POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Control Based Tickless Scheduling

Relatore: William Fornaciari
Correlatore: Federico Terraneo

Tesi di Laurea di:
Ahmad Golchin, matricola 832923

Anno Accademico 2016-2017

Contents

List of Figures

List of Tables

Abstract

The use of theoretical methodologies in systems design is highly endorsed in many fields of engineering due to simplicity, flexibility, and uniformity of the solutions as well as the possibility to assess the results even in unpredictable run-time situations. The latter is probably the strongest incentive for exploiting such approaches also in the design of operating system components. This thesis is a part of the aforementioned research trend, as it is devoted to enhance robustness and responsiveness of an existing task scheduler named I+PI which is designed in the light of control theory. This scheduler works by partitioning a run-time modifiable amount of time, called the round time, among all the active tasks, and correcting on-line for any observed discrepancy. The fraction of the round time allotted to each task is also configurable on a per-task basis to best suit the tasks needs. The original scheduler was shown to have a performance closely chasing the EDF scheduler for schedulable task pools, and outperforming it for pools that may become transiently unschedulable. However, a weakness was found in I+PI with pools of periodic tasks having highly asymmetric periods. Scheduling such task pools with I+PI calls for short round times, which increases the scheduling overhead. The contribution of this thesis is to improve I+PI's responsiveness-overhead trade-off by redesigning the control scheme and exploiting the concept of tickless kernels. In such kernels, the time system is designed in an aperiodic fashion that provides other subsystems with dynamic ticks rather than having a periodic interrupt from the system timer. The performance of employing a control theory based task scheduler on top of a tickless kernel is

demonstrated in this work through suitable experiments showing that it outperforms the original I+PI scheduler in all the evaluated cases, and tends to be strictly better than EDF in many situations. We will show that the new approach is a highly qualified candidate in real-time applications with or without deadlines and without imposing any particular symmetry on the arrival pattern of tasks.

Italian abstract

L'uso di metodologie teoriche nella progettazione dei sistemi è una strada seguita in molti campi dell'ingegneria per via della sua semplicità, flessibilità e uniformità delle soluzioni nonchè per la possibilità di valutare i risultati anche in situazioni imprevedibili di esercizio. Quest'ultimo è probabilmente il più forte incentivo per sfruttare tali approcci anche nella progettazione di componenti di un sistema operativo. Questa tesi è parte di questa linea di ricerca, in quanto è dedicata a migliorare la robustezza e la reattività di uno scheduler esistente denominato I + PI, progettato usando la teoria del controllo. Questo scheduler partiziona un tempo modificabile a run-time, detto tempo di round, tra i task attivi nel sistema, correggendo on-line eventuali discrepanze osservate nel loro comportamento. La frazione del tempo di round assegnata a ogni task è anch'essa configurabile, e può essere diversa per ogni task in modo da meglio adattarsi alle necessità specifiche dei task. Lo scheduler originale ha dimostrato delle prestazioni che si avvicnano allo scheduler EDF nel caso in cui il task pool sia schedulabile, e prestazioni migliori di EDF nel caso in cui il task pool diventi temporanemaente non schedulabile. Tuttavia, lo studio dello scheduler I+PI ha evidenziato un problema nel caso di pool di task periodici con periodi molto diversi tra loro. In queste condizioni è necessario scegliere tempi di round molto bassi, cosa che causa un impatto sull'overhead di scheduling. Il contributo di questa tesi è quello di migliorare le performance di I+PI in queste condizioni, riprogettando lo schema di controllo e traendo vantaggio dalle caratteristiche di un kernel tickless. In questi kernel, il sottosistema temporale è progettato in

modo aperiodico, generando interrupt solo quando richiesto invece di avere interrupt ad un periodo fisso. Le performance del nuovo scheduler e del kernel tickless sono state verificate tramite un'opportuna campagna sperimentale. I risultati mostrano come le performance del nuovo scheduler siano strettamente superiori allo scheduler I+PI originale, e migliori di EDF in alcuni casi. Mostreremo che il nuovo approccio è un ottimo candidato in applicazioni real-time con o senza deadline e senza imporre una particolare simmetria sul modello di arrivo dei task.

Chapter 1

Introduction

Apart from design choices imposed by hardware architectures, operating system components have been heavily depending on heuristic designs and algorithms and thus, maintenance and evolution of system software have become increasingly difficult with the ongoing advances in hardware. For instance, in [14] it is evident that the Linux CFS (Completely Fair Scheduler)[6], albeit being operational, has a serious inefficient processor utilization problem that have been just revealed after a long time. Or, there are some other efforts such as [17] and [24] to redesign kernels from scratch to become inline with and better utilize the newest hardware capabilities. As far as this work is concerned, some of the problems are still dealt with to a significant extent through the use of heuristics that are not rooted in formal and strong mathematical models that allow better uniformity in design and assessment methods. Various parts of kernel such as task scheduler, load balancer, USB bandwidth scheduler and etc., can be viewed as closed-loop time-varying systems to be controlled, hence, a right tool for modeling these problems would be Control Theory.

An interesting research work in this direction is the I+PI scheduler[16] which proposes a model for single-core scheduling problem ensuring responsiveness and fairness through two control loops in which an outer loop makes sure that all the active tasks will receive processor's control within a user-

defined window of time - i.e. round duration - and an inner loop that ensures each task consumes a prescribed portion of the round duration, hence, guaranteeing fairness. For instance, if the round duration is 5 milliseconds, all active tasks will be processed at least once every 5 milliseconds. I+PI easily outperforms MFQ (Multilevel Feedback Queue) and closely follows EDF scheduler in the Hartstone[28] benchmark in terms of the first deadline miss but, outperforms both in the number of deadlines missed over a specific duration in which the processor utilization exceeds one hundred percent which means it does not suffer from the domino effect [1] of EDF and proves to be a good substitute in real-time applications [2].

The purpose of this work was to improve the performance of I+PI scheduler through the use of high-resolution timers and the tickless kernel design pattern which is relevant since the equations of dynamic systems do not usually match perfectly with periodic time steps unless endowed with a high enough resolution. Naturally, the higher the system timer's resolution, the more frequent the kernel invocation will be and thus the unnecessary overhead. However, during the investigations into I+PI, another potential responsiveness improvement regarding the tasks that yield the control voluntarily was discovered in which they would not gain the processor back in the same and next round in the case that their inactivity time is less than the end of the next round. For instance, consider a thread that sleeps after consuming half of its prescribed CPU time. If it wakes up during the current round, it just misses the remaining half of the CPU time allotted to it but, if it wakes up after the end of the round, not only the thread will lose its share from the current round, it is also not scheduled during the next round which would be problematic in the case that the round duration is long. This issue

[1] In the scheduling of tasks with deadlines, the domino effect is referred to the situation where all the deadlines are missed. This can happen with EDF scheduler when utilization is over hundred percent.

[2] Applications wherein the correctness of a computation relies not only on the logical outcomes of the computation but also on the time at which the results are produced are referred to as real-time[4]

was not revealed before[16] since the round duration was set to a very small value that prevented the gap from being large enough to miss the deadlines.

In fact, the inner loop was designed to account for fairness, based on its classic definition [12] that essentially is about assigning a prescribed amount of a shared resource (Processor's time in this case) to each task, however, a task may not need the whole share assigned to it and yields on purpose - for instance, consider a periodic task sending a signal from a wireless sensor node which has been assigned a larger window of time necessary to prepare and transmit the packet. The fact that tasks are not always striving for the resource impacts the responsiveness and this has led to a modeling error in I+PI that tries to force threads to consume as their prescribed CPU share for them which is not always possible.

In conclusion, this work provides a new control scheme along with proper actuation layer to solve the responsiveness problem of I+PI while dealing fairness nominally.

1.1 Brief literature review

The classical time-invariance assumption is often not met in real-life applications. Considering the OS kernel as an arbiter of shared resources, the assumption hardly fits the setting where tasks' demand of those resources does not remain constant during their lifetime. Therefore, it is inevitable to provide the kernel components with adaptation capabilities in such unpredictable environment. This fact is even more relevant when focusing on real-time or embedded systems with limited resources, power and (soft or hard) deadlines.

A common practice for the introduction of adaptive abilities in systems design is to extract and exploit feedback signals from the system at runtime that is equivalent to the closed-loop system design in control theory jargon. A classic example would be the MFQ scheduler which provides the system with different task queues and selects the next task to run from the first non-

empty queue with the highest priority. The priority would change during the runtime as the result of using feedbacks (For instance, the last time a task has been scheduled is used as a feedback to possibly increase its priority if it is considerably far in the past) and consequently, the CPU time allotted to the tasks are adapted. Therefore, it is not a new concept to use feedbacks as knobs for online adjustment of the system, however, especially during the last decade, research communities have shed light to the importance of utilizing control models besides choosing the right feedback signals. For instance, [29] proposes a perspective shift from traditional open-loop scheduling of network resources to a closed-loop one in order to mitigate the impact of limited bandwidth and variable workload on the QoC(quality-of-control) of networked control systems. The aim of this work was to maximize the overall QoC by dynamically allocating available network bandwidth through a codesign of control and scheduling and an integrated feedback scheduler that enables flexible QoC management in dynamic environments (under both underloaded and overloaded network conditions). The paper [2] is another research work utilizing feedback signals - and control theory more explicitly - to control CPU usage and memory consumption of a virtual database machine in a data center under a time-varying heavy workload through a design of multiple SISO (single input - single output) feedback controllers. One of the most recent works in this direction and pertaining to real-time systems is [1] which targets both energy consumption and processor utilization where the processor is provided with DVFS (Dynamic Voltage/Frequency Scaling). The paper [1] proposes a feedback scheduler maximizing battery life while minimizing deadline miss ratio by adjusting the processor speed in proportion to the available energy in the batteries and the processor utilization. All the aforementioned works follow the ARTIST2 project which was aimed at defining a roadmap on control of real-time computing systems[3]. In the vast majority of cases, the controlled item is the allocation of computing and communication resources.

Limiting the scope of the topic to the scheduling problem, the same tech-

nique is applicable through viewing the task pool as a system with measurable outputs that are used as feedback signals to adjust the policy of the scheduler. In this regard, in the article [15], Lu et al. 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 in which the latter corresponds in system-theoretical terms to feedback. 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. Notice that admission control, in general, is very popular in the context of web servers, where the tasks could be rejected to preserve utilization. Thus the application domain is the main difference between the referenced paper and I+PI (and therefore this thesis) i.e. in one case it being a server and in the other an embedded device. Consequently, the amount of control and the place where this control is introduced is different.

As the target environment for the proposed scheduler in this thesis is embedded devices that are limited in power and resources, it is worthwhile to highlight another remarkable trend in operating systems i.e. the "Tickless" kernels. Provided that such a kernel eliminates unnecessary system calls, advantages in terms of performance and power usage are delineated by various research works and many operating systems including Linux[18, 8, 9] have been switched in this manner for the same reasons. For instance, in [13] authors have characterized power consumption of the POWER6 system in different layers including the OS and, have successfully demonstrated the remarkable effect of tickless Linux kernel.

