale affermazione, la tipica procedura di progettazione dei sistemi di controllo è stata applicata ad un problema software: lo scheduling dei task in un sistema operativo,

- modellando prima il problema in dal punto di vista teorico,
- poi progettando una soluzione nel formalismo scelto,
- e solo alla fine, implementando la soluzione attraverso un algoritmo in un linguaggio di programmazione.

Tale approccio è stato applicato in questa tesi in tutte le sue fasi. Il risultato è la realizzazione di un "controllore-scheduler" chiamato I+PI per ragioni che verranno esplicitate in seguito.

Infatti, l'adozione della teoria del controllo non solo porta al miglioramento delle prestazioni e della robustezza a situazioni impreviste, ma consente anche di provare formalmente proprietà di interesse per il sistema di controllo.

La tesi è organizzata come segue. Il primo capitolo fornisce un'introduzione al problema e una revisione della letteratura scientifica. Il capitolo due presenta in dettaglio lo scheduler I+PI. Nel terzo capitolo viene introdotto il

kernel Miosix, presentando le sue caratteristiche innovative, con particolare attenzione alla sua idoneità per il benchmarking degli scheduler. Il quarto capitolo è dedicato ai benchmark eseguiti, infine il quinto capitolo trae le conclusioni del lavoro svolto.

I risultati ottenuti possono essere riassunti come l'effettiva implementazione di uno scheduler basato sulla teoria del controllo in un kernel. Ciò dimostra che i risultati precedentemente ottenuti –più "teorici" rispetto a quelli di questo lavoro– possono essere effettivamente applicati in pratica. Tutto questo permette di sperare che l'argomento di ricerca qui presentato possa fornire ulteriori buoni risultati.

Chapter 1

Introduction

Feedback control of computing systems is nowadays an important research subject, and some of the related problems (not all, as will be shown here) have been receiving much attention; the book [16] is quite clear an evidence of the statement just given.

One of the main ideas behind such a huge research *corpus*, quoting again [16, p. xv], is that "by understanding the essential elements of control theory, computing practitioners can design systems that adapt in a more reliable manner".

As can be seen by means of an even short glance at the literature, many attempts have been made to apply control techniques to computing systems by following the idea sketched above; a full review on the matter is impossible to give here. However, to the best of the author's knowledge, one word in the statement quoted above has not yet been realized, and that word is "design". In fact, virtually the totality of the documented research starts from some computing system *already designed and functional*, obtained with limited (if any) control theoretical foundation. Then, said system is taken as "the plant" in the sense given to the term in the control domain, and (simplifying for brevity) some loops are closed around that plant to attain the desired specifications. Doing so is certainly a straightforward way to use control tools in computing systems, and the research presented here is in no

sense a criticism to that *modus operandi*. However, strictly speaking, the mentioned *modus operandi* is not *designing* a computing system by means of the control theory.

Moreover, among the control application domains, the computing system one is unique. In fact, the "plant" and the "controller" are homogeneous (basically software) objects, making it possible and natural to design the overall system in a unitary manner right from the beginning. Also, computing systems are inherently immune from most of the problems encountered in other contexts. For example, measurement errors and state variables' inaccessibility are of very limited (if any) concern. Indeed, in computing systems the idea of "process/control codesign" can find maybe the most effective realization.

This manuscript applies the idea above to the context of preemptive feedback scheduling in uniprocessor multitasking systems, and its contribution can be summarized as follows.

First, based on some previous results, a control theoretical solution to the addressed problem is devised, leading to the necessary controller *model* to allow control system analysis and simulation tests.

Then, said controller model is turned into a real-life scheduling *algorithm* by following substantially the same process used to turn a model into a controller in any other domain, incidentally showing that many of the involved quantities and phenomena do admit a system theoretical counterpart also at this level.

Finally, experimental evidence is obtained for the expected results. This implies realizing the solution on a real kernel, applying classical and well recognized benchmarks to allow fair comparisons with alternative proposals, and also introducing some benchmark extensions to allow appreciating some aspects of the problems that simply cannot emerge if a system theoretical attitude is not taken.

1.1 Brief literature review

Generally speaking, "closing a loop" in some software system, and especially in a scheduling algorithm is not a new concept. Probably, the first scheduling algorithm based on the concept of "feedback" is the *Multilevel Feedback Queue*, described, together with others in [9]. 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 the highest priority non-empty 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 ARTIST2 project was aimed at defining a roadmap on control of real-time computing systems [7]. In the vast majority of cases, the controlled item is the allocation of computing and communication resources. 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 will address this limitation at least for the specific problem of CPU allocation.

As another example of introducing adaptivity at the operating system level, [30] controls CPU utilization of a virtual server running a web server with various model predictive control strategies, and compare them. Differ-

ently from the quoted work, the purpose of this thesis is not to complement an existing operating system but to re-design part of it with a control oriented attitude.

	Time computation	**Task selection**	**Activation**	**Task admission**
I+PI	×	×	×	
[4]	×			
[8]		×		
[10]			×	
[11]	×			
[12]	×			
[17]				×
[18]				×
[19]	×	×		
[20]	×			
[23]				×
[22]				×
[25]	×			
[29]			×	

Table 1.1: Summary of control functionalities introduced in related work.

Focusing on the specific topic of scheduling, in the case of single processor (possibly real-time) operating systems addressed herein, the Earliest Deadline First (EDF) has been proved to be theoretically the optimal scheduling algorithm whenever the device is underloaded [26]. This means that EDF is able to schedule every task pool that is schedulable. Such off-line considerations are however not suited to address online behaviour. In fact, the performance of EDF decreases exponentially when the device becomes slightly overloaded. For completeness, in this work the proposed scheduler will also be compared 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 [19].

Apparently, [22] is very close to this 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 I+PI. Notice that admission control in general is very popular in the context of web servers, where the tasks could be rejected to preserve utilization [17, 18]. 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 thesis is the application domain, in one case it being a server and in the other an embedded device.

Some papers introduce controllers to adjusts the "burst", i.e., the times assigned to the tasks, with the purpose of keeping the system utilization below a specified upper bound [4, 11, 12, 20, 25]. 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. 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 all bursts at once.

On a different page, the authors of [8] reorder the list of tasks to be scheduled with a Round Robin (RR) 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.

1.2 Contributions of this work

This thesis is a continuation of the author's research work started in its BsC thesis [13]. Given that, it is relevant to remark the contribution that this thesis brings to its research area, by briefly outlining here the additional work that was performed.

- A scheduler named I+PI has been implemented on a microcontroller kernel named Miosix. The I+PI scheduling algorithm is the direct realization of a control structure made of a diagonal integral regulator and a SISO PI one, whence the name.
- The suitability of the I+PI scheduler to real time systems was demonstrated, addressing specifically the ability of the scheduler to handle a pool of tasks having deadlines to meet. This is a relevant fact since the design of I+PI is *not* tied to the idea of "deadline"; on the contrary, its control theoretical origin makes it capable to handle tasks of different types and with different requirements without the need to realize and coordinate different scheduling policies—a relevant architectural simplification indeed.
- The I+PI scheduler requires inputs (e.g., a set point) that will be explained in detail later on. A systematic mapping of task requirements as seen from the computer engineering (not the control theory) point onto those inputs is crucial to achieve good performance and usability. This mapping has been thoroughly investigated, and a performant solution was found that still preserves the control theoretical design framework, namely consisting of the introduction of a feedforward control path and a controller reinitialization policy.

- The I+PI scheduler has been compared with other well known schedulers such as the Earliest Deadline First and Round Robin by means of standard benchmarks. Also, extension of said benchmarks were introduced that allow to deepen the analysis consistently with the mentioned control based design, for example to systematically investigate the effects of off-design conditions.

- In order to achieve the goal of the previous points, the Miosix kernel was extended with an API called "pluggable scheduler" that permits more than one scheduler to coexist in the same codebase, so as to allow performing comparative scheduler benchmarks under the same hardware and software conditions.

1.3 Thesis organization

This work is organized as follows.

Chapter two, after presenting the key ideas about the approach at designing computing system using control theoretical principles, focuses on describing the new approach to scheduling taken in this thesis, and presents in detail the I+PI scheduler and its set point generation logic.

In chapter three the Miosix kernel is introduced, detailing its innovative features with particular emphasis put on its suitability for the implementation and benchmarking of schedulers.

Chapter four is dedicated to the extensive benchmarks performed to compare the Miosix implementation of the I+PI scheduler with the EDF and Round Robin ones, as well as to assess the correctness of the implementation.

Lastly, chapter five draws conclusions regarding the presented work, and outlines future directions.

Chapter 2

Designing software components as controllers

The application of control theory to improve software systems is not a new approach. There are countless examples of the introduction of feedback loops in applications or OS components, see e.g. [16, 21] and the papers quoted therein. Doing so is known to improve performance, and more specifically to yield "self adaptation" (to adopt the computer engineering jargon) in the face of unpredicted run time conditions.

However, the traditional approach has been to consider a software component that performs a particular function, identify controllable inputs, find a way to measure some properties of said system which are of interest, and design a controller that acts on the inputs so that the measures follow a given reference set point. The key aspect of this methodology, is that no redesign phase of the software component is performed. Said component remains designed as before, without any use of the control theory and remains capable of correctly operating also without the controller. The controller remains just an "add-on" to improve performance, and not an integral part of the system.

Another way to express the same concept, maybe more suited for the sequel of this work, is that (feedback) control is traditionally introduced in

an *already fully functional* system. For example, when a loop is closed to adapt some parameter of an existing scheduler for some purpose irrelevant to discuss now, this does not result in a system that *requires* the added loop in order to function, but can work –without the added "adaptation", of course– also in its absence. Here, the idea is to use controllers *as* system components; in so doing the potential of the control theory can be exploited to much greater an extent than if just external loops are closed around a functional system, because said methodology is introduced right from the *design* phase.

The additional steps required are to identify within the software component the part that exhibits the behaviour that we want to control (the "plant") and the previously designed part that tries to obtain the desired behaviour (the "old controller"), and replace the old controller in its entirety with a new one, designed from the start using the control theory.

This can provide many advantages, of which one is surely the realization of simpler solutions, both in term of algorithmic complexity as well as with less lines of code. Obviously, a software system cannot be made simpler by adding additional layers, such as an additional control based controller. On the other hand, if the previously existing control logic is *replaced* with a new one, there is at least the opportunity for simplification. This opportunity often concretizes as the control schemes produced from this methodology are usually simple. For example, the core part of the I+PI task scheduler presented in this thesis is only 40 lines of C++ code.

Another less obvious form of simplification that derives from the removal of the non control theoretical control logic, is that the models of the plant, which are necessary to design a controller, become simpler, often *much* simpler.

In fact, when designing a controller for a fully functional software component, the preexisting control logic ends up being part of the plant. Now, let alone that this is contradictory from a theoretical point of view, the complexity of modeling that control logic can be greater than the one required

to model the real plant.

Taking again as example the task scheduling problem, which will be the main subject of this thesis, the plant has a very simple (e.g: linear) representation, while traditional scheduling algorithms, designed using other criteria than control theory, can result in complex (e.g: nonlinear) models.

Lastly, perhaps the most important advantage of this approach is the ability, as part of the design phase, to analyze and assess solutions without resorting to implementation and direct testing, but instead in a formal and system theoretical way. Once the problem has been suitably modeled, one can access a wide range of solutions, in the form of control structures already thoroughly characterized in terms of performance and robustness.

The resulting design phase can be therefore performed entirely on models, and after a satisfactory solution is found, the implementation phase is usually straightforward and most importantly it is done only once, avoiding continuous design cycles that modify the system in an attempt to tackle individual issues without any guarantee that such "fixes" do not introduce undesired side effects.

Now, the class of software problems that lend themselves to such an approach is quite wide, ranging from QoS middleware [21], to flow control [4], adaptive video streaming [6], etc. In this thesis, the attention will be focused on the problem of task scheduling within an operating system, a problem where the introduction of control loops to enhance classic schedulers has already been attempted [5], but that, to date, has never been tackled with an entirely control designed scheduler.

2.1 The scheduling problem, revisited

As previously mentioned, the innovative approach at designing software systems using the control theory starts with the selection of a suitable modeling formalism to represent the problem, and it is from here that this section starts.

The problem used as example in this thesis is the preemptive task scheduling problem on uniprocessor systems. The phenomenon that a task scheduler is responsible to control is a pool of tasks competing to be executed on the available CPU. Having considered the preemptive case, the scheduler can both give and take the CPU to/from tasks; in essence, the scheduler can assign the CPU to a task for a given period of time, that will be called "burst" from now on, after which the task is preempted.

2.1.1 Task model

The task can therefore be modeled as a discrete time dynamic system, with the following equation

$$\tau_t(k) = b(k-1) + \delta b(k-1) \tag{2.1}$$

where $b(k-1)$ is the burst value the scheduler assigned to the task and $\tau_t(k)$ is the actual time the task has occupied the CPU. The disturbance $\delta b(k-1)$ is used to model any reason why the actual burst differs from the nominal one (including the task yielding, sleeping or locking while performing some I/O operation or synchronization primitive). Lastly, the index k is used to count the scheduler interventions.

2.1.2 Task pool model

Now that a task has been modeled, it is necessary to extend the model to the entire task pool. This can be easily done resulting in the following system of equations

$$\begin{cases} \tau_t(k) &= b(k-1) + \delta b(k-1) \\ \tau_r(k) &= \sum \tau_t(k) \\ t(k) &= t(k-1) + \tau_r(k) \end{cases} \tag{2.2}$$

For generality, in this model it is assumed that one scheduler intervention can assign bursts to more than one task, from a minimum of one up to the number of tasks effectively present in the pool. Given that there is only

one CPU core, the tasks will be assigned sequentially to the CPU, in some unspecified order, and run each for their assigned burst (plus or minus their disturbance). Note that this is a pure generalization, and that the model can still be used with schedulers that assign the CPU to only one task per scheduler intervention, as it suffices to set all burst values to zero except the selected task.

Other that that, there are a number of interesting comments that can be made regarding this model. First, despite being a discrete time dynamic system, its index k does not represent time but scheduling interventions. This explains the need for the third equation to describe the evolution of time. Second, it is a linear system, except for the not expressed constraint that in a realization of such a model the disturbance magnitude is bounded in one direction, as $b(k) + \delta b(k)$ cannot be negative.

It is interesting to note that simple extensions of this formalism would allow to represent existing scheduling algorithms and simulate their behaviour [24], but this is not the goal of this thesis, so we'll move immediately to the implementation of a new scheduler.

2.2 I+PI, theory of operation

Now that a model of the plant is available, a controller can be designed to impose the desired behaviour. The next step is therefore to identify said desired behaviour.

2.2.1 Regulator requirements

First, given that there is a number of tasks all of which needs the CPU, it sounds reasonable to periodically assign the CPU to each of them. Therefore, by taking advantage of the formalism's ability to assign the CPU to more than one task, a scheduler is designed that can compute a burst value for each of them in only one scheduler intervention. As a result, the tasks will be executed one after the other each for their actual burst τ_t. The name

"round" was given to such sequential execution of tasks without scheduler intervention. The round duration is, from the same equation of the model, τ_r.

Having chosen this execution model, the first desire that one may want is control over the round duration. It is in fact intuitive that this value has implications in the tradeoff between system responsiveness and scheduling overhead, as it basically determines the number of context switches per amount of time. Other than that, not all tasks in the pool are equal, some may be carrying out heavier operations and therefore would require a greater share of CPU than the others. It is therefore desirable the possibility to redistribute the round among tasks in different ways.

2.2.2 Burst control

Now that the requirements are clear, it is finally possible to start designing a controller. To do so, first notice that setting the round time to a desired set point value, and setting the round distribution among tasks means essentially assigning a set point for the burst of each task. Therefore a controller is needed to maintain the actual bursts as close as possible to their set points. In this work an integral regulator is chosen, for its simplicity and ability to achieve zero steady-state error. Thanks to the concept of disturbances introduced earlier to abstract –among other– task synchronization, the model for the individual tasks is decoupled, so a diagonal integral regulator can be used having as inputs the burst set points τ_t° and the actual measured bursts τ_t, and as output the computed bursts b. The resulting control structure, up to now, is illustrated in Figure 2.1, where thick lines indicate vector signals.

A diagonal integral regulator has the following time domain equation

$$b(k) = b(k-1) + k_{pi} I(\tau_t^\circ(k-1) - \tau_t(k-1)) \tag{2.3}$$

where I is the identity matrix whose dimension equals the size of the task pool and k_{pi} is the integrator gain. Given that all the single integral regulators that compose the diagonal one have the same eigenvalues, it is possible to

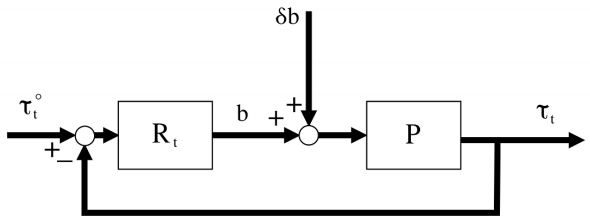


Figure 2.1: The bank of integral regulators to control task bursts.

consider only one of them for the selection of k_{pi}. Simple computations lead to the following (scalar) closed loop transfer function

$$\tau_t = \frac{k_{pi}}{z^2 - z + k_{pi}} \tau_t^\circ + \frac{z-1}{z^2 - z + k_{pi}} \delta b \tag{2.4}$$

where it is clear that the poles of the diagonal regulator are in $0.5 \pm \sqrt{0.25 - k_{pi}}$. This helps in selecting a proper value for k_{pi}; both the values 0.25 (two poles in 0.5) and 0.5 (poles in $0.5 \pm 0.5i$) have provided good results, with the second one being preferred.

2.2.3 Round control

The next step is controlling the round duration. To do so the individual τ_t can be summed obtaining τ_r, which together with the set point τ_r° allow to make a controller also for this part. Now, the output of this controller is a round value, and to match it into the individual bursts adaptation is required, so a parameter named α which dictates the round distribution within tasks is introduced. The resulting controller has a cascaded structure outlined in Figure 2.2.

The last choice to be made is the selection of the R_r block. To do so, it is necessary to compute the transfer function between b_c and τ_r which is

$$\frac{T_r(z)}{B_c(z)} = \frac{k_{pi}}{z(z-1)} \tag{2.5}$$

Note how the transfer function does not depend from the α parameter.

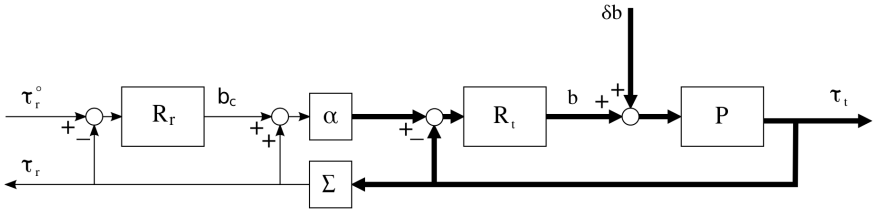


Figure 2.2: The complete structure of the I+PI scheduler.

Given that integral regulators are already available in the inner loop one may be tempted to use a proportional regulator, obtaining the I+P scheduler. This scheduler has actually been implemented (in the BsC thesis [13]) but has problems in maintaining the round set point if one or more tasks blocks. By "blocking" it is intended a sufficiently long period of time (greater than one scheduling intervention) where a task refuses the CPU. This can happen in a real system if a task sleeps, or locks in a I/O operation or synchronization primitive. The solution is to employ a PI controller, hence the I+PI name for the scheduler implemented as part of this thesis.

The time domain equations of the PI regulator used are here reported

$$\begin{cases} x_R(k) &= x_R(k-1) + k_{rr}(1+z_{rr})(\tau_r^\circ(k-1) - \tau_r(k-1)) \\ b_c(k) &= x_R(k) + k_r r(\tau_r^\circ(k) - \tau_r(k)) \end{cases} \tag{2.6}$$

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

$$\frac{T_r(z)}{T_r^\circ(z)} = \frac{k_{pi}k_{rr}(z - z_{rr})}{z^3 - 2z^2 + (1 + k_{pi}k_{rr})z - k_{pi}k_{rr}z_{rr}} \tag{2.7}$$

The choice of the parameters k_{rr} and z_{rr} is not treated here, but the interested reader can find it in [24].

2.2.4 Saturations

As always when using integral regulators, it is required to either prove that the regulator will never be subject to a constant error, or employ a proper saturation scheme to avoid their state variable diverging to infinity. Now, the

diagonal integral regulator in the inner loop will surely experience a constant error whenever a task blocks. Therefore, in the actual implementation the $b(k)$ state vector is clamped between two limit values. The upper bound is the maximum time a task can run without preemption, and is chosen in order to maintain responsiveness. The lower bound can be zero, enforcing the physical constraint that negative bursts are not allowed, or can be a small positive value to avoid assigning the CPU to a task for such a small time that is comparable to the context switch time.

Saturation of the outer integral regulator is a little trickier. Each inner integral regulator has an additional positive saturation signal that is asserted whenever said event occurs. The logical and of those signals is used to avoid the external integrator runaway. Whenever all the inner regulator have reached the positive saturation condition, the state variable of the outer regulator can only decrease, and not increase. The lower saturation bound is instead set to clamp b_c to $-\tau_r$ so that the round time value passed to the inner loop remains positive.

2.2.5 I+PI overview

After the design phase, we are left with a block diagram representing the I+PI regulator. This *scheduler* is a cascade regulator having a diagonal integral regulator in its inner loop and a PI regulator in its outer loop. It takes as inputs the $\tau_t(k)$ vector which are the measures of actual task execution time in the last round and produces $b(k+1)$, the desired bursts for the next round, therefore this scheduler runs only once per round computing all the next bursts at once.

It has a set point, τ_r° which is the round duration set point and a switching parameter, α, that specifies how the round is going to be distributed among tasks. While the τ_r° set point is self explanatory, it is maybe better to further explain the use of the α parameter. For proper operation of the scheduler, the sum of the elements of the α parameter should always be one, and the individual vector elements represent the percentage of the round duration

that will be given to each task. Therefore, to give an equal share of CPU to a pool of three tasks, α should be equal to $[0.33\ 0.33\ 0.33]'$, while to give 50% CPU time to the first task of a pool of six tasks, and to share the rest of the round equally, one can set α to $[0.5\ 0.1\ 0.1\ 0.1\ 0.1\ 0.1]'$. The α parameter is said to be a *switching* parameter as it is allowed to change while the regulator is operating. This makes it possible to change the distribution of the CPU among tasks on the fly. Also, adding and removing tasks is equally straightforward, requiring only to add/remove one integral regulator in the inner loop, together with its state variable.

Recall again that in any round, some of the computed task bursts can be zero, which means that those tasks will not be activated at all. Therefore, although we adopted the term "round" for uniformity with the literature, its meaning in this context is *not* the same as in other scheduling policies, such as the Round Robin.