1.2 Contributions

This thesis is a contribution to the research trend trying to incorporate control theory into the design of operating system components with the focus on single processor scheduling and in fact, pursues the thesis "Control based

design of OS components"[26]. This section delineates briefly the additional work performed through this thesis with respect to [26] that encompasses the design and implementation of I+PI which is the ground for the new scheduling mechanism (Multiburst scheduler) introduced in this thesis. Both I+PI and Multiburst schedulers are implemented in C++ and on top of the Miosix kernel[7] which is an open source OS kernel running on 32-bit microcontrollers. It provides a platform for the multi-threaded programming model where applications are statically linked with the kernel. Figure 1.1 depicts architecture of Miosix in which blocks shaded in gray correspond to amendments made as the contribution of this work.

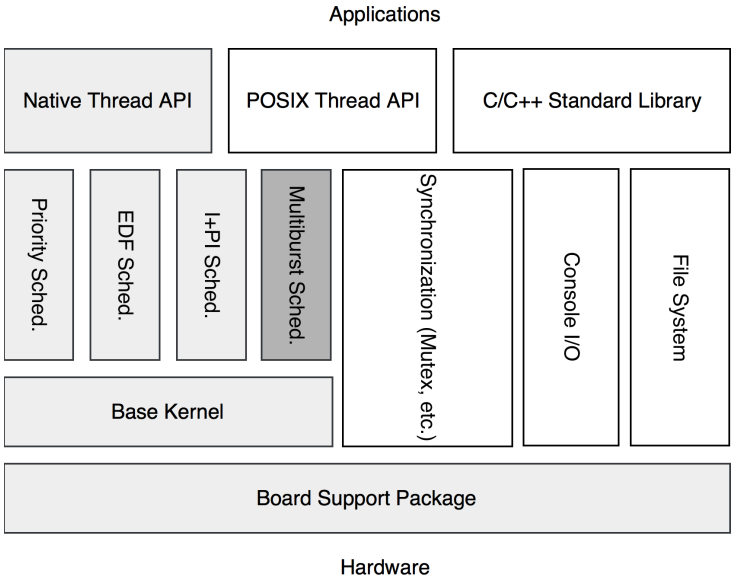

Figure 1.1: Miosix kernel architecture

As it is mentioned earlier, the work can be categorized into two main groups i.e. transforming Miosix into a tickless kernel, and design and implementation of the new scheduler.

- Since Miosix kernel is targeting multiple architectures which in turn provide different hardware capabilities for timing and synchronization, the first task was to design an abstract uniform interface for the kernel

to provide the required base for both periodic and continual system timer. The interface enables the kernel or scheduler to set the next interrupt in nanoseconds (and regardless of the resolution) while processing the current one.

- Modification of the startup flow to make the kernel inline with the new timer interface. This involves bypassing of the usual periodic systick timer [3] , setting up the selected timer peripheral and configuring the desired resolution through the instantiation of the proper implementation of the interface.

- Implementation of timer/counter drivers according to the new interface for the various architectures and embedded boards supported by Miosix.

- Principally, the kernel was using the systick for two reasons: task preemption and management of sleep/wake-up of threads. Thus, the next modification to the kernel was pertaining to the system calls in charge of initiating preemptions and a subset of Miosix's native thread API. Moreover, since Miosix is transformed to a tickless kernel, all the time-dependent routines have been amended to operate in terms of nanoseconds rather than "kernel ticks" which can vary from a configuration to the other.

- With the new interface in place, an additional task of setting the next context switch point is implemented for Miosix schedulers as there are no automatic periodic interrupts anymore. This enables the scheduler to avoid prescribing fixed CPU allocation time and behave in an adaptive fashion. Whether the scheduler is exploiting this feature or not,

[3]Systick is the default timer/counter peripheral circuit found in every computing system which supports multi-tasking and is used to fire an interrupt according to a configurable period of time. OS kernels use this peripheral to gain the control back from the running task and perform a context switch.

all the existing schedulers should set the next context switch point in order to fully integrate with the new tickless kernel.

- A comprehensive review of the I+PI scheduler has been done and the root cause of its weakness regarding responsiveness is detected. One of the major contributions of the thesis is the design of a new control scheme to establish a better trade-off between response time and the number of context switches.

- The new control scheduler (Multiburst) is implemented according to the generic interface provided by Miosix and on top of the new tickless kernel. After performing a number of profiling tests to verify the functionality, some standard benchmarks have been used to evaluate the performance and cost of the Multiburst scheduler with respect to that of EDF, MFQ, and I+PI.

1.3 Thesis organization

The thesis is structured as follows.
Chapter 2 illustrates the key concept of tickless kernels and presents the recent practice in existing operating systems. Then it provides a detailed description of the previous time system of Miosix kernel and integration of the new method along with a general view of timer/counter drivers implemented to provide the required hardware support. In chapter 3 the scheduler subsystem of Miosix and implementation of I+PI is described along with the transition steps to derive the new scheduler. Chapter 4 is dedicated to the extensive benchmarks used to evaluate the correctness and performance of the new approach and the results are compared to that of Round Robin, EDF and I+PI schedulers with/without presence of dynamic ticks provided by the tickless kernel. Finally, the last chapter draws conclusions regarding the work presented in this thesis and outlines the future directions and possible extensions to the work.

Chapter 2

Tickless kernel

The system timer is one of the most crucial requirements of a multi-tasking operating system from hardware support point of view. It is used in order to fire an interrupt request every once in a while so that the kernel can regain the control by preempting running tasks mainly in order to switch the context to the next task to run according to a scheduling policy. A common methodology that has been practiced in the design of virtually all types of kernels was to utilize a periodic timer/counter to generate the aforementioned interrupt request at regular intervals, gave birth to the concept of the tick of the kernel.

However, since the kernel tick is not only used for context switches, but also for timekeeping and waking up tasks that sleep, the kernel tick imposes a tradeoff between the resolution of the `sleep` primitive (and the one to return the current time, at least in kernels that use the tick for both) and the overhead of the ticks themselves. In theory, one could have a ticked kernel with wakeups at microsecond resolution, but it would require one million tick interrupts per second, effectively overloading the CPU with interrupts.

Fortunately, the availability of timer/counter circuits with large registers for timekeeping (counter and capture registers) has endowed system software with the possibility to have the processor interrupted whenever it is necessary rather than relying on the periodic IRQ for kernel intervention and hence,

one could design a tickless kernel. A tickless kernel refers to an OS kernel that employs an aperiodic time scheme wherein timer interrupts are only delivered as required by the kernel space code. Moreover, avoiding unnecessary ticks allows the kernel to put idle processor cores in the deep-sleep state as much as possible in order to decrease power consumption. Although the advantages of this fundamental change may be more tangible in real-time and/or embedded kernels which are run on lower performance hardware platforms with limited power supplies, general purpose operating systems such as Linux, Minix[1], Solaris, Windows and OSX have also undertaken the same path of transforming their kernels to a tickless one. The Linux kernel on s390 from 2.6.6[27] and on i386 from release 2.6.21 can be configured to turn the timer tick off (tickless or dynamic tick) for idle CPUs using CONFIG_NO_HZ, and from 3.10 with CONFIG_NO_HZ_IDLE extended for non-idle processors with CONFIG_NO_HZ_FULL. The XNU kernel[2] in OSX 10.4 on, and the NT kernel in Windows 8 kernel [3] are also reported to be tickless. The Solaris 8 kernel introduced the cyclic subsystem which allows arbitrary resolution timers and tickless operation[4].

2.1 Time subsystem in Miosix

This section provides essential information about the time subsystem of Miosix kernel necessary to understand the transition into a tickless counterpart that is exploited to design a highly responsive task scheduler later, hence, the description provided here pertains to the state of the Miosix kernel before the fulfillment of this thesis.

Before stepping into implementation of the kernel it is worthwhile to point out an important naming convension in Miosix that hints the state in which

[1]http://wiki.minix3.org/doku.php?id=tickless

[2]See e.g.: https://github.com/darwin-on-arm/xnu/blob/master/osfmk/arm/rtclock.c

[3]Bright, Peter. "Better on the inside: under the hood of Windows 8", October 2012

[4]Bryan Cantrill (former Solaris kernel engineer) comment at https://news.ycombinator.com/item?id=13091162 (Retrieved 2017-01-07)

a routine should be called. The names prefiexed by `IRQ` should be called if and only if interrupt requests are disabled and those that start with `PK` can be invoked securely only when the kernel is paused, hence, there would be no preemption or context switch.

Limiting the scope of the discussion to task management and scheduling, Miosix kernel provides a native definition and implementation for thread-/process, a pointer to the current running thread and its context information and a linked list sorted by wake-up times of the threads (a.k.a sleeping list) and some routines to manage the list. On the other hand, the scheduler provides a function `IRQfindNextThread` which is responsible for setting the next thread to run by changing the current thread pointer, saving and, switching the context defined in the kernel. This function is called whenever a thread yields the execution [5] or upon a preemption call invoked by the `Systick` interrupt (fired by the system timer). This workflow is distributed throughout the following parts of the kernel.

- **Portability Interface** which abstracts the hardware layer and endows the kernel with an architecture independent interface which contains:
  - Architecture dependent startup code (function `IRQportableStartKernel`).
  - Sequence of instructions required to perform a context switch and enable/disable interrupts.
  - Interrupt handlers for performing preemption (function `Systick_Handler`) and yielding (function `ISR_Yield`). `Systick_Handler` will call another function (`IRQtickInterrupt`) which accounts for a single tick of the kernel and leads to a context switch.
- **The basic kernel interface** that defines and implements the following independently of the hardware which is encapsulated by the portability interface:

[5] upon calling sleep primitives, calling peripheral services that lead to an I/O waiting status or, the will to terminate.

- Hardware independent startup routine which creates the main thread[6] and the Idle thread [7] and calls the function `IRQportableStartKernel` to configure and setup the interrupt controller device and finally the system timer to enable kernel's ticks.
- Functions and RAII classes[8] for toggling the interrupts
- Functions and RAII classes to pause/resume the kernel, hence, enabling/disabling preemption which should be respected by scheduler.
- A native definition and implementation of threads which particularly includes front-end functions for a thread to yield or sleep. The sleep functions are fed with number of ticks of the kernel which in turn correspond to ticks of the system timer.
- Information about the current thread and context which is used by the scheduler's code to perform context switches
- A linked list sorted by the kernel's tick in the order which threads desire to become ready again and the two functions `IRQaddToSleepingList` and `IRQwakeThreads`. The former is called by threads whenever a thread desires to sleep for a particular amount of ticks while, the latter is called upon every tick (by `IRQtickInterrupt`) to check whether it is time to activate some sleeping threads or not. This check being done on a fixed time step, implies imprecision when dealing with time-critical tasks or unnecessary kernel code execution when the first thread activation point is far in future.

Therefore, for Miosix to be transferred into a tickless kernel, two major workflows should change i.e. whenever a tick occurs and upon current thread's

[6] In Miosix, the user's application is compiled with the kernel through a function named `main()` which is executed by the main thread. This thread can spawn other threads later on.

[7] The Idle thread will be selected when there is no active task to run e.g. when all the threads are in a waiting/sleeping state

[8] Resource acquisition is initialization

will to sleep. Figure 2.1 represents the call graph for the tick processing in the kernel.