2.2.6 Simulation

To show the advantages of using control theory for the design of a software component, the presented I+PI scheduler is here simulated, without the need for an implementation, by just using the previously written time domain equations for the scheduler as well as for the task pool model.

Figure 2.3 shows the I+PI ability to maintain its set point even in the presence of disturbances. A set of three tasks is considered, both the round time set point and CPU distribution is changed during the simulation, and step like disturbances have been introduced.

This allows to assess the scheduler's performance and to tune its parameters (in this case k_{pi}, k_{rr} and z_{rr}), in other words to have an idea of how the scheduler will work once implemented.

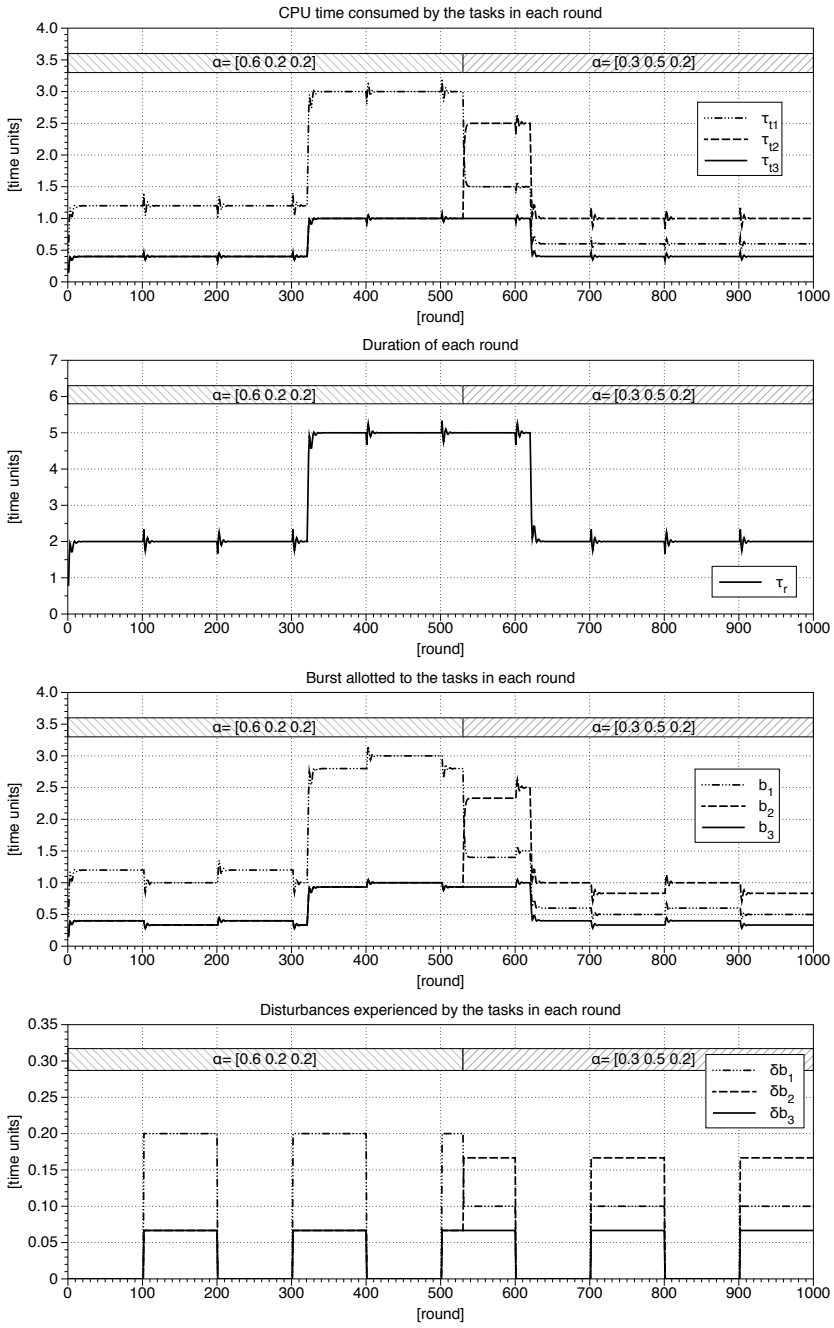


Figure 2.3: Simulation results of the I+PI scheduler with a pool of three tasks.

2.2.7 Implementation

Another advantage of this approach is the ease of implementation. Once the scheduler structure has been found in the design phase, and after it has been assessed and tuned by means of simulation, the implementation is unambiguously obtained from the scheduler's equations.

Algorithm 1: I+PI scheduler

```
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 τ_t(k − 1)
    Compute the measured duration of the last round as τ_r(k − 1) = Σ_{i=1}^{N_t} τ_{t,i}(k − 1)
    Read the required round duration τ_r°(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(τ_r°(k − 1) − τ_r(k − 1)) − k_rr z_rr(τ_r°(k − 2) − τ_r(k − 2))
        Apply saturations to b_c(k)
        Compute the vector α(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:
                τ_{t,i}°(k) = α_i(k)τ_r°(k)
                b_i(k) = b_i(k − 1) + k_it(τ_{t,i}°(k) − τ_{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
```

The Algorithm 1 is a pseudocode implementation of the I+PI scheduler. It is of course short being a pseudocode, but also the real C++ implementation within the Miosix kernel is.

2.3 Set point generation

The I+PI scheduler as described until now is the core of the scheduler or, in control theoretical terms, the closed loop part. The full scheduler includes an additional part, the set point generation logic, whose purpose is the gen-

eration of the τ_r° set point and the α parameter starting from the tasks' needs.

As the reader may have noticed, the term "set point generation" contains a slight terminology abuse, as strictly speaking, τ_r° is a set point for a linear time-invariant control system, while the input α can be either regarded as a parameter, thus making the system time-varying, or as an input, which makes the same model nonlinear.

However, it can be proven that the control system is asymptotically stable for every α, and given that in the real world everything is quantized, the same system can be thought of as a switching one. In the case of such a system –incidentally, with system-independent switching signal– the asymptotic stability of all possible dynamic matrix ensures the existence of a finite dwell time for the system to remain stable under switching. As such, if one simply assumes that α is changed "sparingly" amidst the scheduler's operation, everything is safe. Note also that the discrete time in this work counts the scheduler interventions, not a fixed period. Hence, if some dwell time estimate were available –which is certainly feasible but was not realized since the resulting computational burden would impair most of the advantages– it would be enough to lower the round duration set point to have the scheduler intervene more (i.e., the necessary number of) times between two subsequent α modifications—in other words, switching stability can surely be enforced, sometimes maybe at the cost of a transiently increased scheduling overhead.

As for τ_r°, its generation –no matter how done provided this does not close other loops– apparently cannot destabilize the system.

The full scheduler is outlined in Figure 2.4, where the I+PI part is drawn as a single block, having already been discussed in detail, while the set point generation logic is shown in greater detail, and will now be explained.

2.3.1 τ_r selection

Many different schemes for the selection of the desired round duration can be envisioned but experiments have shown that simple solutions are already

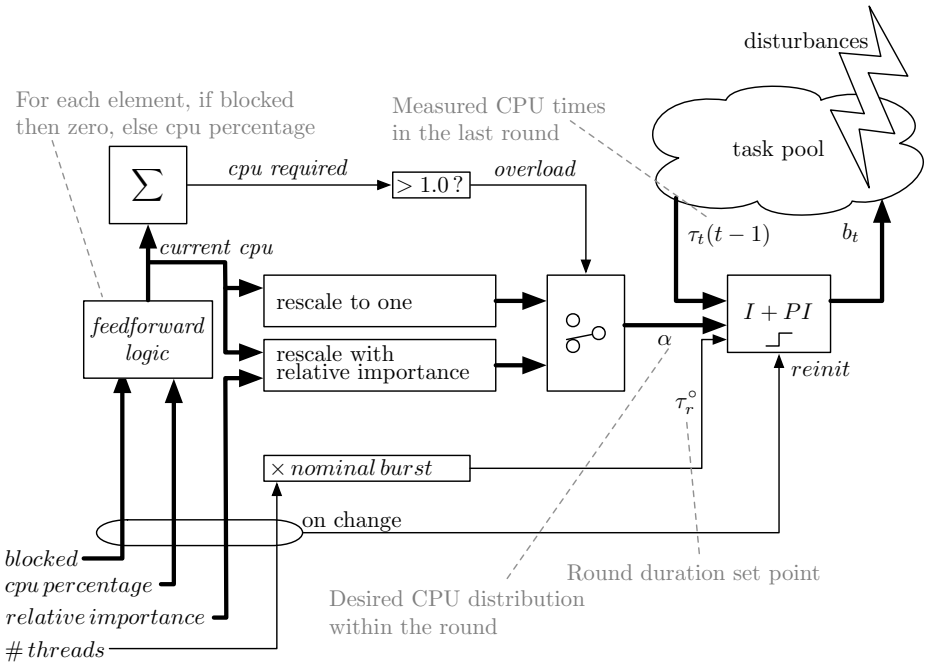


Figure 2.4: The I+PI scheduler attached to its set point generator.

more than effective.

A first approach could be setting the round set point to a constant value. While this solution is of course the simplest, it does not scale well with the number of tasks, since as more tasks are added to the system, the burst values decrease resulting in an increase of the number of context switch, and a higher overhead.

A better approach that balances simplicity and performance is to select a desired burst value and have the round set point be equal to that nominal burst times the number of tasks. This has proven to be an effective policy, so it is the one currently implemented.

2.3.2 Round partitioning

Up until now, the kernel could figure out how to set the scheduler's parameters without help from the tasks. To choose how the round should be distributed among tasks instead, the tasks should pass some information to

the scheduler. This is nothing new, and historically the most widespread solution has been to set up some form of prioritization, where tasks pass to the scheduler a priority value directly or indirectly by passing other data, such as deadlines and let the scheduler map this into a prioritization scheme. The most performant solution found to date for the I+PI scheduler, is to have tasks pass to the scheduler two information:

- The desired relative CPU time a task needs. For example a periodic task such as a video player might require, on average, a 30% share of the CPU to perform video decoding in real time, or a system administrator might assign 5% of CPU share to a long batch job to maintain system responsiveness.

- A second parameter, named "relative importance" which is used to select how the desired CPU share is going to be reduced in case the total system load becomes unfeasible.

Both parameters are not fixed at task creation, they can be changed at runtime during the task's lifetime to reflect changes in its requirements.

The first part of the set point generation logic is to map those two inputs coming from the tasks into the α parameter. At first one may be tempted to map the desired CPU percentage directly into α, but it would not work. This is because the sum of α's elements must be 1, while this is not necessarily true with the desired CPU percentages. The solution adopted is to rescale the desired CPU percentages coming from tasks so that the sum equals one. Actually, two different rescaling policies are adopted depending whether the system is underloaded or overloaded.

In the most common situation of underload, the simple "rescale to one" policy is used. Consider an example with three tasks requiring 20% CPU share each. The rescale to one policy will result in an α value of $[0.33\ 0.33\ 0.33]'$ so that each task will receive a 33% CPU share. By design, this ensures that each task will receive a CPU share greater or equal than the desired one, therefore allowing it to complete its job on time.

In case of CPU overload, the previous solution does not work. In fact the rescaling policy would give to each task *less* CPU share than the one they require, and if some of those tasks have deadlines to meet, they would not be able to do so. Therefore, in case of CPU overload a different rescaling policy is adopted, named "rescale with relative importance", which first multiplies the desired bursts by the relative importance parameter, and then rescales the result to one, using this as α parameter.

On a superficial examination, the relative importance might be confused with traditional priorities, but a deeper understanding of its potential shows it is a much wider concept, and also a more "well behaved" one. First, it should be noted that the relative importance of a task is only taken into account if CPU overload occurs, unlike priorities which are always in effect. Second, this new approach allows to *predict* the CPU share that each task will receive both when the system is underloaded or overloaded, something that can't be said for priorities. Third, the relative importance can also be used to somewhat emulate traditional priorities by restricting the possible values to zero and one, resulting in tasks with a zero relative importance not running in case of overload. If instead this constraint is relaxed, allowing any (positive) value, it is possible to implement "graceful degradation" in a way that would be difficult, if even impossible, to do with priorities.

It is also interesting to compare relative importance with deadline-based approaches, such as EDF. Exactly as before, deadline approaches do not allow to predict the CPU share tasks will receive in case of system overload. Also, not all tasks have deadlines. For example, long batch jobs to receive a CPU share in a scheduler such as the EDF, would require either setting fictitious deadlines or having two scheduling policies coexisting in the kernel, one to handle batch jobs, and one for tasks with deadlines. On the other hand, the relative importance solution makes I+PI "execution pattern agnostic", that is, capable of handling periodic, aperiodic and batch jobs, with and without deadlines, in a unitary way.

Last, notice that this solution also allows the scheduler to unambiguously detect a CPU overload situation. This is simply identified by the sum of all required CPU shares exceeding unity.

2.3.3 Dynamic performance improvements

The last detail added to the set point generation logic is an improvement in the scheduler's *dynamic* performance, that is, its ability to react quickly to changes in its set points. The main issue is always related to task blocking. When a task blocks, it stops accepting CPU bursts, and therefore its τ_t becomes zero. On the other hand, its burst set point remains unchanged, so the integral regulator that controls the task is subject to a constant error. This has already been addressed with saturations, so the integrator's state variable will reach positive saturation and then stop increasing. However, when the task unblocks, the first burst that will be assigned to it is equal to the saturation value itself, a fairly large value indeed. Even if the feedback loop will quickly regain control over both round and burst duration, these "spikes" in burst assignments whenever a task unblocks can cause problems. For example, when performing the real time benchmarks in chapter five, these spikes caused spurious deadline misses.

Solutions to this problem required the introduction of two additional components in the set point generation logic, both of which are grounded on control theoretical principles.

The first one is the "feedforward" logic. It works on the assumption that task blocking is a measurable disturbance. In fact, a task, to sleep or more in general block has to call some OS API and so the kernel (and the scheduler) are informed of this. This provides the opportunity to act directly on the set points, by setting the α element of the blocked task to zero for all the time the task is blocked. This results in a "freezing" of the integral regulator dedicated to control that task as it results in a zero error (zero burst set point due to α element being zero, and zero actual burst). The feedforward name comes from this being an open loop policy to compensate the effect of a

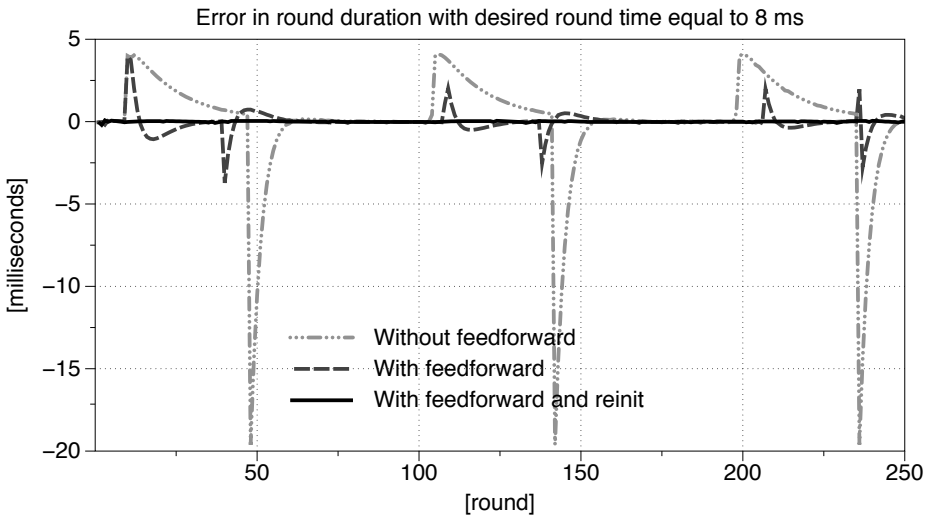


Figure 2.5: Comparison of the effect of task blocking with different countermeasures (data from Miosix implementation).

disturbance. It is also interesting to remark how this policy plays nicely with the CPU overload detection, as blocked tasks are not considered in the CPU overload count (as they temporarily need no CPU), therefore improving the correctness of this estimate.

The other policy is the "regulator reinitialization". It works by reinitializing the state variables of the I+PI regulator whenever a change in its set point happens, such as task creation/deletion or task blocking/unblocking. It is grounded on the fact that such events (that happen on a slower scale with respect to context switches) represent a change in the scenario the scheduler is operating into, and that starting from a default state is therefore better.

To show the effect of those policies, a simple experiment was performed with two tasks, one of which constantly consumes its CPU share, while the other periodically blocks. The results in Figure 2.5 show how the spikes caused by task blocking are reduced if feedforward is enabled, and are completely eliminated by the coordinated action of feedforward and reinitialization.

Chapter 3

The Miosix kernel

Miosix is an OS kernel for microcontrollers developed by the author of this thesis, that has been under active development since 2008. It is released under an open source/free software license, and can be downloaded here [3]. A chapter is dedicated to it both because it has been used to test the I+PI scheduler and compare it with other scheduling algorithms, and because it has some innovative features that are worth mentioning.

The goals of Miosix are to provide a multithreaded environment for writing applications on microcontrollers, as well as to provide full support for the C and C++ standard libraries. To achieve this goals it was decided to focus on 32 bit microcontrollers as 8 and 16 bit ones are usually too resource constrained to take full advantage of such features. An overview of the Miosix kernel architecture is represented in Figure 3.1.

The kernel provides threads, but not processes as the microcontrollers it is targeted at lack an MMU which is necessary to enforce the memory isolation processes require, as well as because the total available RAM memory in a typical Miosix system is too low to allow dynamically loading of process' text segments.

As of today, its design goals are mostly achieved, with the introduction of the standard pthread API and thread safety fixes at the standard libraries, even though there is always room for improvements, like implementing the

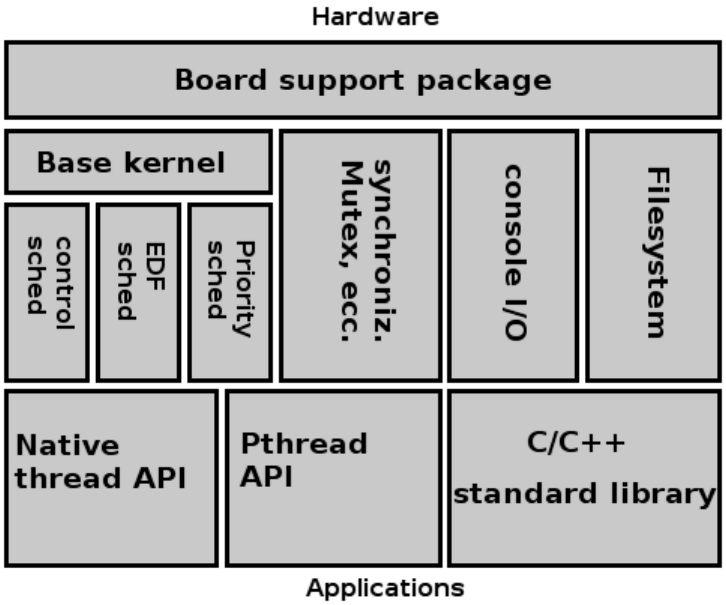


Figure 3.1: The architecture of the Miosix kernel

C++0x threading API. Development effort is currently focused on providing APIs to access complex peripheral classes such as USB controllers, displays and ethernet controllers, two of which have been already partially addressed.

This chapter is dedicated to the presentation of Miosix's distinctive features, as well as comparing them with other RTOSes of the same kind to evidence its state of the art status.

3.1 Pluggable scheduler API

In many operating systems, both targeted to microcontrollers or larger machines, the scheduler is tightly coupled with the rest of the kernel. Reasons for this design choice include attempts at improving the scheduler performance, the lack of need of multiple schedulers for the applications the kernel is targeted at, and the complexity at designing an API to separate the kernel from the scheduler.

However, especially in embedded systems, being able to choose a suitable scheduler for the application being developed can be a great benefit. And while doing research on schedulers it is even essential to compare two or more schedulers under the same hardware and software conditions to obtain meaningful results regarding their performance.

Therefore, one of the first features added to Miosix as part of this thesis is the so called "pluggable scheduler" API.

3.1.1 Compile time versus run time pluggable schedulers

The idea of having more than one scheduler in the same codebase can be implemented in three ways: the first possibility is to allow scheduler selection at compile time only, the second is to allow to switch scheduler at runtime, moving tasks from one scheduler to another during the switch, while the third one is to have more schedulers running at the same time each with its own set of tasks.

Each implementation has its advantages and drawbacks, that are quickly summarized here. At first the third possibility might seem the more flexible one, and at least for "clients" of this API, it is true. However, having more schedulers that coexist at the same time requires coordination between them, which would add much complexity to the schedulers implementation, and that complexity would likely result in an undesired coupling between different schedulers as well as a performance penalty. Additionally, the need for cooperation would somewhat limit the freedom of implementation of said schedulers, imposing uniformity for example on the way context switches are implemented or even imposing the same data structure for all schedulers. Therefore this solution is unsuitable for researching new schedulers, as it would require to write more code to implement a new scheduler, which poses difficulty in a "rapid prototyping" style of development, and the uniformity requirements is simply unacceptable.

The runtime switchable scheduler option simplifies the implementation

with respect to the coexistence option, but creates problems during the switch between one scheduler and the other. For example, in a real-time system it would be quite complex to guarantee that no deadline misses may occur during the switch. In addition, even this solution requires some runtime indirection that results in a performance penalty as well.

The compile time option, which is the one selected for implementing the Miosix pluggable scheduler API, has many advantages. First, it is well suited for embedded systems, which is the application type Miosix is targeted at. This is because it is the option with the lowest possible performance penalty (and code size penalty), as after the compile phase the kernel has only one scheduler. Also, embedded systems are used to perform specialized tasks unlike general purpose personal computers, so the need to change scheduler dynamically is usually not present. Finally, for the research and testing of new schedulers, this option offers the highest possible design freedom, allowing to change nearly everything including context switch code, data structures, etc.

3.1.2 The Miosix scheduler API

After explaining the design choices that led to the selection of a compile time pluggable scheduler API, said interface is briefly outlined, along with the list of implemented schedulers.

A scheduler in Miosix is a C++ class that must implement a specific interface, represented in Figure 3.2. To allow each scheduler to use its own customized data structure the list of currently running threads is part of the scheduler and not of the kernel, therefore the scheduler API includes functions to add and remove threads, as well as to query existing threads. Also, it includes functions to pass "hints" from applications to the scheduler. The generic term hints is here used to underline the scheduler-specific nature of this data, that for example can be the (CPU%,relative importance) tuple for the I+PI scheduler, deadlines for an EDF scheduler or priorities for a priority based scheduler. This in Miosix takes the form of a generic class

that each scheduler has to redefine and that, for backwards-compatibility reasons is still called “Priority”.