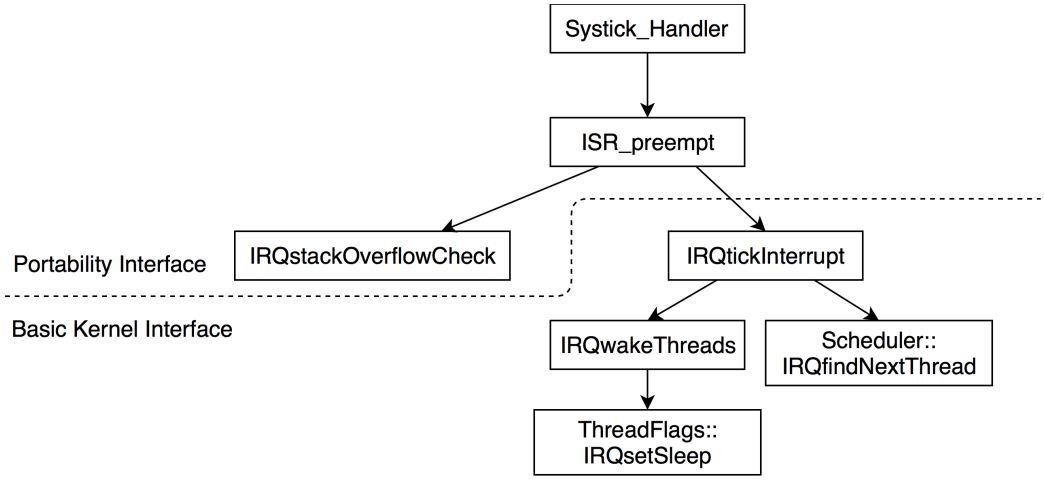

Figure 2.1: Miosix - Kernel tick call graph

The `IRQtickInterrupt` can be called whenever a preemption should take place or just a thread should be woken which does not necessarily lead to a context switch. Thus, in tickless setting, this flow should change in particular in a way to distinguish between these two cases so as to eliminate the dependency of the time domain resolution of sleep primitives with that of scheduler interventions. Moreover, since there would be no periodic timer interrupts, the scheduler should set the next preemption point in `IRQfindNextThread` function.

As for the second workflow pertaining to the sleep primitive routines which is depicted in figure 2.2, it is clear that sleep functions should not depend on kernel tick any longer and instead rely on a universal concept i.e. the time unit (e.g. nanoseconds). In this regard, the sleeping list should be ordered by the actual time unit and `IRQaddToSleepingList` should function aligned with the scheduler to keep the next interrupt to the minimum of the first wake-up time (head of the sleeping list) or the next preemption point set by the scheduler during the previous preemption.

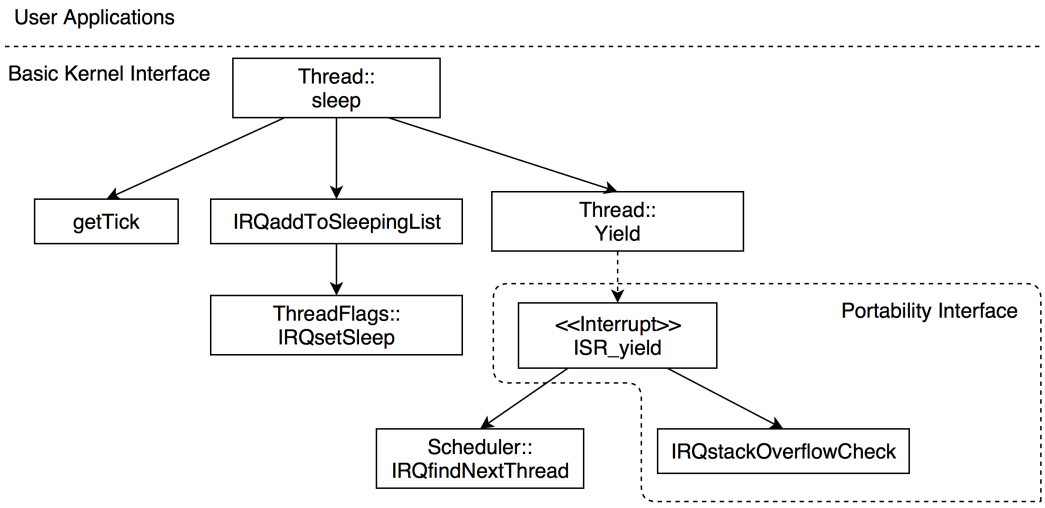

Figure 2.2: Miosix - Sleep primitive call graph

2.2 Transformation of Miosix into a tickless kernel

In addition to the benefits enumerated about tickless kernels and the trend of other operating systems towards this design, having Miosix kernel operate in tickless mode is a crucial requirement for the purpose of this thesis. As it will be explained deeper in the next chapter, the CPU time alloted to the tasks may significantly vary due to some tuning parameters and the control scheme behind the new scheduler proposed by this work. Therefore, the following steps have been taken in order to provide the scheduler with an aperiodic timing system - hence the ticklessness - while keeping the changes so few that do not disrupt the operation of other modules relying on the previous time system of the kernel.

- Defining a new interface for the kernel as the unified model of timer drivers used to manage preemptions in different architectures
- Modification of kernel startup routines to switch from `systick` to the new interface

- Integration of task sleep/wake-up management in the kernel and thread interface with the new time subsystem
- Decoupling kernel's time unit from the underlying hardware from user perspective by introducing an internal tick-to-nanosecond conversion
- Implementation of new high resolution timer drivers for Miosix supported architectures according to the new interface
- Integration of existing schedulers with the new interface, making them able to cause preemptions.

2.2.1 Context Switch Timer interface

The simple scheme of having an interrupt handler being invoked periodically by the `systick` timer which in turn calls the preemption routine, is replaced by the interface shown in the figure 2.3 that is defined to support aperiodic configuration of next preemption point and automatic conversion of timer ticks to/from nanoseconds so as to separate timing logic of other components and user applications from the speed configured for the hardware timer.

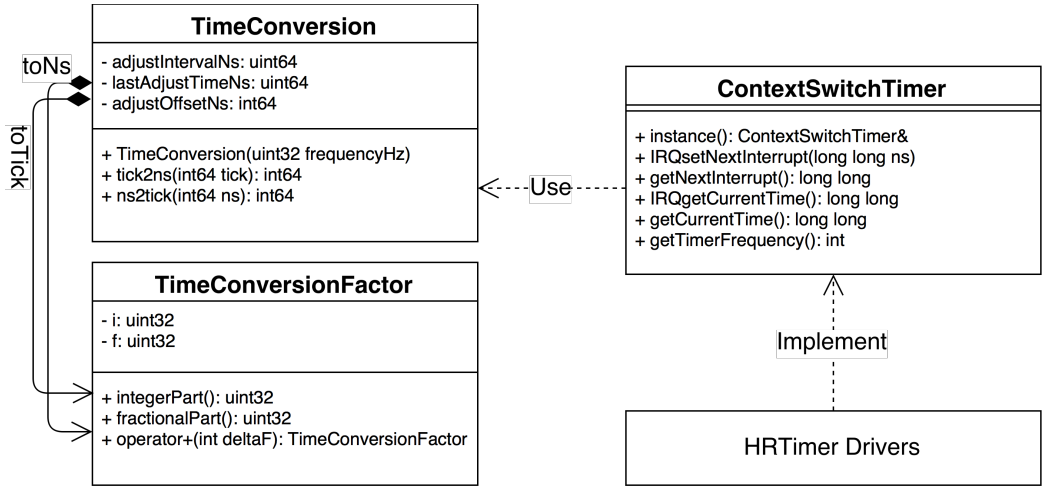

Figure 2.3: Context Switch Timer interface

Depending on the architecture the kernel is compiled for, there should be a proper implementation of the `ContextSwitchTimer` as a singleton class which is instantiated by the hardware dependent kernel startup routine i.e. `IRQportableStartKernel`. It is expected that the implementation provides an apparently infinite time horizon which always moves forward through the use of large enough hardware or hardware/software counter registers. For instance, a high resolution timer operating at 1 GHz (one tick every nanosecond) with a 64-bits counter register, the system can keep track of the time without any rollovers for 584 years which is large enough to be considered an infinite upperbound for any system's uptime.

However, most of the timer/counter circuits usually support 16/32-bit registers that would easily overflow in a very short time - 65 microseconds and 4.3 seconds respectively - and therefore they must be expanded through software controlled memory by the driver.

2.2.2 Implementation

The `ContextSwitchTimer` interface illustrated in the previous section has been implemented and tested on the architectures reported by the table 2.1 for which none of the target architectures provides 64-bits registers and thus, the counter is extended by the driver through additional variable. Of course the expansion is achieved at the cost of additional interrupt requests due to overflow of the hardware register and therefore it is really important to limit the overflow interrupt handler to few instructions and decrease the timer's frequency as low as possible.

	Architecture - SoC	**Peripheral Timer Used**	**Register Capacity**
1	CortexM3 - STM32 [23]	TIM2	16 bits
2	CortexM3 - EFM32GG [20]	TIM3	32 bits
3	CortexM3 - STM32F2 [21]	TIM2	32 bits
4	CortexM4 - STM32F4 [22]	TIM2	32 bits

Table 2.1: Architectures for which tickless Miosix has been ported into

The listings below present the implementation of functions defined by `ContextSwitchTimer` in pseudocodes equivalent to the C++ implementation of drivers for the target archtectures.

The timers provide two registers keeping track of current tick count and a checkpoint set by the software. When the value of the former becomes equal to the latter a "Capture" IRQ will be fired that is the usual capacity found in virtually all the timer/counter peripherals and is exploited to notify the kernel about the next time event. Therefore, the driver need two variables as the most significant 32 bits of the total value.

Algorithm 1: ContextSwitchTimer driver - Initialization

Set timerBits = 32 or 16 according to target timer peripheral ;
Set overflowInc = 1 shifted left by timerBits;
Set lowerMask = overflowInc - 1;
Set upperMask = 0xFFFFFFFFFFFFFFFFLL-lowerMask;
Set ms32time = 0; //most significant 32 bits of counter
Set ms32chkp = 0; //most significant 32 bits of checkpoint
Clear lateIrq; // Boolean variable indicating a checkpoint in the past
Enable the peripheral timer clock;
Set timer's mode to up-counter ;
Enable interrupt flags for Compare/Capture channel of choice;
Enable the interrupt controller to pass IRQ of the timer;
Set the timer's prescalar to tune the tick frequency according to lowest desirable time resolution ;
Instantiate an instance of TimeConversion class;
// Reset Compare/Capture register and start the timer
Set timer.CCR = 0;
Enable the timer;

In order to set the next timer intervension, a time value in nanoseconds should be taken as the input and its equivalent in timer's tick should be stored in the checkpoint upper part variable `ms32chkp` concatenated with

the register. The driver should also react promptly if the point is in the past as the time is not periodic and always increasing and thus the event could be missed eventually. Therefore, after setting the next checkpoint on the timer, the driver should check and raise an interrupt if the point is in the past. The order matters since otherwise, it is possible to have a checkpoint close enough to the current time that would be passed before the timer register is set.

Algorithm 2: ContextSwitchTimer driver - Setting the next event

Input: ns: Desired time point in nanoseconds

Precondition: Interrupts should be disabled

```
tick = ns2tick(ns);
ms32chkp = tick BIT_AND upperMask;
timer.CCR = tick BIT_AND lowerMask;
if checkpoint is in the past then
    Set lateIrq;
    Set timer's interrupt PENDING through interrupt controller unit;
end
```

Reading the next event set by the software is as simple as a bitwise OR between the variable `ms32chkp` and the checkpoint register.