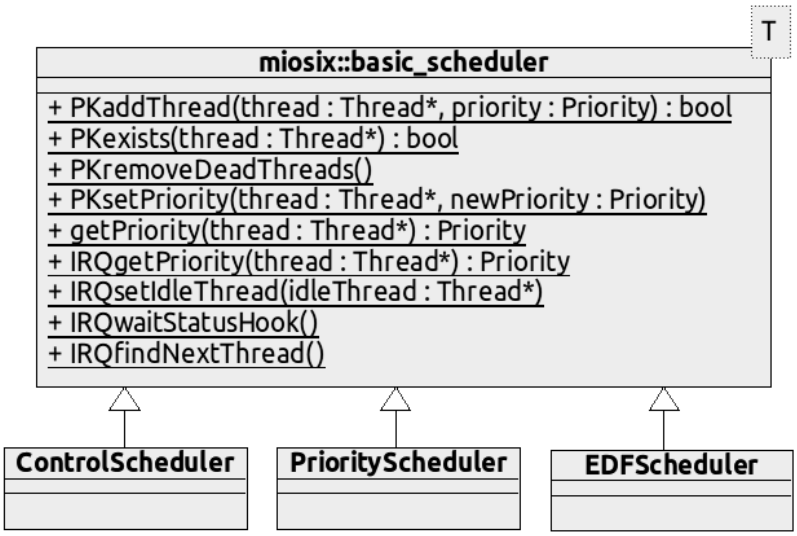


Figure 3.2: The pluggable scheduler API in Miosix

There is then a function that low level interrupt handling code calls to ask the scheduler to perform a context switch. It is in general called from within a timer interrupt service routine. This function selects the next thread that will run, the main task a scheduler is designed to solve.

Lastly, there are a number of “notification points” or “hooks”, functions that the kernel calls to inform the scheduler of certain events it may be interested into, like a thread blocking or unblocking.

3.1.3 Implemented schedulers

The Miosix codebase currently has three schedulers that implement the pluggable scheduler API. The first is the I+PI one, complete with feedforward and regulator reinitialization. Then there is the EDF scheduler, optimized to handle periodic tasks as this has been its major use during the benchmarks. Lastly, there is a priority based scheduler that always schedules the thread with the highest priority, and if there are more threads with the same priority

it fallbacks to a Round Robin policy. This is the scheduler used as a Round Robin scheduler in the benchmarks.

3.2 I+PI implementation in Miosix

The I+PI scheduler has been fully implemented in the Miosix kernel. This section is dedicated to complement the scheduler description found in the theory chapter with some relevant implementation details.

3.2.1 Sensors and actuators

Focusing for a moment at the I+PI part of the scheduler without considering the set point generation logic (feedforward, reinit) it is evident that, being a feedback regulator, it requires *measures* from the task pool it is scheduling to compare them with its set points, and also requires *actuators* to control the task pool.

The quality of these sensor and actuators affects the scheduler's achievable performance, so it is important to understand their requirements, how they differ from the requirements of a traditional scheduler and how they can be met in a real implementation such as Miosix's one.

First, consider a simple Round Robin scheduler. Such a scheduler assigns the CPU to tasks always for a fixed amount of time (unless the scheduled task blocks). Therefore its actuator is a generic API call that gives the CPU to the task selected by the RR algorithm for a fixed and constant amount of time. RR, by not being a feedback scheduler, does not make use of sensors.

The EDF scheduler, instead, requires an actuator that assigns the CPU to a task with no timeout. As EDF always schedules the task with the earliest deadline, only the arrival of a task with a closer deadline would cause a context switch, not the expiration of a timeout as for RR. Typically, this scheduler does not make use of sensors as deadlines are usually passed to the scheduler by the the task themselves rather than by direct sensing.

I+PI on the other hand, computes a desired burst value for each thread, and this value changes over time. Therefore the required actuator is similar to the one required for a RR scheduler, in that it needs to give the CPU to a task selected by the scheduler and preempt it after a timeout value, but it has the notable difference that the timeout is no longer a fixed, constant value, but is dictated by the scheduler to be equal to the desired burst. Such burst can also be zero meaning the task would not run for a given round. However, the existence of zero bursts does not add complications to the actuator as tasks with a zero burst are simply skipped. Also, the I+PI ability to give an exact CPU share to tasks even in the presence of unpredicted run-time disturbances is grounded on the availability of a measure of the *actual* burst values, which are the actual time a task has run.

3.2.2 Context switch implementation

The idea of varying the burst values is not new in the scheduling world. Such a modification is often done to basic schedulers like RR [15], as well as more complex ones such as the multilevel feedback queue one to achieve various goals including increasing responsiveness, reducing cache misses and so forth. Also the idea of using sensors to measure the effective execution time of a task is widely used like for example in the Linux kernel's completely fair scheduler which always schedules the task with lowest accumulated execution time with the aim of improving fairness [2].

However, in most cases the assignable burst value is a multiple of a fixed time quantum. This is because the actuator is still implemented as a periodic interval timer that generates interrupts to preempt tasks, and the variable burst is implemented by counting a given number of interrupts before doing a context switch.

Despite the advantage of simplicity, this approach has the disadvantage that both task execution time sensing, and burst assignments are relatively coarse grained, with a resolution that can hardly go below the order of milliseconds, as well as a performance implication of generating more interrupts

than what is strictly required to perform context switching.

In a control based scheduler the coarse grained sensing and actuation of task bursts is seen as a quantization, which adversely affects the scheduler's ability to maintain its set points. Worse, this quantization is of the same order of magnitude of the computed burst values.

For quantization to be of negligible impact, the resolution of sensors and actuators must be suitably lower than the typical assigned burst values, therefore if burst values are in the order of milliseconds, sensors and actuators need to have a resolution at least in the order of microseconds. This resolution cannot obviously be reached by "polling" the running task with a periodic timer, as the overhead caused by the time spent in a so frequently called interrupt service routine would be unacceptable.

Therefore, a completely different approach to context switches is used in the Miosix implementation of the I+PI scheduler. An hardware timer is always used, but it is configured at the start of each burst to generate exactly one interrupt at the end of the burst.

To understand the details of the implementation it is better to explain how a hardware timer works. Timers are usually composed of three parts, a prescaler whose aim is to divide the system frequency before feeding it into the actual counter, a counter which is a readable and writable peripheral register that is incremented (or decremented) in hardware, and a configurable interrupt generation logic.

In the Miosix implementation of I+PI the prescaler is configured to output a 100KHz frequency; in this way the counter has a resolution of 10us. At the start of each burst the counter is reset to zero and the interrupt logic is configured to generate an interrupt when the counter reaches the desired burst value. When the scheduler is called again, either because the burst expired or because of the task has blocked, it first reads the timer's counter therefore having a measure of the task's execution time. Of course, in the absence of blocking the read value would equal the desired burst value, while in case of blockings (calls to yield(), sleep() etc.) the value would be lower.

The read value could also be greater than the desired burst value, for example in case the thread has been in some critical section with interrupts disabled.

It is noteworthy to outline the benefits of such approach. First, a single hardware timer is used both as sensor (by reading the counter at the end of a task's *actual* burst), and actuator (by generating an interrupt at the end of a task's *nominal* burst) and second, it allows to have the fine grained resolution the I+PI scheduler requires. Even a 16bit timer as the one used in the Miosix implementation allows to impose/measure bursts of up to 655.35ms with a 10us resolution. Third, it allows to implement context switching more efficiently, interrupting the task only when it has to be preempted.

Even mainstream kernels such as the Linux kernel have for a long time used the periodic timer approach to context switches, and only recently a similar implementation based on high resolution timers has been developed [1], even though the Miosix implementation predates the Linux one.

3.3 Effort on standard compliance

Unlike many (RT)OSes for microcontrollers, Miosix aims at providing a working implementation of the C and C++ standard libraries. Assuming it does not exceed the microcontroller's available memory, an application that depends on those libraries only should compile and run with no modifications.

The reason for this goal is not only to favor porting of existing code to microcontrollers and back, but also to provide a "familiar environment" for new developers that can write applications relying on their knowledge of such libraries. It was felt that in the embedded systems world the need for "reinventing the wheel" is still widespread, and the decision to provide good support for the standard libraries is an attempt at solving this problem.

The current lack of support for all but the most basic facilities of the C standard library in many OSes for microcontrollers is probably explained by the difficulty of achieving such goal. In fact, as will be explained later on, it is even necessary to patch the standard library implementation to make

it fully work in a multithreaded microcontroller environment. Therefore, if not done by the OS developers, it could not be expected that application developers, maybe facing issues such as "time to market" requirements will spend such a development effort in integrating the chosen OS and the standard libraries. As such, the author believes that or this task is performed by the OS developers, or the goal will not be achieved at all.

3.3.1 Focus on C++, not just C

The Miosix kernel has a special attention to providing C++ support on microcontrollers. Also, most of the kernel as well as the Miosix specific API is written in C++. The reason behind that is twofold. First, it is believed by the author that using a higher level language such as C++ simplifies writing and maintaining code, and second because current 32bit microcontrollers have enough resources to actually allow developing applications in C++.

Of course, it is always possible to write applications for Miosix in C, therefore having support *also* for C++ is definitively a value added.

It is also interesting to note that certain development tasks are widely believed to be better suited for an object oriented language. One such example is the development of GUI applications where the most well known example of GUI toolkit in C, GTK+, makes use of an object system developed atop of C. Therefore, the availability of an object oriented programming language simplifies such tasks without having to design an object system in C, which looks suspiciously like reinventing the wheel.

Full support for C++ as well as its standard library is a rare feature available in an OS for microcontrollers, in fact, as the OS survey at the end of this chapter will show, many OS provide no support at all, some of them provide minimal support often barely tested and lacking features such as exception handling, and very few provide full support.

3.3.2 POSIX thread API

Given its effort on standard compliance, it is not surprising to discover that Miosix supports the POSIX thread API, being the most portable API of its kind.

The advantages over OSes for microcontrollers that provide a custom threading API only are as previously mentioned increased code portability as well as reduced the learning curve by providing an API many developers are already familiar with.

Actually, the full pthread API is composed of many facilities, of which only some of them are currently implemented in Miosix. The implemented features are however the most used ones, and are here summarized:

- `pthread_create()`, `pthread_join()` and other functions to manage threads. Of notable interest is that `pthread_attr_setstacksize()` is implemented, as it is very useful to tweak the stack size of spawned threads when the available memory is scarce as it happens on microcontrollers.
- `pthread_mutex_lock()`, `pthread_mutex_unlock()` and other functions related to mutexes.
- `pthread_cond_wait()`, `pthread_cond_signal()` and in general the condition variable functions.
- `pthread_once()` is also provided.

A notable missing feature is the implementation of `pthread_key_*` thread local storage, which is currently not supported owing to the difficulty of providing a run time and space efficient implementation suitable to microcontrollers.

It is also interesting to delve for a moment in the implementation details of how POSIX threads are implemented in Miosix, and discover that an efficient implementation has required patching the C standard library.

For now, consider the pthread mutex API, even if what will be explained applies equally also to the condition variable API. All the mutex related functions take as argument an opaque type named `pthread_mutex_t` which is the data structure containing the mutex's state. It is important to know how such a data structure can be implemented. Mainly, there are two ways: the first is to have `pthread_mutex_t` being an opaque pointer to a struct, and the second is have `pthread_mutex_t` as a typedef for a struct.

Despite the first solution would provide better encapsulation (C, unlike C++ lacks access control to struct fields), it also have significant performance issues. First, if `pthread_mutex_t` is a pointer to a struct, this means that the *actual* struct has to be dynamically allocated on the heap, therefore reducing the performance of the `pthread_mutex_init()` and `pthread_mutex_destroy()` functions. But these functions are not expected to be called very often, so this is not the real problem. The problem is that such an implementation also slows down the `pthread_mutex_lock()` function, which is likely one of the most crucial ones (together with `pthread_mutex_unlock()`) when considering mutex performance. The reason is not so obvious, and derives from the fact that a mutex can be initialized not only with `pthread_mutex_init()`, but also with the syntax in Listing 3.1.

Listing 3.1: POSIX mutex initialization

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
```

Since we are exploring the case when `pthread_mutex_t` is a pointer to struct, for this syntax to be valid `PTHREAD_MUTEX_INITIALIZER` must be a macro that redefines it as `NULL` or any other special value. Therefore, a null pointer may be passed to `pthread_mutex_lock()`, and this function must check whether the pointer is null and allocate the struct if it is. Worse, `pthread_mutex_lock()` does not only need to check if the pointer is null, it must do so *atomically* otherwise if two threads are trying to lock a null mutex at the same time they would both allocate a mutex struct and succeed in locking the mutex, defeating the very purpose of mutexes, other than leaking

memory.

The additional run time cost of atomically checking the mutex for null is an additional run time cost every time the `pthread_mutex_lock()` is called, which explains the performance issues associated with this option.

The other option instead, having `pthread_mutex_t` as a typedef for the mutex state struct does not require dynamic memory allocation of structs nor atomic check for null every time a mutex is locked and is therefore much more performant. However, the definition of `pthread_mutex_t` is in `pthread.h` which is part of the C standard library.

Miosix uses the newlib C library, because it is small and tailored to the needs of embedded systems. Its implementation of `pthread_mutex_t` is simple: for "well known" operating systems, such as Linux, the OS specific implementation of `pthread_mutex_t` is used, while for bare metal/RTOS systems, not knowing how the mutex state structure has to be implemented for those systems, it fallbacks to the pointer implementation.

It is now clear the reason why to have an efficient implementation of POSIX threads it has been necessary to patch the C standard library by injecting the Miosix specific mutex struct in the system headers.

3.4 Thread safety of the C and C++ standard libraries

Providing support for the C and C++ standard libraries in an OS for microcontrollers does not only mean redirecting printf to a serial port, or integrating *stdio* file function with a filesystem driver, it also means making sure that the standard libraries, as well as all the programming language constructs are *thread safe*. For strange that it may seem at first, even the implementation of the standard libraries, just like any program, requires locking or other countermeasures to be used concurrently by multiple threads. Even more, there are some programming language constructs that admit no implicitly reentrant implementation and require explicit locking to be made

thread safe.

Such an issue is often not considered when designing OSes for microcontrollers, partly because it is not a well known issue, partly because many OSes are not even designed to work with standard libraries, but most importantly due to the difficulty of achieving this goal. Reason for this difficulty is mainly because cooperation between the OS and the compiler/standard library implementation is required, while they are usually independently designed.

From now on, these considerations will be mainly targeted to the GCC compiler, as the thread safety oriented code review and fix have been performed on this compiler only, but the problem is compiler independent. The way GCC, newlib (C library) and libstdc++ (C++ library) handle this issue is the following: for well known OSes (Linux, Windows, etc.) locking and other countermeasures are implemented to make the standard libraries and language constructs safe in a multithreaded environment, and this is done both by calling proper OS APIs, for example mutex APIs, and by making use of lock free algorithms using suitable assembler instructions such as the "exchange and add" and "compare and swap" ones, if the architecture supports them. while on the other hand, for "bare metal" embedded systems, partly based on the (wrong) assumption of the lack of threads, and partly due to the lack of knowledge of the OSes API, it is assumed that threads do not exist and therefore all locking operations become "no-ops".

Now, as already mentioned, certain thread related issues can be solved by the compiler alone by using lock free algorithms making use of special assembler instructions. But this is not enough both because such instructions are often not present in the instruction set of microcontrollers, and also because not all thread related issues found in standard libraries admit a lock free implementation. Therefore both in the case of "emulating" lock free algorithms in architectures that lack the necessary opcodes, and for locked algorithms the only solution is to call the OS, and GCC can't do it because it doesn't know the OS API.

The solution adopted in Miosix is the following: the compiler and standard libraries have been patched so as to call the threading API that Miosix provides whenever required. While this has the drawback that compiling the Miosix kernel is only supported through a patched compiler, it is the only way to achieve thread safety.

A list of specific points within standard libraries, as well as programming language constructs that require attention in threaded program is here reported, along with the implemented fixes.

3.4.1 FILE * locking

The *stdio* API offers a portable abstraction to the console and filesystem allowing to perform read ad write operations. Such operations are buffered, and concurrent access to those buffers is what causes multithreading issues.

The most common way this can occur is when more threads are concurrently writing to the console through `printf()` or other similar functions. For this reason, POSIX [27] states that a conformant implementation of all `FILE *`, including *stdin* and *stdout* should include a lock that is acquired by the functions that read and write from it, before modifying the buffers.

Newlib implements this by making calls from the *stdio* library to a set of functions declared in `sys/lock.h` that, as anticipated, are unimplemented in bare metal targets. Therefore, one of the patches is dedicated to implement these locks in terms of the pthread mutex API.

3.4.2 Static constructors

The `static` modifier can be used in C to declare variables within functions whose state is preserved across function calls. This syntax has been extended in C++ to have class instances declarable static in the same context. The main difference is that classes need a constructor call to be initialized before they can be used. For static classes, the constructor is required by the C++ standard to be called the first time the function that contain the static class

instance is called.

Therefore, this creates an issue if two threads call a function with static class instances in them. To avoid calling the constructor more than one time, locking is required. Being a programing language feature and not a standard library call, the locking code is inserted by the compiler. GCC secretly inserts calls to functions named `__cxa_guard_acqire()`, `__cxa_guard_release()` and `__cxa_guard_abort()`, and they are implemented as stubs that do no locking at all for bare metal systems. Miosix includes patches to correctly implement these functions.

3.4.3 C++ exception handling

The internal machinery required to perform exception handling relies on a per-thread data structure to be reentrant.

Failing to keep this data structure separate on a per-thread basis could result in application crashes in the event that two exceptions are thrown simultaneously by two threads. Issues of this kind are usually also difficult to debug.

Miosix includes patches for this, as well as a specific test in the testsuite where eight threads continuously throw exceptions in order to verify the correct behaviour.

3.4.4 Reference counting within libstdc++

The C++ standard library makes use of reference counting in some places, of which the most notable is the `shared_ptr` class. Other than that, reference counting is also used internally, like in the `string` class, and also within the `iostream` library.

Of course, reference counting does not work as-is in multithreaded systems, locking is required. This is one case that could be dealt with by using atomic operations such as the "exchange and add". However, both the Cortex M3 and ARM7 CPUs that Miosix supports lack such opcodes.

The solution adopted is a patch that implements those atomic operations by briefly disabling interrupts.

3.5 Additional kernel features

Other than providing a multithreaded environment with support for the C and C++ standard libraries, like any other kernel, Miosix aims at providing a software interface to access the hardware.

Such software interfaces include

- `mxgui` is a library to control various monochrome or color display devices. It provides an abstraction over a display with some common drawing primitives like font rendering with antialiasing, line drawing etc.

- `mxusb` is an USB device library that tries to automatically configure an USB peripheral starting from USB descriptors. It is integrated with the kernel in the sense that it is possible to have threads waiting on USB endpoints, without having to deal with interrupt service routines directly.

- A device-independent way to access GPIOs. It uses C++ template metaprogramming and is highly optimized with the aim of minimizing the number of generated assembler instructions, without having any part directly written in assembler. Atop of this, classes to emulate in software some common serial protocols such as SPI and I2C have also been developed.

- The facilities of the C and C++ standard library to access files are implemented in terms of a filesystem driver currently supporting the FAT32 filesystem, and various backends to read and write from SD cards.

- The console related facilities of the C and C++ standard library (e.g. `printf()`) can be redirected to a serial port or other suitable channel to be usable especially for debugging during code development.

3.6 Comparison with other OSes in the same class

A survey has been done to compare various OSes for microcontrollers, to compare the key features of Miosix with other kernels, and the result is summarized in Table 3.1.

From the gathered data it is clear that, other than the obvious fact that only Miosix implements the I+PI scheduler, only two other kernels: eCos and RTEMS have similar features to Miosix, despite they have been under development for much longer, and they have not been entirely developed by a single person.