Algorithm 3: ContextSwitchTimer driver - Reading the next event

Precondition: Interrupts should be disabled

```
nextEvent = ms32chkp BIT_OR timer.CCR;
return tick2ns(nextEvent);
```

Reading the current time may appear to be feasible by just performing a bitwise OR between the software variable `ms32time` and the hardware counter register, but it involves more delicacy since

- The user code may have disabled interrupts, spent some time with interrupts disabled and then have called this function.
- If a timer overflow occurs while interrupts are disabled, the upper bits that are stored in `ms32time` are not updated

and thus, the routine may return a wrong value. To prevent this issue,

the overflow pending bit of the timer should be checked and current time value be adjusted accordingly. Reading the overflow pending bit is not an atomic operation and the counter may roll over exactly at that point in the time. Thus, the timer's value should be read a second time to examine if the counter has rolled over. This algorithm imposes a limit on the maximum time interrupts can be disabled which is equal to one hardware timer period minus the time between the two timer reads in this algorithm.

Algorithm 4: ContextSwitchTimer driver - Reading the current time value

```
if Interrupts are not disable then
    Disable interrupts;
end
Store timer.CNT in counter;
if timer.Overflow bit is pending AND timer.CNT ≥ counter then
    Store (ms32time BIT_OR counter) + overflowIncrement in result;
else
    Store (ms32time BIT_OR counter) in result;
end
if Interrupts were not disabled at the beginning then
    Enable interrupts again;
end
Return result;
```

Last but not least, the order in which the overflow bit and timer's value are read matters and may introduce a race condition if applied otherwise. Were the driver to read the timer's value prior to the overflow flag, the counter may roll over right after the second time the code reads timer's value and therefore the result would be off by one epoch in future.

Finally, the driver should properly handle overflows of hardware counter and matches of capture/compare channel designated to keep track of checkpoints set by kernel or user code. In the first case, the expected response would be an increment in the higher 32 bits of the counter variable `ms32time` while for the latter event a call to `IRQtimerInterrupt` (equivalent to `IRQtickInterrupt`

in the tick-based Miosix) should happen in the case that both software and hardware parts of the checkpoint are matched to the current time. The following listing represents a typical timer interrupt handler just mentioned.

Algorithm 5: ContextSwitchTimer driver - Interrupt handler

```
Precondition: Interrupts should be disabled
if There's a match on Capture/Compare Channel OR lateIrq is set
 then
    Clear Capture/Compare match flag in the hardware;
    if ms32time==ms32chkp OR lateIrq is set then
        Clear lateIrq;
        call IRQtimeInterrupt;
    end
end
if Timer's overflow flag is set then
    Clear overflow flag in hardware;
    ms32time += overflowIncrement;
end
```

As the final remark on the interrupt handler, one may consider the situation where the checkpoint is set a few ticks after an overflow event and conclude that the checkpoint would be missed as the code first checks the equivalence of higher 32 bits of the checkpoint to the higher 32 bits of the current time in which the latter is not updated yet. The race condition does not occur as the algorithm clears individual flags for each event that is processed and others will remain pending. Thus, in the aforementioned scenario, the interrupt handler is called twice wherein the first time the overflow would be applied and the second time the checkpoint is reported.

Considering the modifications to the kernel introduced in this chapter, it is evident that the impact would be significant enough to easily lead to malfunction of the kernel, schedulers, device drivers and other components relying on the time subsystem. Thus, as an effort to prove the reliability and correctness of the new time subsystem, various tests have been executed to

inspect potential inconsistencies between different components of Miosix and the new tickless implementation. In this regards, the official test suite that is provided by the Miosix kernel [9] has been used.

[9]The test suit can be found under: https://github.com/fedetft/miosix-kernel/tree/master/miosix/_tools/testsuite

Chapter 3

The control based tickless scheduler

Regardless of the policy conducted by any scheduler, the problem corresponds to a simple phenomenon that is present in every description provided for task scheduling which can be modeled perfectly via control theory as described below:

The scheduler interrupts the execution flow every once in a while and assigns to a subset of tasks some particular processor's time to run - which is called CPU burst throughout this document and is also known as "scheduling bandwidth" or "reservation period" - until the next scheduler's intervention. Tasks will consume a certain amount of CPU time in this time span which is not necessarily equal to the time designated for them.

3.1 Scheduling as a control problem

In control terms, a pool of tasks running on a processor core can be viewed as a physical plant and a regulator in which the former accounts for the phenomenon constituting the essence of the problem and the latter corresponds to the scheduler. For each task, the plant model can be translated into the difference equation

$$\tau_t(k) = b(k-1) + \delta b(k-1) \tag{3.1}$$

where k is the index of scheduler interventions, τ_t is the CPU time task t has consumed between time span from $k-1$ to k and, b the burst allotted to t by the scheduler. The term δb is the disturbance that accounts for any action on the phenomenon other than that of allotting b, such as for example anticipated CPU yields, delays in returning the CPU whatever the cause is, and so forth. Considering the entire pool, one can obtain the model

$$\begin{cases} \tau_t(k) = b(k-1) + \delta b(k-1) \\ t(k) = t(k-1) + \sum b(k-1) + \sum \delta b(k-1) \\ \tau_r(k) = \sum \tau_t(k) \end{cases} \tag{3.2}$$

where summations are over the pool, t accounts for the system time and, τ_r is the time between two subsequent scheduler interventions regardless of the number of tasks that were allotted a nonzero burst and the order.

Model 3.2 respects the rules, in that it is entirely physical, accounts for all the entities acting on the phenomenon, and exposes both the actually used CPU times and the time between two subsequent instants when the scheduler regains control. Figure 3.1 better illustrates the meaning of the quantities involved in this model along with an example of their behavior in time, scheduler operations (i.e. burst allocation and context switches).

τ_t values being the effectively elapsed times while a tasks were in execution and b_t values the desired bursts set by the scheduler, three cases are anticipated

- The first task exhibits the perfect situation where the execution time τ_{t1} is equal to the burst b_1 which is alotted to it during the first burst allocation time in the figure.

- It may happen that the task consumes less than what was designated for it due to for example waiting on a mutex or I/O operation or going

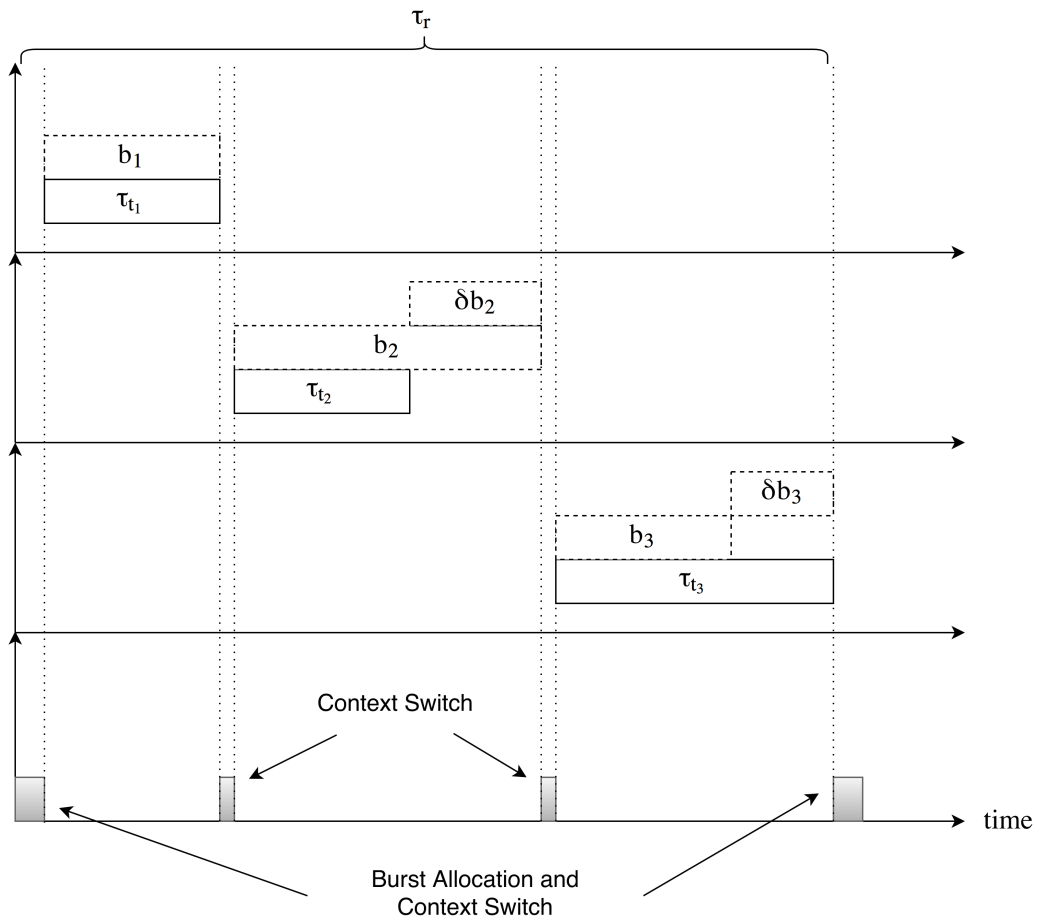

Figure 3.1: Task pool model and behavior

to sleep. The second task represents such a situation in which τ_{t2} is less than b_2 and thus, the disturbance term δb_2 is negative.

- If the task is not preempted in time, the actual CPU time consumed by the task would be greater than its burst and therefore, the error will be positive. An example would be a task that performs a critical section which entails disabling interrupts or pausing the kernel right at the end of its burst.

Typical metrics used to evaulate behavior of schedulers such as fairness, responsiveness and etc. can be suitably reformulated based on the quantities

introduced by 3.2. In fact the I+PI and the Multiburst schedulers are built on top of this model and provide a controller plus an actuation layer which is an algorithm that wraps around the controller to fully integrate it with kernel interface.

3.1.1 The I+PI scheduler

This section is devoted to establish the necessary background needed to better understand the motivation and the design flow upon which the Multiburst control scheduler is based, through an in-depth examination of its predecessor i.e. the I+PI scheduler. The overal I+PI scheduling scheme consist of the model 3.2 plus regulators introduced by the I+PI that are completely represented as a block diagram in figure 3.2. The scheme is used to synthesize the I+PI controller, assess the policy, perform of verification-oriented simulations prior to implementation and integration of I+PI to the kernel.

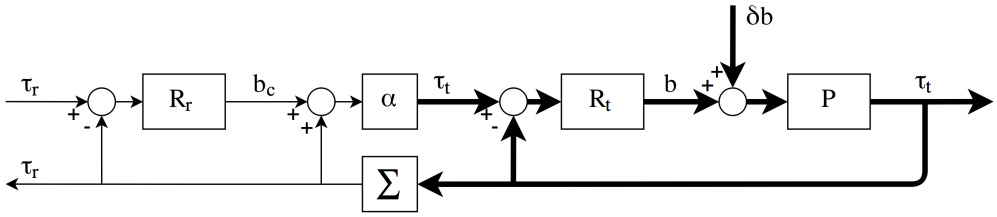

Figure 3.2: Control block diagram of I+PI scheduler

In this scheme, the transfer function P together with the summation node on the pool corresponds to the "controlled plant", that is, the first equation in 3.2 with the inputs b and δb - that account for bursts alotted to tasks and disturbances respectively - and the output τ_t. The third equation in 3.2 which produces the round duration τ_r is realized through the block denoted by Σ. All the other blocks compose the controller - hence the I+PI scheduler - which its aim is to guarantee responsiveness through maintaining round duration to the desired set point and to assure fairness by keeping track of processor usage of tasks againts the fraction of round duration configured for them. As for each objective, the designers have considered a loop and

a regulator resulting to a cascade type control scheme where the loops are nested.

Inner Loop: The inner loop is composed of the task pool and a diagonal integral (I) regulator, realized by block R_t, whence the first part of "I+PI". Since also model 3.2 is diagonal as for the $b \mapsto \tau_t$ relationship, the result is a diagonal (or "decoupled") closed-loop system that can be studied by simply considering one of its scalar elements. Also, the choice of the (diagonal) I structure stems from the pure delay nature of said elements-evidenced by the first equation in 3.2-as a typical control design procedure; see Franklin et al[10]. In view of this, if for each burst b_i an integral discrete-time controller with gain K_I is adopted, that is,

$$b_i(k) = b_i(k-1) + K_I(\tau^{\circ}_{t,i}(k-1) - \tau_{t,i}(k-1)) \tag{3.3}$$

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

$$\begin{bmatrix} \tau_{t,i}(k) \\ b_i(k) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -K_I & 1 \end{bmatrix} \begin{bmatrix} \tau_{t,i}(k-1) \\ b_i(k-1) \end{bmatrix} + \begin{bmatrix} 0 \\ K_I \end{bmatrix} \tau^{\circ}_{t,i}(k-1) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \delta b_i(k-1) \tag{3.4}$$

Observing system 3.4 with inputs $\tau^{\circ}_{t,i}$, δb_i and output $\tau_{t,i}$, it can be concluded that the disturbance is asymptotically rejected and the set point followed with a response time (in rounds) dictated by K_I provided that the eigenvalue magnitude is less than the unity, that is, $|1 \pm \sqrt{1-4k_1}| < 2$. A good default choice is to have two coincident eigenvalues in 0.5, hence $K_I = 0.25$. Higher values of k_I make the controller respond "more strongly" to the difference between the desired and achieved τ_{t_i} , thus making the system faster at rejecting disturbances (owing to a prompt action) but easily producing oscillatory responses to set point variations (owing to a possibly transiently excessive action); lower values of K_I , intuitively, cause the reverse to happen. Figure 3 illustrates the matter, and shows why 0.25 could be used as default for a scheduler with no real-time requirements, and 0.5 could be used for a soft real-time one.

Outer Loop: Once the inner loop is closed, the convergence of the actual CPU times to the required ones is ensured since choosing eigenvalues with magnitude lower than the unity ensures asymptotic stability, and the regulator contains an integral action[10]. To determine the set point τ_t° , an outer loop is used that provides an additive correction (b_c in figure 3.2) so as to maintain the round duration τ_t to a prescribed value τ_t°; the computation of b_c is accomplished by block R_r. It can be verified that choosing a single K_I for the inner loop results in a $b_c \mapsto \tau_r$ relationship independent of α. A suitable controller structure for the outer loop, along considerations analogous to those that led to the I one for the inner loop, is then the Proportional plus Integral one (PI), whence the rest of "I+PI". Reasoning in the same way as for 3.4 this leads to determine the closed outer loop's behavior in time as ruled by

$$\begin{cases} \tau_r(k) & = 2\tau_t(k-1) - (1+K_IK_R)\tau_r(k-2) + K_IK_RZ_R\tau_r(k-3) \\ & \quad +K_IK_R\tau_r^\circ(k-2) - K_IK_RZ_R\tau_r^\circ(k-3) \\ X_R(k) & = X_R(k-1) + K_R(1-Z_R)(\tau_r^\circ(k-1) - \tau_r(k-1)) \\ b_c(k) & = X_R(k) + K_R(\tau_r^\circ(k) - \tau_r(k)) \end{cases} \tag{3.5}$$

,

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

$$z^3 - 2z^2 + (1+K_IK_R)z - K_IK_RZ_R = 0 \tag{3.6}$$

in the unknown z, have magnitude less than the unity, which provides easy parameter bounds, while disturbance rejection is still guaranteed by the contained closed inner loop. As a result of the synthesis process just sketched, the I+PI algorithm is unambiguously defined as follows.

Algorithm 6: I+PI controller algorithm

Initialize the I and the PI state variables;
for *each scheduling round* k **do**
  Read the measured $\tau_t(k-1)$ i.e. the CPU times used by the N_t tasks in the previous round;
  Compute $\tau_r(k-1) = \sum_{i=1}^{N_t} \tau_{t,i}(k-1)$ i.e. the duration of the previous round;
  Read $\tau_r^\circ(k-1)$ i.e. the required (set point) duration of the previous round;
  if *cardinality of the task pool or parameters have changed* **then**
    Reinitialize $b_i(k)$ to $\alpha_i \tau_r^\circ$;
    Apply saturations to $b_i(k)$;
  else
    Apply the PI regulator to obtain the burst correction: $b_c(k) = b_c(k-1) + k_{rr}(\tau_r^\circ(k-1) - \tau_r(k-1)) - k_{rr} z_{rr}(\tau_r^\circ(k-2) - \tau_r(k-2))$;
    Apply saturations to $b_c(k)$;
    Re-compute the vector $\alpha(k)$ for tasks;
    for *each task* i **do**
      // Apply the I regulator to obtain the burst vector $\tau_{t,i}^\circ = \alpha_i(k)\tau_r^\circ(k)$;
      $b_i(k) = b_i(k-1) + k_{ti}(\tau_{t,i}(k) - \tau_{t,i}(k-1))$;
      Apply saturations to $b_i(k)$;
    end
  end
  Activate the N_t tasks in sequence, preempting each of them when its burst is elapsed;
end

To fully implement I+PI scheduling algorithm, "set point generation" which can be further divided into "overload detection and rescaling" and "reinitialization and feedforward" should be integrated to the I+PI controller algorithm. Set point generation needs running only when changes occur in

the task pool, the required CPU distribution, the required round duration or any combination thereof.

- **Overload Detection and Rescaling** refers to the recomputation of the CPU shares (for all nonblocked tasks) to have unity sum. For instance, when a task is blocked its share becomes zero and thus others can benefit a temporarily increased CPU usage. On the other hand, if an overload is detected, the shares should be rescaled with relative importance to unity.

- **Reinitialization and feedforward** policy is introduced to enhance the scheduling performance dynamically in the presence of task blocking events such as a task sleeping or waiting on a Mutex. Reinitialization refers to resetting internal state of I+PI to its default value whenever the task pool parameters change. This includes resetting the saturated integral regulator of a blocked task, therefore improving the dynamic response to task blockings. The feedforward policy is instead grounded on the fact that task blocking is a measurable disturbance which the scheduler is informed about through kernel API. This allows to further improve the dynamic response by changing the I+PI set points, namely by setting to zero the α elements corresponding to blocked tasks and distributing the round time among nonblocked tasks only.

3.1.2 Motivations for a new control scheduler

Albeit I+PI's performing near EDF in Hartstone benchmark as reported in [16] and also in this thesis, there are some drawbacks regarding this scheduler which gave birth to the Multiburst control scheduler. As it is mentioned before, the PI and I control loops in I+PI are designed and tuned to assure responsiveness and fairness respectively, by keeping the round duration and distribution of processor shares to some desirable set points.

One of the problems with this algorithm is that whenever a thread sleeps

(or waits for I/O) before the new round is begun - which is the point that the corresponding bursts are computed for the threads - since the thread is in a waiting state, it does not gain CPU time even if it wakes up again during the current or even the next round. This imposes a maximum length for round time duration if there are periodic tasks with deadlines in the sense that τ_r should be less than half of the smallest period among the tasks. This issue is better illustrated in the figure below.

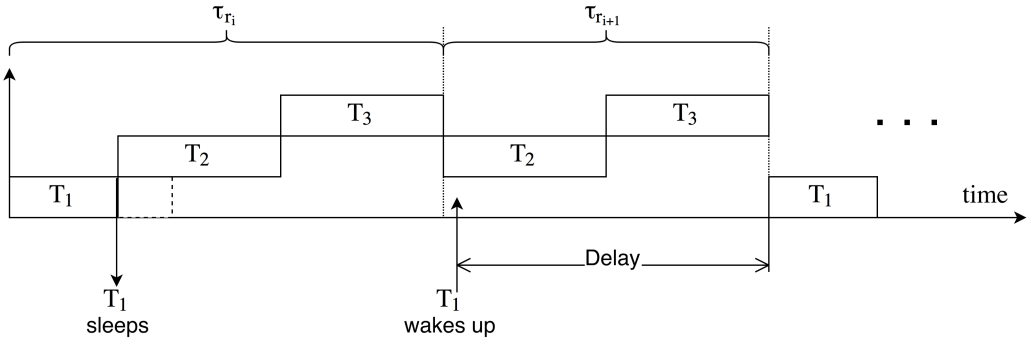

Figure 3.3: Reponsiveness problem of I+PI with tasks that give the CPU up early

Figure 3.3 exemplifies the problem with three threads equally sharing the CPU in which the first task goes to sleep earlier than the set point and thus can be delayed more than a full round duration[1] for it will be absent from the next round and that would lead to deadline misses. It is noteworthy that higher number of tasks in the pool leads to longer round duration according to the I+PI algorithm. Support of multiple bursts for each task inside a single round in the case that the thread has still reasonable amount of remaining time to run is our solution for this problem and thus, it will relax the constraint on the maximum value for τ_r. In other words, while the round duration is still depending on the pool size, the delay demonstrated by the example would not be affected by the round duration anymore. Regarding the inner loop and its impact on fairness, consider following remarks:

[1]The best case would be being delayed equal to minimum burst alloted to other threads and the worst case would be a delay of twice the sum of all other bursts

- The consumed CPU time being less than the allocated burst is compensated by the inner loop via increasing thread's burst in next rounds which may lead to saturation and this is in fact anticipated in the algorithm by applying an upper bound to b_t. This strategy is proper when the thread gives the control up deliberately for example by calling sleep primitives which may be interpreted as the task does not need the processor share assigned to it. Moreover, it is counter-intuitive to force a task to consume more than its demand and is also in contradiction with responsiveness which matters the most in real-time applications. So, either the scheduler should differentiate between mandatory/voluntarily yields or deal with fairness nominally.

- The actuation layer around I+PI forces the controllers to be re-initialized frequently and in the case that the cardinality of the pool changes too often the control scheme becomes useless and is reset every round and falls back to a blind round-robin like scheduling.

The problem with the inner loop stems from the premise that the threads are always striving for the processor and if an early scheduler intervention happens, the remainder burst should be given back to the thread in future. In the case of this thesis, since real-time applications are the primary focus, the problematic inner loop has been eliminated and fairness is being dealt nominally in the sense that the task may only use its remaining burst from the current round which is practical since the support of multiple bursts allows longer round durations.

Figure 3.4 represents the expected task execution sequence for the example provided by figure 3.3 when applying the Multiburst scheduling on the pool.

It is noticable that the new strategy, while improving the responsiveness of the first task, does not affect other threads from responsiveness/fairness point of view.