As anticipated, pluggable schedulers is a feature that most microcontroller kernels lack, and the ones that implement it usually impose restrictions on the data structure that the scheduler has to use, or on the way context switches are performed.

When it comes to C++ support, standard libraries and POSIX threads it can be concluded that many kernels still lack support for those features, being unavailable in more than half of the considered kernels.

	Atomthreads	BeRTOS	ChibiOS/RT	Contiki	eCos	Embox	ERIKA	FreeRTOS	FunkOS	Milos	Miosix	Nano-RK	NuttX RTOS	Prex	RTEMS	scmRTOS	SDPOS	uOS	Y@SOS
I+PI Scheduler	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-	-	-	-	-
Pluggable schedulers	-	-	-	-	P	P	P	-	-	-	Y	-	M	-	P	-	-	-	-
C++ support	-	-	M	-	Y	-	-	-	P	-	Y	-	M	-	Y	P	-	M	-
Standard libraries	-	-	M	-	Y	M	-	-	-	-	Y	-	C	C	Y	-	-	C	-
pthreads	-	-	-	-	Y	-	-	-	-	-	Y	-	Y	-	Y	-	-	Y	-
Threadsafe stdlib	-	-	-	-	Y	-	-	-	-	-	Y	-	C	C	Y	-	-	C	-

Y = Full support
P = Partial support
M = Minimal support
C = Applies only to the C language, not C++
- = No support at all

Table 3.1: Microcontroller kernel comparison

Chapter 4

Benchmarks

After introducing the control theoretical viewpoint to the scheduling problem and having described the I+PI scheduler, it is now time to assess said scheduler's performance. To do so, I+PI is here compared with two other well known scheduling algorithms, EDF and Round Robin. Tests have been performed using the Miosix implementations of those schedulers.

Other than testing the scheduler's *performance*, the issue of testing the *correctness* of the scheduler implementations within the Miosix kernel has also been addressed.

The results here presented are from four benchmarks. The first is the MiBench benchmark that has been used to test the Miosix platform in general. Then a MiBench extension has been devised to verify the scheduler implementations. The third benchmark is instead taken from the Hartstone benchmark suite and has the aim of comparing the performance of the schedulers, while the fourth one is an extension of the third that compares schedulers under the specific condition of an unschedulable task pool, evidencing the greater flexibility of the I+PI scheduler.

All the tests here presented have been run on a `stm3210e-eval` board, equipped with a 72 MHz ARM Cortex M3 microcontroller and a 1MB external RAM, from which the code executes. Source code for the benchmarks is available on the Miosix website [3].

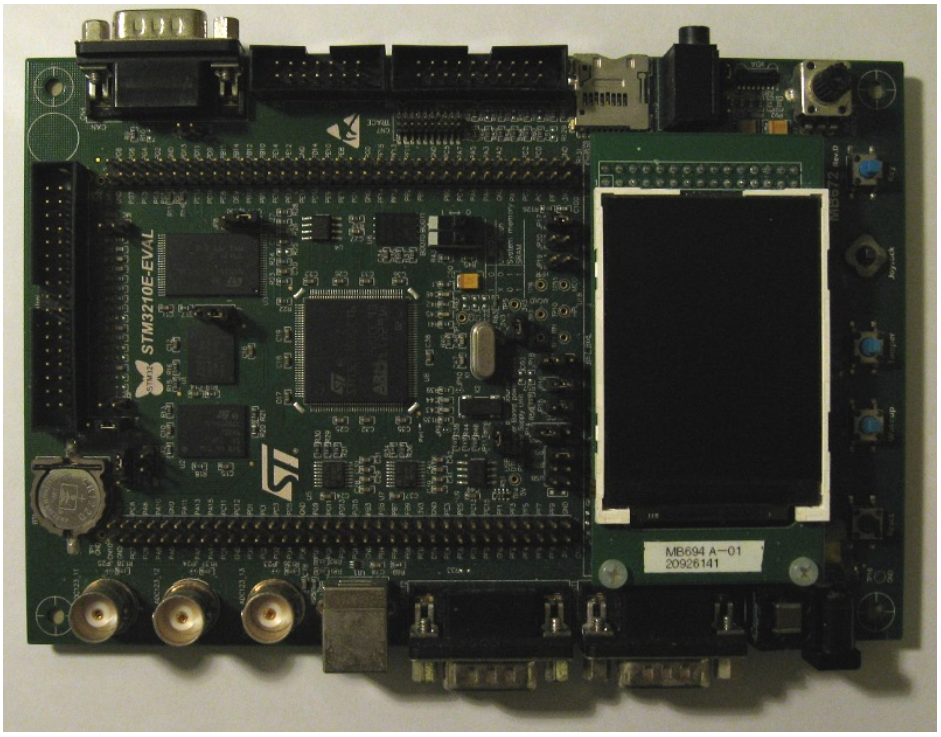


Figure 4.1: The stm3210e-eval board, used to perform the benchmarks.

4.1 Correctness check: MiBench

The MiBench [14] benchmark suite is a set "real world", commercially representative applications, distributed along with reference input and output datasets. Applications are all written in the C programming language and span from mathematical computation, to image encoding, network routing, cryptography and audio encoding/decoding. There are two datasets per benchmark, a "small" and a "large" one, and due to the limited available memory in the chosen development board only the small input ones have been used.

Despite being originally designed to test the CPU, cache and memory performance of an embedded system, MiBench is here repurposed to test the "correctness" of the Miosix platform. In fact, while being proposed as an embedded benchmark, many of its applications are designed for full fledged

operating systems such as Linux rather than for microcontrollers, and as such make an extensive use of the C standard library functions. This allows to achieve a broad coverage of such library as well as the filesystem implementation as most applications operate on files.

The MiBench benchmark suite is also a proof of the advantages of having a kernel with native support for the C and C++ standard libraries. Attempting to port MiBench's applications to an embedded kernel without support for the C standard library would have required either to modify the applications so as not to depend on the C library or to integrate a C library with the kernel, both of which would have required a massive amount of work.

With Miosix instead, most applications only required small tweaks and some even compiled just fine with no modifications at all.

Table 4.1 reports execution time and memory occupation results from some of the MiBench applications. Nearly all of the benchmarks, when run on Miosix, produce an output identical to the provided reference. Some of the output contain insignificant differences that are here reported for completeness

- The `basicmath` program prints many floating point numbers, some of them differ from the reference ones in the last decimal digit, but this has been tracked down to difference in rounding between the newlib's implementation of `printf()` (used in Miosix) and GLIBC (used on Linux).

- The `tiff` benchmarks produce images that compare equal pixel by pixel, but the image files do not compare equal byte by byte. Investigation found that the tiff file contains the file name and its path in one of the metadata fields, and the file path is what causes the files to not compare equal.

- The `blowfish` program segfaults when run on Linux, and crashes also when run on Miosix. To some extent it can be said that the behaviour is consistent.

Category[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.

Table 4.1: Summary of relevant MiBench [14] execution data.

- The `rijndael` program does not compile with recent versions of GCC, both on Linux and on Miosix. This is because it assumes that `fpos_t` is a typedef for an integer type, which probably was true in older versions

of the C library but is not mandated by the standard. Fixing it resulted in the same output both on Linux and Miosix, but different from the reference output.

4.2 Scheduler correctness: Parallel MiBench

The MiBench benchmark introduced before implies the execution of one application at a time, and all the applications are single threaded. Therefore, it does not adequately stress the scheduler.

To address this shortcoming, a simple extension of the MiBench benchmark is here presented. This extension consists in running one application per category in a thread of its own. The chosen applications are basicmath, jpeg, stringsearch, dijkstra, sha and gsm; they have been chosen so as not to exceed the 1MB available memory. The benchmark begins by spawning all the threads simultaneously and ends when the last application has terminated. The I+PI and RR scheduler were tested, while EDF was not as it is incapable of scheduling tasks that lack a deadline and would have required setting fictitious deadlines. With both schedulers, the parallel execution results do not differ from the sequential ones.

Table 4.2 reports the execution times of the individual benchmarks when run sequentially, whose sum is T_{seq}, and the execution time when running the applications simultaneously (T_{sim}).

This data already gives us a hint on the performance of the two schedulers. First, since the parallel execution time is lower than the sequential one in both cases, both schedulers are capable of re-assigning CPU time when one or more thread blocks, in this case due to I/O operations.

The sequential total execution time is lower using the RR scheduler. This is not surprising given that RR is one of the simplest and computationally lightweight schedulers, and having only one thread to schedule is a corner case that doesn't allow I+PI to express its potential. It can even be discussed whether a scheduler is required *at all*, in such a case.

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 4.2: Summary of simultaneous MiBench [14] execution times [s].

On the other hand, the parallel execution time is lower with I+PI, showing the superiority of the new scheduler.

It can be concluded that, despite being slightly computationally heavier than RR, I+PI is capable of providing a speedup when scheduling real world applications thanks to better CPU management.

4.3 Hartstone

It should be noted that comparing scheduler performance is a difficult matter. Many of the scheduler design criteria, such as fairness and responsiveness are often expressed qualitatively and have an unclear quantitative definition, which is required to set up a benchmark. Other attempts at comparing schedulers are based on algorithmic complexity which conflicts with the aims of this thesis as it focuses on the data structures and algorithms of the scheduler, rather than reasoning at a higher abstraction level. Also, as the previous benchmark shows, saying that a scheduler has a lower computational complexity with respect to another one does not imply that it is able to schedule a given task pool better. Only in specific areas, such as real time systems, there are quantitative measures that allow to characterize a scheduler, like the ability to meet deadlines.

The Hartstone benchmark [28] is here used to compare the performance of I+PI, EDF and RR. Hartstone is a benchmark suite with many tests all designed in the same way. The tests start with a baseline workload which is increased till an assertion fails, usually the miss of a deadline. The amount

of workload a system can withstand is used as a measure of its performance.

From the various tests Hartstone is composed, the PH series is here used, which stresses the system using periodic tasks having harmonic frequencies. Hartstone measures the workload assigned to tasks in KiloWhet per second (KWIPS), where a Kilo Whetstone is another benchmark related to floating point operations. Given that the purpose of the benchmark is in this case to compare the *relative* performance of the scheduler, rather than assessing the performance of a CPU architecture and compiler (which was the original goal of the Hartstone benchmark), the KiloWhet code has simply been implemented as a busy wait delay that keeps the CPU occupied for 1.25ms.

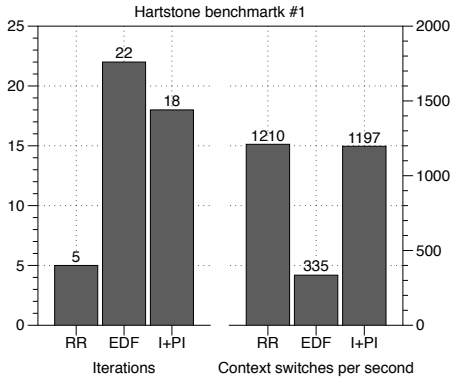
The baseline system of the PH test series is composed of five periodic tasks with a specified workload reported in Table 4.3.

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

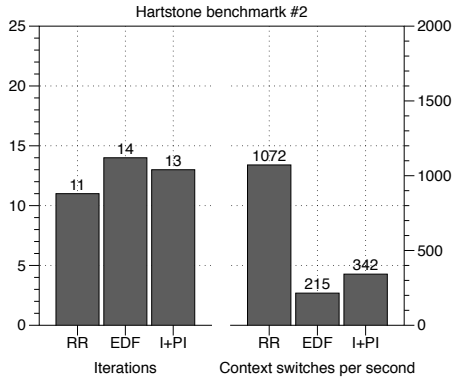
Table 4.3: The Hartstone [28] baseline task set.

The task period coincides with the deadline meaning that a task should complete a period's workload before the next one begins. Tasks that complete their workload before the end of the period sleep till the beginning of the next period.

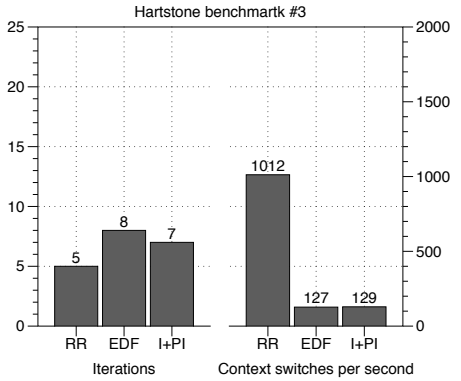
The PH test series is composed of four tests that start from the same baseline load but increase the workload in different ways. In the first benchmark the frequency of the fifth task is increased by 8Hz at each iteration till a deadline miss occurs. This test allows to measure the ability of a scheduler to switch rapidly between tasks. The second benchmark scales the frequencies of all tasks by 1.1, 1.2, ... till the first miss, and has the property of maintaining a balanced workload between the tasks. The third benchmark



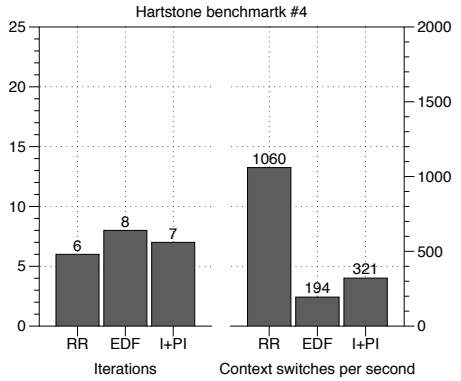
(a) Hartstone test 1



(b) Hartstone test 2



(c) Hartstone test 3



(d) Hartstone test 4

Figure 4.2: Results for the Hartstone benchmark [28].

increases the workload of all tasks by 1 KiloWhet at a time resulting in an unbalanced workload increase. Lastly, the fourth benchmark increases the workload by adding tasks with a workload and period equal to the third task of the baseline, measuring the ability of the scheduler to handle a large number of tasks.