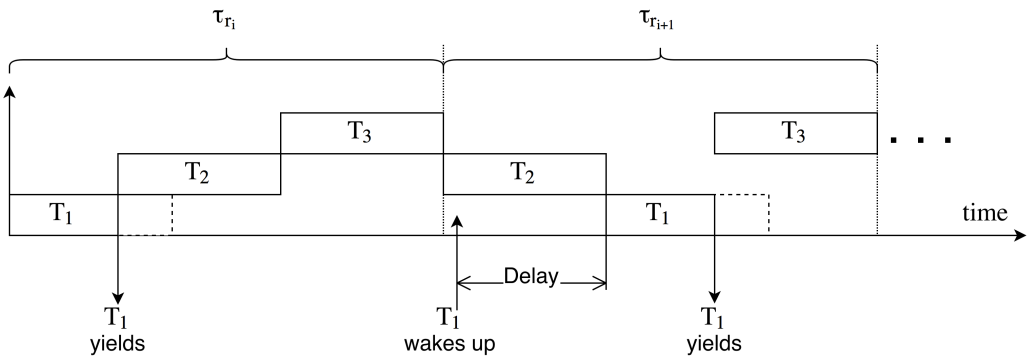

Figure 3.4: Expected behavior of the Multiburst scheduler in the case of I+PI's problem

3.2 The Multiburst scheduler

3.2.1 Design

The figure 3.5 depicts the control model associated with the new scheduler in which there is only a single regulator $C(Z)$ in charge of keeping the round time τ_r to the prescribed value τ_r°. The same reasoning of the outer loop being Proportional-Integral in $I + PI$ also applies to $C(Z)$ in this scheme.

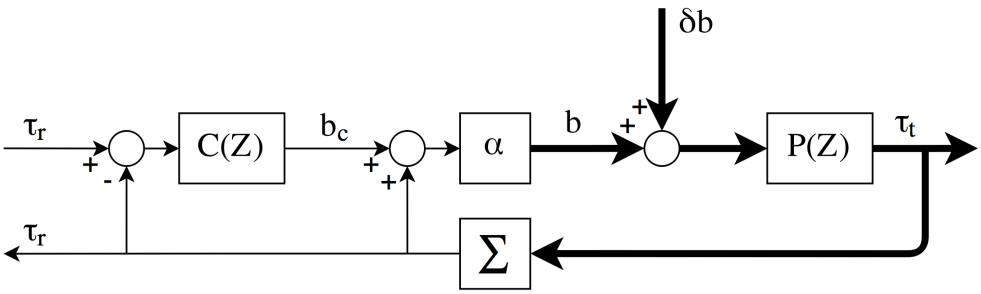

Figure 3.5: Control block diagram of PI-Multiburst scheduler

Given that the parameters w_1 and w_2 are chosen properly, it is desired to gaurantee the disturbance rejection by through the burst-correction term b_c it generates as the output. Therefore the relation for $C(Z)$ would be of the form below.

$$\begin{cases} b_c(k) = b_c(k-1) + w_1\epsilon_r(k-1) + w_2\epsilon_r(k-2) \\ \epsilon_r(k) = \tau_r^{\circ}(k) - \tau_r(k) \end{cases} \tag{3.7}$$

To derive suitable values for w_1 and w_2, one can consider transfer function $Q(Z)$ with b_c as input and τ_r as output to account for the task pool plus code responsible for measuring the burst times (the node α) and the total round duration (i.e. the summation node). This is all that is needed to have $C(Z)$ tuned so that ϵ_r will be asymptotically zero. Since the disturbance is embedded in the transfer function $Q(Z)$, it is necessary to understand the role of δb first. Consider the figure 3.6 which depicts the transfer function $Q(Z)$.

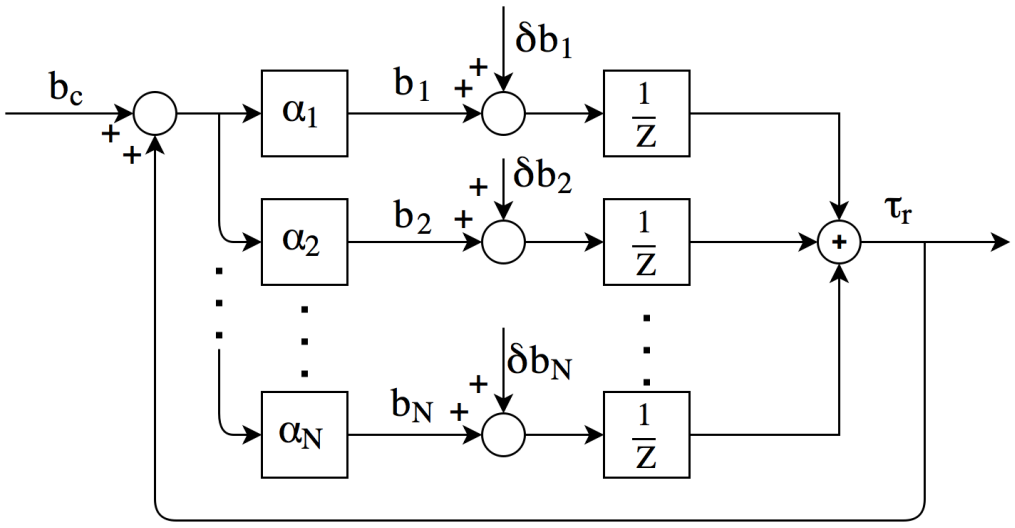

Figure 3.6: Expanded model of the task pool together with disturbances and CPU shares

Assuming a ficticious cut between the summation node and the output that is equivalent to considering the system in the open-loop situation, the closed-loop system would be as shown in figure 3.7 and is derived as following

$$\sum_{i=1}^{N} \frac{1}{Z}(\alpha_i(b_c + \tau_r) + \delta_b) = \tau_r$$

$$given \sum_{i=1}^{N} \alpha_i = 1, therefore,$$

$$\frac{1}{Z} b_c + \frac{1}{Z} \tau_r + \frac{1}{Z} \sum_{i=1}^{N} \delta b_i = \tau_r$$

in which the last term is the sum of disturbances (d). Hence

$$\tau_r = \frac{1}{Z-1}(b_c + d)$$

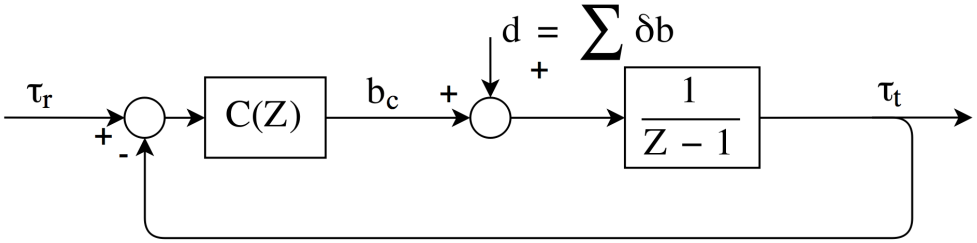

Figure 3.7: Simplifed model for tuning Multiburst scheduler's regulator

Requiring $\tau_r(Z)/B(Z) = (Z-1)/Z^2$, thus $\tau_r/\tau_r^\circ = (2Z-1)/Z^2$ and so the regulator's transfer function would be $C(Z) = 2Z - 1/Z^2$. Hence for disturbance rejection, $w_1 = 2$ and $w_2 = -1$. Given the aforementioned reasoning, the scheduling algorithm is explanable by the two listings below.

Algorithm 7: The Multiburst scheduling algorithm - Control model

Initialize the PI state variables;
for *each scheduling round* k **do**
  Read the measured $\tau_t(k-1)$ i.e. the CPU times used by the N_t tasks in the previous round;
  Compute $\tau_r(k-1) = \sum_{i=1}^{N_t} \tau_{t,i}(k-1)$ i.e. the duration of the previous round;
  Read $\tau_r^\circ(k-1)$ i.e. the required (set point) duration of the previous round;
  if *cardinality of the task pool or parameters have changed* **then**
    Reinitialize $b_i(k)$ to $\alpha_i \tau_r^\circ$;
  else
    Apply the PI regulator to obtain the burst correction:
    if *A thread has woken up during the round* **then**
      $b_c(k) = \frac{\sum_{t=1}^{N_t} \tau_t(k-1)}{\sum_{i \in ActiveThreads} \alpha_i} - \tau_r$;
    else
      $b_c(k) =$
      $b_c(k-1) + 2(\tau_r^\circ(k-1) - \tau_r(k-1)) - (\tau_r^\circ(k-2) - \tau_r(k-2))$;
    end
    for *each thread* i **do**
      if t *is active* **then**
        $b_i(k) = \alpha_t(b_c(k) + \tau_r(k-1))$;
      else
        $b_i(k) = 0$;
      end
    end
  end
  Activate the N_t tasks in sequence, preempting each of them when its burst is elapsed;
end

Algorithm 8: The multiburst scheduling algorithm - Multiburst actuation

if *task i is ready and was blocked when the round started* **then**
  Compute remainder of the round duration as $\tau_{rem} = \tau_r^\circ - \tau_r$;
  Assign a burst to i from the remainder: $b_i = \alpha_i \, \tau_{rem}$;
  // Normalize bursts allocated to active tasks
  $\tau_{tot} = \tau_{rem} + b_i$;
  for *each thread i s.t.* $T_i \in activeThreads$ **do**
    $b_i = (\tau_{rem}/\tau_{tot})(b_i - T_i) + T_i$;
  end
end

In contrast to I+PI which does not impose any particular order on execution of the tasks during each round, Multiburst applies three specific policies for task preemption and selection in the case of the wake-up events in order to balance responsiveness and number of context switches. As it is mentioned in section 3.1.2, the absence of multiple execution slots per thread in each round will inflict an upper bound for the round time so as not miss deadlines. Although the introduction of multiple bursts relieves this constraint, it may unleash the scheduler and result in much a higher number of context switches that is counter-productive. In this regards, an additional property (Multiburst priority) has been considered for the tasks in order to better realize the trade-off between responsiveness and context switches. This property determines what should happen upon re-activation of a thread as below.

- **Immediate preemption:** A preemption will occur as soon as the task wakes up and the control is transferred to the newly woken task. The next task to run would be the preempted one.

- **Preemption after current burst:** When a task of this priority wakes up, it will be marked for execution right after the end of the current running task.

- **Preemption at the of current round:** The newly woken task will

gain the CPU at the end of the round.

Overall the first policy provides a task with the highest degree of responsiveness at the cost of disturbing others and is more prone to rise context switch rate, thus, it should be considered only for tasks with tight deadlines. It also should be noted that since the inner loop has been removed and the newly woken task can not consume more than the burst allocated to it, the number of context switches will be limitted. The last two policies are less prompt to wake-up events but restrict the scheduler's intervention more than the immediate preemption case. Although this ordering method is very simple (by the use of a double-linked list with a pointer to the current running thread, the cost of re-ordering upon a wake-up event would be of order $O(1)$), it is effective enough to gain better performance with even fewer context switches than I+PI as it will be shown in the next chapter. Table 3.1 reports the worst case delay for tasks after they wake up according to the policy assigned to them.

Policy	**Worst case delay**
Immediate preemption	One preemption interrupt
Preemption after current burst	Maximum burst time
Preemption at end of the round	Round time

Table 3.1: Worst case delay after wake-up for Multiburst policies

Please note that in the case of multiple wakeups the newly running thread can be preempted again. To better understand the impact of these levels of Multiburst priority, we can consider a problematic case with I+PI in which Multiburst would outperform it with even fewer interventions. Consider a pool of four tasks equally sharing the CPU in which the third one executes a few instructions and sleeps for a specific time. It should acquire the CPU at most $1ms$ after it wakes up or it will miss the deadline. Figure 3.8 depicts the scenario and the way I+PI schedules the pool.

Given the $1ms$ timeout for the third task to regain the processor and the fact that there are five execution bursts in between, I+PI should be

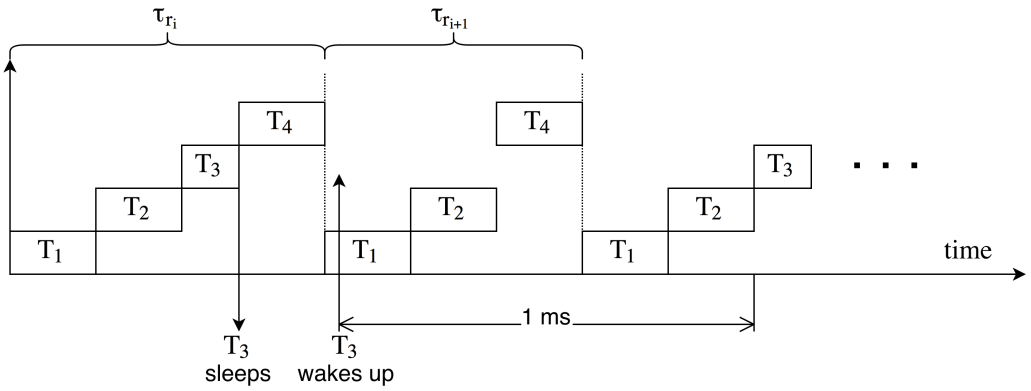

Figure 3.8: Problematic example - I+PI scheduling

configured with $\tau_r^\circ = 200$ microseconds to meet the deadline. Therefore, on average it would perform 5000 context switches per second. Instead, a Multiburst scheduling scheme adopting the "Preemption after current burst" policy for the third thread would execute the tasks in the order depicted by figure 3.9.

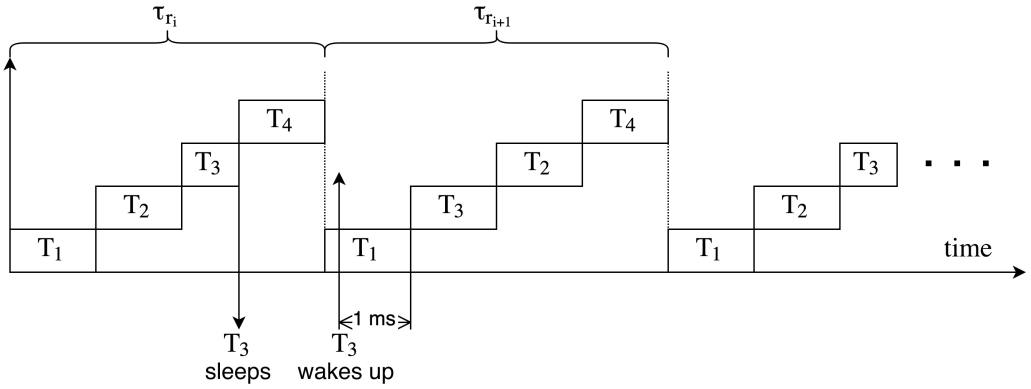

Figure 3.9: Problematic example - Multiburst scheduling

In this case, the deadline is met with $\tau_r^\circ = 1$ millisecond, hence, through 1000 context switches per second.

3.2.2 Scheduling in Miosix

Unlike many operating systems developed for both embedded devices and larger machines in which the scheduler is tightly coupled with the rest of the kernel, Miosix provides a well-designed uniform interface to support multiple schedulers that limits to a significant extent the implementation of our new scheduler to a single source file. It is worth mentioning that schedulers in Miosix can not coexist at run-time, hence, only one can be active at the same time. This, in fact, endows the developer with a consistent platform to implement and evaluate different scheduling algorithms isolated from one another and in an impartial manner. The multiplexing between available schedulers in Miosix is only possible at compile time by defining - in the global configuration header file - the C macro corresponding to the include-guard of the source file of the target scheduler which implements the interface. This is also desirable for the purpose of this work as it would not impose virtually any performance penalty on the scheduler.

A scheduler in Miosix is a C++ class that implements a specific interface, represented in figure 3.10. 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[2] , as well as to query existing threads[3]. 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”.

There is then the `IRQfindNextThread` function that low level interrupt handling code calls to ask the scheduler to perform a context switch. It is in

[2] `PKaddThread()` and `PKremoveDeadThreads()`

[3] `PKexists()`

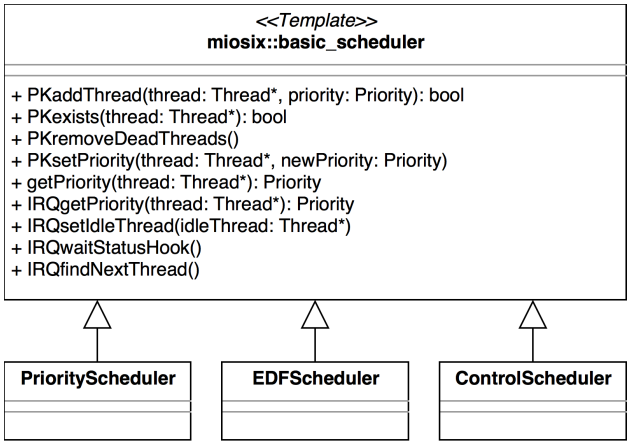

Figure 3.10: The scheduler interface in Miosix

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 is the `IRQwaitStatusHook` function that the kernel calls to inform the scheduler of certain events it may be interested into, like a thread blocking or unblocking which is particularly useful in the implementation of Multiburst scheduler as it is the main point where the `activeThreads` list that is required in algorithm 8 can be managed.

3.2.3 Implementation and integration into Miosix

Integration of the Multiburst scheduler with the Miosix kernel consists of implementing the algorithms7 and 8 contained in a C++ class according to the interface depicted in figure 3.10. Among the various functions introduced by the interface, the source code of `IRQfindNextThread` and `IRQwaitStatusHook` are presented here since as for others the content is quite the same as that of I+PI[19].

There are essentially two groups of variables required by the scheduler among which the relevant ones are represented by listing below. The first group consists of external variables defined in the kernel that helps the sched-

uler to co-operate with the kernel and the second group contains internal variables needed to implement the Multiburst algorithm.

Listing 3.1: Multiburst scheduler implementation - Data members

```
// 1) External - These are defined in kernel.cpp
// Pointer to current running thread
extern volatile Thread *cur;
// Kernel Lock Flag
extern unsigned char kernel_running;
// A double linked list of sleeping threads
extern IntrusiveList<SleepData> *sleepingList;

// 2) Internal
// The tickless system timer
static ContextSwitchTimer& timer = ContextSwitchTimer::instance
    ();
// List of active threads
static IntrusiveList<ThreadsListItem> activeThreads;
// Pointer to the current running active thread
static IntrusiveList<ThreadsListItem>::iterator curInRound =
    activeThreads.end();
// Flags for IRQfindNextThread
static bool dontAdvanceCurInRound = false;
static bool threadWokenInBurst = false;
// Flag for regulator's code
static bool threadWokenInRound = false;
// List of all the threads
Thread *ControlScheduler::threadList=0;
unsigned int ControlScheduler::threadListSize=0;
// A pointer to the IDLE thread
Thread *ControlScheduler::idle=0;
// Multiburst controller's variables/measurements
long long nextPreemption = LONG_LONG_MAX;
long long burstStart = 0;
int ControlScheduler::SP_Tr=0;
int ControlScheduler::Tr=bNominal;
int ControlScheduler::bco=0;
int ControlScheduler::eTro=0;
```

```
bool ControlScheduler::reinitRegulator=false;
```

The next listing shows the C++ code for finding the next thread to run. `IRQfindNextThread` first performs the algorithm 8 by re-assigning the remaining time of the current round to the thread that has just woken up during the round (and was not assigned a burst before) and normalizing the bursts and then looks for the next thread to run by selecting form the `activeThreads` list which contains only the threads that are ready for execution during the current round. Note that this function also executes the Multiburst controller (algorithm 7) when it detects the end of the round which is equivalent to reaching the end of the `activeThreads` list. Functions `IRQsetNextPreemption*` rely on the new tickless time interface to set the next preemption point while taking care of the next wake up time for sleeping threads.

Listing 3.2: Multiburst scheduler implementation - IRQfindNextThread

```
void IRQsetNextPreemptionForIdle();
void IRQsetNextPreemption();
unsigned int ControlScheduler::IRQfindNextThread()
{
    //If kernel is paused, do nothing
    if(kernel_running!=0) return 0;

    if(cur!=idle)
    {
        //Not preempting from the idle thread, compute actual
        //burst time used by the preempted thread
        int Tp = static_cast<int>(timer.IRQgetCurrentTime() -
            burstStart);
        cur->schedData.Tp+=Tp;
        // Update the actual round-time
        Tr+=Tp;

        if (threadWokenInBurst)
        {
            threadWokenInBurst = false;
```

```
            // Burst Correction - re-assign the remainder
            // of round-time to all ready threads including
            // the one that has just woke up (t)
            int remTr = SP_Tr - Tr;
            int tot = remTr;
            // Assign burst to woken threads from remainder
            // of the round
            for(Thread *it=threadList;it!=0;it=it->schedData.
                next)
            {
                if (it->schedData.wokenDuringBurst)
                {
                    it->schedData.wokenDuringBurst = false;
                    it->schedData.bo = it->schedData.alfa *
                        remTr;
                    tot += it->schedData.bo;
                }
            }
            // Normalize bursts
            float nrmFact = (float)remTr / tot;
            for(Thread *it=threadList;it!=0;it=it->schedData.
                next)
            {
                it->schedData.bo =
                  (it->schedData.bo - it->schedData.Tp) *
                  nrmFact + it->schedData.Tp;
            }
        }
    }

    //Find next thread to run
    for(;;)
    {
        if(curInRound!=activeThreads.end()){
            if (dontAdvanceCurInRound)
                dontAdvanceCurInRound = false;
            else
                curInRound++;
```

```
        }
        if(curInRound==activeThreads.end())
        {
            if(activeThreads.empty())
            {
                //No thread is ready, run the idle thread

                curInRound=activeThreads.end();
                cur=idle;
                ctxsave=cur->ctxsave;
                IRQsetNextPreemptionForIdle();
                return 0;
            }

            //End of round reached, run scheduling algorithm
            curInRound = activeThreads.front();
            IRQrunRegulator();
        }

        if((*curInRound)->t->flags.isReady() && (*curInRound)->t
            ->schedData.bo>0)
        {
            //Found a READY thread, so run this one
            cur=(*curInRound)->t;
            ctxsave=cur->ctxsave;
            IRQsetNextPreemption(cur->schedData.bo);
            return 0;
        } else {
            //The thread has no remaining burst time
            //so just ignore it
            curInRound++;
        }
    }
}
// Should be called when the current thread is the idle thread
static inline void IRQsetNextPreemptionForIdle()
{
    if (sleepingList->empty())
```

```
        // normally should not happen unless an IRQ is
        // already set and able to preempt idle thread
        nextPreemption = LONG_LONG_MAX;
    else
        nextPreemption = sleepingList->front()->wakeup_time;
    timer.IRQsetNextInterrupt(nextPreemption);
}

// Should be called for threads other than idle thread
static inline void IRQsetNextPreemption(long long burst)
{
    long long firstWakeupInList;
    if (sleepingList->empty())
        firstWakeupInList = LONG_LONG_MAX;
    else
        firstWakeupInList = sleepingList->front()->wakeup_time;
    burstStart = timer.IRQgetCurrentTime();
    nextPreemption = min(firstWakeupInList, burstStart + burst);
    timer.IRQsetNextInterrupt(nextPreemption);
}
```

The following C++ code is equivalent to Multiburst's controller which computes the burst correction term and assigns CPU time to the threads for the next round.