The test results are reported in Figure 4.2, that shows the number of iterations till the first deadline miss for the three schedulers, as well as the number of context switches per second. Obviously, a good scheduler should score high results for the number of iterations, while minimizing the number of context switches.

The first fact that can be noticed is that RR, despite the chosen time

quantum of 1ms in an attempt to increase its performance, scores the lowest results in all four benchmarks. This is an example of how the idea of optimizing the scheduler in terms of algorithmic complexity to minimize the context switch cost, and than making many context switches so as to improve system responsiveness does not produce the desired results. Despite making more context switches than the other schedulers, its performance is the lowest.

EDF on the other hand is optimal, which is not surprising as it has been formally proven optimal under the hypothesis of a schedulable task pool.

What *is* surprising though, is that the performance of I+PI approaches the one of EDF in all but the first benchmark. And it does that without the scheduler having any knowledge of deadlines whatsoever, unlike EDF. This is a proof of the validity of the control-based methodology used to design the I+PI scheduler, that can *implicitly* meet deadlines by ensuring tasks have enough CPU time not to miss.

The reason why I+PI performs lower than EDF in the first benchmark (while still performing much better than RR) is that it is the most extreme case, where the period of one tasks differs so much with respect to the other. In an embedded system it would even possible (and preferred) to handle such a low period task using a dedicated timer interrupt rather than a thread.

4.4 Extended Hartstone

To show the ability of a scheduler designed using a feedback approach to handle various different workloads, the Hartstone benchmark was extended to address a rarely considered situation: "what happens if the required CPU transiently exceeds unity and the task pool becomes unschedulable?"

This can happen in soft real-time systems where, for cost reasons, the CPU is not dimensioned to handle the *maximum* predicted workload, but rather it is dimensioned somewhere in between the average and maximum workload.

This extended Hartstone benchmark consists of running the system for a

total time of 120s. In the first 30 seconds the workload is kept constant at 48% CPU utilization. At t=30s the workload is increased to 120% and kept constant at that value till t=45s when it is decreased again to 48%, and kept at that value till the end of the 120s.

This gives a period of schedulable workload to stabilize the system, followed by a transient unschedulable period, and a long period in which the workload is again feasible to allow the schedulers to recover normal operation.

The way of increasing the workload is what differentiates the four benchmarks, and is the same as the one used in the four benchmarks of the classic Hartstone.

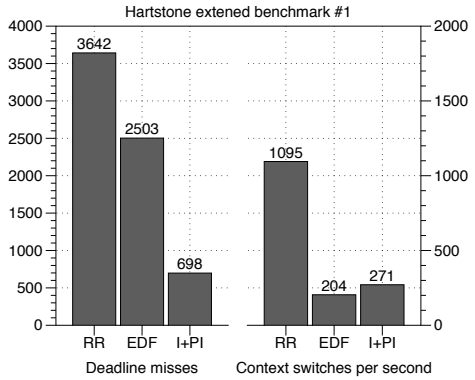
Performance is measured in number of deadline misses (lower is better), and is reported in Figure 4.3, along with the number of context switches per second to appreciate the ability of the schedulers to avoid unnecessary context switches.

RR, as in the classic Hartstone, despite making significantly more context switches than the other two schedulers, performs worse in all four benchmarks. On the other hand, I+PI outperforms EDF in all four tests, at the price of a minimal increase in the number of context switches.

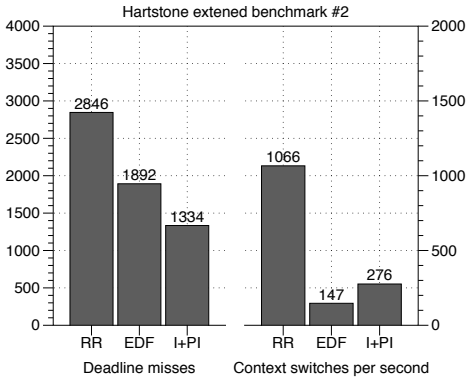
Additionally, it is noteworthy to remark that I+PI thanks to the relative importance parameter, allows to *predict* the CPU share that will be given to each task *even in the case of CPU overutilization* allowing hard real-time tasks to coexist with soft real-time ones. This is in contrast with EDF where no guarantee can be given regarding *which* tasks will miss deadlines in case of overload, and to priority based schedulers, where no guarantee at all can be given on the received CPU share of tasks, except maybe the ones with the highest priority.

Adding the ability of I+PI to schedule batch tasks that have no deadline at all (unlike EDF) it is perhaps now clear the "execution pattern agnostic" term used to describe I+PI in the theory chapter.

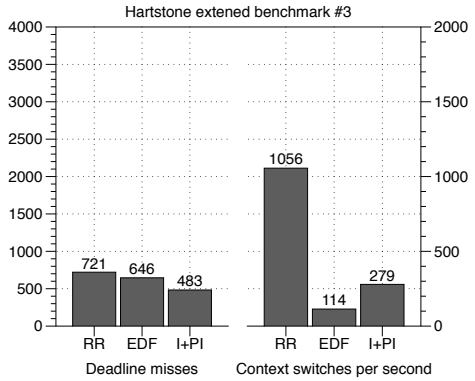
As a final remark, the I+PI scheduler is the only one of the three that makes use of floating point calculations, which is relevant from a performance



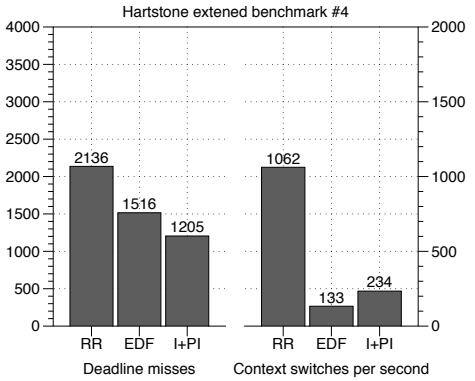
(a) Extended Hartstone test 1



(b) Extended Hartstone test 2



(c) Extended Hartstone test 3



(d) Extended Hartstone test 4

Figure 4.3: Results for the extended Hartstone benchmark.

point of view as the architecture used to perform the benchmarks lack an hardware floating point implementation. A fixed point implementation of I+PI, or moving to a processor with hardware floating point will increase the performance of I+PI even further.

Chapter 5

Conclusions

This thesis has presented a fully exploited example of how computing systems can be improved with the introduction of control theoretical principles in their *design* phase. The focus was here set on a specific problem, namely task scheduling in an operating system, for which a solution was devised relying entirely on said methodology.

The proposed scheduler has not only been simulated, but also implemented in software within an operating system and run on real hardware; benchmarks were used to compare its performance with other well known schedulers, and said benchmarks were also extended to show facts that do not emerge unless the test procedures themselves are conceived as they need to be for the assessment of a control system.

The obtained results proved the efficacy of this innovative approach and its practical realizability.

5.1 Future directions

In the future, effort will be spent both to further extend the designed I+PI scheduler, as well as to apply the control theoretical approach to other computing system problems.

The current version of the I+PI scheduler is an efficient solution to sched-

ule tasks on an *uniprocessor* system. While for embedded systems this constraint is not yet an issue, attempts at implementing I+PI in desktop or server operating systems, such as the Linux kernel, would need to generalize the approach to cope with multicore systems. This is probably the next step that will be taken.

Regarding the application of the control theoretical approach to other problems, bandwidth partitioning and Quality of Service (QoS) issues are a promising application that could benefit from such approaches. For example, USB bandwidth scheduling is a problem that could, at least in theory, be tackled with a similar solution as the one presented in this thesis.

Bibliography

[1] Completely fair scheduler and its tuning. http://www.scribd.com/doc/42318144/Cfs-Tuning-2.

[2] Inside the linux 2.6 completely fair scheduler. http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/.

[3] Miosix kernel. http://www.webalice.it/fede.tft/miosix/index.html.

[4] Luca Abeni, Luigi Palopoli, Giuseppe Lipari, and Jonathan Walpole. Analysis of a reservation-based feedback scheduler. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 71–80, 2002.

[5] Biswas K. Alam B., Doja M.N. Finding time quantum of round robin cpu scheduling algorithm using fuzzy logic. pages 795–798, 2008.

[6] Devi U.C. Anderson J.H., Calandrino J.M. Real-time scheduling on multicore platforms. In *12th IEEE Real-Time Embedded Technology and Applications Symposium*, pages 179–190, 2006.

[7] Karl-Erik Årzén, Anders Robertsson, Dan Henriksson, Mikael Johansson, Håkan Hjalmarsson, and Karl Henrik Johansson. Conclusions of the artist2 roadmap on control of computing systems. *SIGBED Review*, 3:11–20, July 2006.

[8] Ken W. Batcher and Robert A. Walker. 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, pages 260–263, New York, NY, USA, 2008. ACM.

[9] Peter Brucker. *Scheduling algorithms*. Springer, 2007.

[10] Anton Cervin and Peter Alriksson. Optimal on-line scheduling of multiple control tasks: A case study. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Self-tuning schedulers for legacy real-time applications. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 55–68, New York, NY, USA, 2010. ACM.

[12] Tommaso Cucinotta, Luigi Palopoli, Luca Abeni, Dario Faggioli, and Giuseppe Lipari. On the integration of application level and resource level qos control for real-time applications. *IEEE Transactions on Industrial Informatics*, 6(4):479–491, November 2010.

[13] Terraneo Federico. *Application of feedback scheduling to a preemptive kernel via discrete-time cascade control*. POLITECNICO DI MILANO, 2008.

[14] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[15] H Harwood, A; Shen. Using fundamental electrical theory for varying time quantum uni-processor scheduling. *JOURNAL OF SYSTEMS ARCHITECTURE*, 47:181–192, 2001.

[16] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley, September 2004.

[17] Maria Kihl, Anders Robertsson, Anders Andersson, and Björn Wittenmark. Control-theoretic analysis of admission control mechanisms for web server systems. *World Wide Web*, 11(1):93–116, 2008.

[18] Martin Ansbjerg Kjaer, Maria Kihl, and Anders Robertsson. Resource allocation and disturbance rejection in web servers using slas and virtualized servers. *IEEE Transactions on Network and Service Management*, 6(4):226–239, 2009.

[19] Ketan Kotecha and Apurva Shah. Adaptive scheduling algorithm for real-time operating system. In *IEEE World Congress on Computational Intelligence, IEEE Congress on Evolutionary Computation*, pages 2109–2112, June 2008.

[20] Douglas A. Lawrence, Jianwei Guan, Shruti Mehta, and Lonnie R. Welch. Adaptive scheduling via feedback control for dynamic real-time systems. In *Proceedings of the IEEE International Conference on Performance, Computing, and Communications*, pages 373–378, 2001.

[21] Chenyang Lu, Ying Lu, Tarek F. Abdelzaher, John A. Stankovic, and Sang Hyuk Son. Feedback control architecture and design methodology for service delay guarantees in web servers. *IEEE Transaction on Parallel Distributed Systems*, 17(9):1014–1027, 2006.

[22] Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems*, 23:85–126, July 2002.

[23] Chenyang Lu, John A. Stankovic, Gang Tao, and Sang H. Son. Design and evaluation of a feedback control edf scheduling algorithm. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 56–, Washington, DC, USA, 1999. IEEE Computer Society.

[24] Martina Maggio and Alberto Leva. A new perspective proposal for preemptive feedback scheduling. *International Journal of Innovative Computing, Information and Control*, 6(4), 2010.

[25] Luigi Palopoli and Luca Abeni. Legacy real-time applications in a reservation-based system. *IEEE Transactions on Industrial Informatics*, 5(3):220–228, August 2009.

[26] Michael Pinedo. *Scheduling Theory, Algorithms, and Systems*. Springer, third edition, July 2008.

[27] POSIX requirements about C stdio threadsafety. http://pubs.opengroup.org/onlinepubs/007908799/xsh/flockfile.html.

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

[29] Feng Xia, Guosong Tian, and Youxian Sun. Feedback scheduling: an event-driven paradigm. *SIGPLAN Notice*, 42(12):7–14, December 2007.

[30] Wei Xu, Xiaoyun Zhu, Sharad Singhal, and Zhikui Wang. Predictive control for dynamic resource allocation in enterprise data centers. In *Proceedings of the 10th IEEE Network Operations and Management Symposium*, pages 115–126, April 2006.