Listing 3.3: Multiburst scheduler implementation - Controller's code

```
void ControlScheduler::IRQrunRegulator()
{
    if(reinitRegulator)
    {
        reinitRegulator=false;
        //Reset state of the external regulator
        eTro=0;
        bco=0;
        //Recalculate per thread set point
        for(Thread *it=threadList;it!=0;it=it->schedData.next)
        {
            it->schedData.SP_Tp=static_cast<int>(it->schedData.
```

```
            alfa*SP_Tr);
        // Apply Saturations
        it->schedData.bo=min(max(it->schedData.SP_Tp,bMin),
            bMax);
        // Reset burst consumption measurements
        it->schedData.Tp = 0;
    }
} else {
    int eTr=SP_Tr-Tr;
    if (threadWokenInRound){
        int sTp = 0;
        float sAlfa = 0;
        for(Thread *it=threadList;it!=0;it=it->schedData.
            next)
        {
            sTp += it->schedData.Tp;
            if (it->flags.isReady())
                sAlfa += it->schedData.alfa;
        }
        bco = sTp/sAlfa - Tr;
    }else
        bco = bco + 2 * eTr - eTro;

    bco=min<int>(max(bco,-Tr),bMax*threadListSize);
    eTro=eTr;
    float nextRoundTime=static_cast<float>(Tr+bco);

    //Recalculate per thread bursts
    for(Thread *it=threadList;it!=0;it=it->schedData.next)
    {
        if (it->flags.isReady())
        {
            it->schedData.SP_Tp=static_cast<int>(it->
                schedData.alfa*nextRoundTime);
            //saturation
            it->schedData.bo=min(max(it->schedData.SP_Tp,
                bMin),bMax);
        } else {
```

```
                it->schedData.SP_Tp = 0;
                it->schedData.bo = 0;
            }
            // Reset burst consumption measurements
            it->schedData.Tp = 0;
        }
    }
    // Reset measurements
    threadWokenInRound = false;
    Tr=0;//Reset round time
}
```

The last listing shows the implementation of the `IRQwaitStatusHook` function which keeps the `activeThreads` list updated and ordered according to the three policies mentioned in section 3.2.1. Principally if a thread sleeps, it should be removed, while if a thread wakes up, it should be added back to the list according to the policy specified for the thread. It is worth mentioning that since the pointer to the current item in this list may be altered both by `IRQwaitStatusHook` and `IRQfindNextThread`, the variable `dontAdvanceCurInRound` has been used to synchronize to the two. Finally, this function should also signal the controller if there is a thread newly woken up thread in the round since the so that the burst correction term can be computed correctly.

Listing 3.4: Multiburst scheduler implementation - IRQwaitStatusHook

```
void ControlScheduler::IRQwaitStatusHook(Thread* t)
{
    // Managing activeThreads list
    if (t->flags.isReady()){
        // The thread has became active -> put it in the list
        addThreadToActiveList(&t->schedData.atlEntry);
        t->schedData.wokenDuringBurst = true;
        threadWokenInBurst = true; // Signal for
            IRQfindNextThread
        threadWokenInRound = true; // Signal for IRQrunRegulator
    }else {
```

```
        // The thread is no longer active -> remove it from the
            list
        remThreadfromActiveList(&t->schedData.atlEntry);
    }
}

static inline void addThreadToActiveList(ThreadsListItem *
    atlEntry)
{
    switch (atlEntry->t->getPriority().getRealtime()){
        case REALTIME_PRIORITY_IMMEDIATE:
        {
            //In this case we should insert the woken thread
            //before the current item and put the pointer
            //to the item behind it so that IRQfindNextThread
            //executes the woken thread and then come back
            //to this thread which is about to be preempted
            auto tmp = curInRound; tmp--;
            if (tmp==activeThreads.end()){
                //curInRound is the first Item and doing
                //curInRound-- twice raises an exceptional
                //behavior in which puts the pointer in the
                //end of the list and IRQfindNextThread
                //detects anmEndOfRound situation by mistake
                activeThreads.insert(curInRound,atlEntry);
                curInRound--;
                dontAdvanceCurInRound = true;
            }else{
                activeThreads.insert(curInRound,atlEntry);
                curInRound--;curInRound--;
            }
            //A preemption would occur right after this function
            break;
        }
        case REALTIME_PRIORITY_NEXT_BURST:
        {
            auto temp=curInRound;
            activeThreads.insert(++temp,atlEntry);
```

```
            //No preemption should occur after this function
            break;
        }
        default: //REALTIME_PRIORITY_END_OF_ROUND
        {
            activeThreads.push_back(atlEntry);
            //No preemption should occur after this function
        }
    }
}

static inline void remThreadfromActiveList(ThreadsListItem *
    atlEntry)
{
    //If the current thread in run has yielded and
    //caused a call to this function the curInRound
    //pointer must advance in the list so as not to
    //lose the track of the list and then we can
    //delete the item
    if (*curInRound==atlEntry){
        curInRound++;
        //Since we are sure that IRQfindNextThread
        //will be called afterwards, it should be
        //prevented to advance curInRound pointer
        //again, otherwise it will skip 1 thread's burst
        dontAdvanceCurInRound = true;
    }
    activeThreads.erase(IntrusiveList<ThreadsListItem>::iterator
        (atlEntry));
}
```

Chapter 4

Evaluation and Benchmarking

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 that are based on algorithmic complexity (such as [5]), albeit being appreciable, 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 work by Maggio et al. [16] shows, lower computational complexity of a scheduling algorithm with respect to another one does not necessary imply that it is able to schedule a given task pool better even when considering the impact of scheduling overhead. 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. This chapter is divided into three section in which the first presents the behaviour of proposed algorithm through simulations so as to demonstrate its conduct of scheduling is as expected. The two last parts shows how it performs with respect to other schedulers on some benchmarks relevant to real-time systems.

4.1 Simulation results

It is a common practice to verify a control scheme through a simulation process that provides the required signals and monitors the behavior of the designed control system. Thus, in order to demonstrate fulfillment of the goals set for the Multiburst scheduler discussed in the previous chapter and in particular its responsiveness regarding periodic tasks, it has been simulated over a pool of three threads and the results are reported in this section. The pool consists of two batch tasks (threads no. 1 and 2) and one periodic task (thread no. 3) in which the latter is active for 10ms and then sleeps for 20ms and so forth. The simulation lasts for 100 rounds with a nominal burst of 1ms for each thread, hence, ideally, each round should take 3ms. All the threads equally share the processor up to the 50th round. Afterward, the shares would change to 20%, 40%, and 40% respectively for threads 1, 2, and 3.

As reported by figure 4.1, once the periodic thread sleeps, a burst correction term is computed by the regulator to compensate for the absence of the sleeping thread, hence, others will start to consume more CPU time and the round time will follow the set point of 3ms.

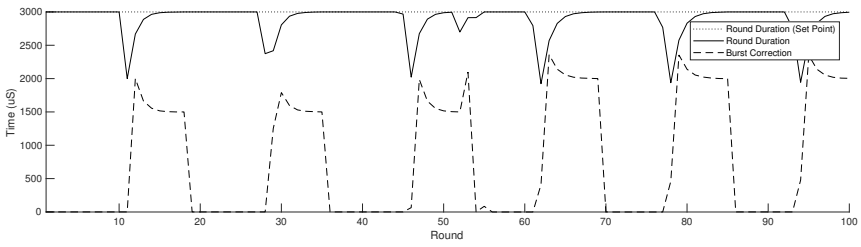

Figure 4.1: Simulation - Round duration and burst correction

Therefore, as desired, the round duration is the main knob to ensure responsiveness. But, in order to better understand what happens if the periodic task wakes up, one should take closer look to the burst usage and set points of the tasks individually. Figure 4.2 reports both quantities for all the three threads.

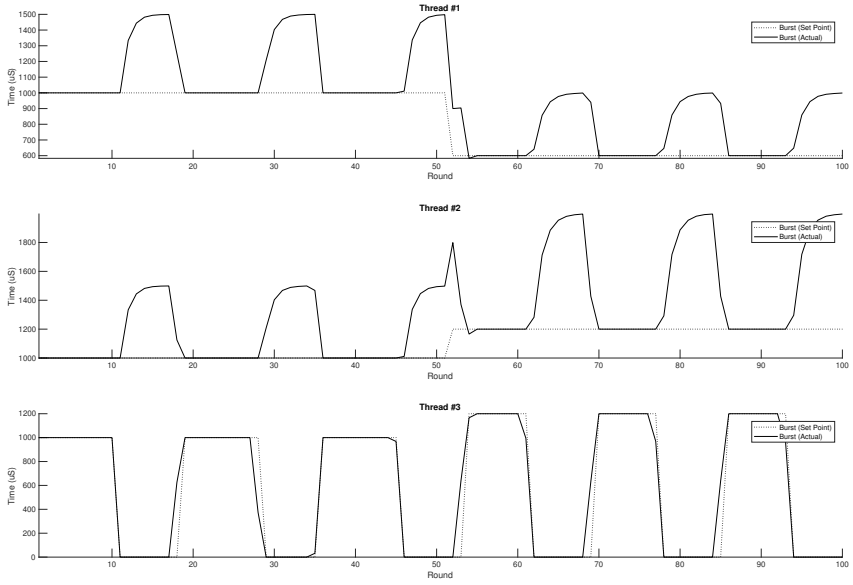

Figure 4.2: Simulation - Tasks' actual and desired bursts

Considering the third task, it is evident that it starts to get CPU as soon as it is activated again and others which has been using extra CPU with respect to their specified share, release the CPU immediately and will follow their own designated burst. For example, the periodic thread wakes up in the middle of 18th round and takes its share from remaining time of the round and as soon as the 19th round begins, it will have 1/3 of the round as expected. On the other hand, at the end of the 50th round in which CPU shares change, the first two threads that are active will immediately experience the change in the following round as well as the last thread that gains according to its new share as soon as it wakes up.

4.2 Hartstone benchmark

The Hartstone benchmark [28] is here used to evaluate the performance of Multiburst, I+PI, EDF and RR (round robin) schedulers developed and in-

tegrated to Miosix tickless kernel. All tests are carried out on a `STM32f407vg discovery` embedded system which is endowed with `STM32F4 (CortexM4)` system-on-chip configured in the same way for all the schedulers.

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 4.1.

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 4.1: The Hartstone [28] baseline task pool

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 which will be described

along with the results of various schedulers. In each benchmark, the number of iterations [1] till the first deadline is missed for the four 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.

4.2.1 Benchmark one - asymmetric pool

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 and also the extent to which the scheduler can handle asymmetric task pools. As it is reported by

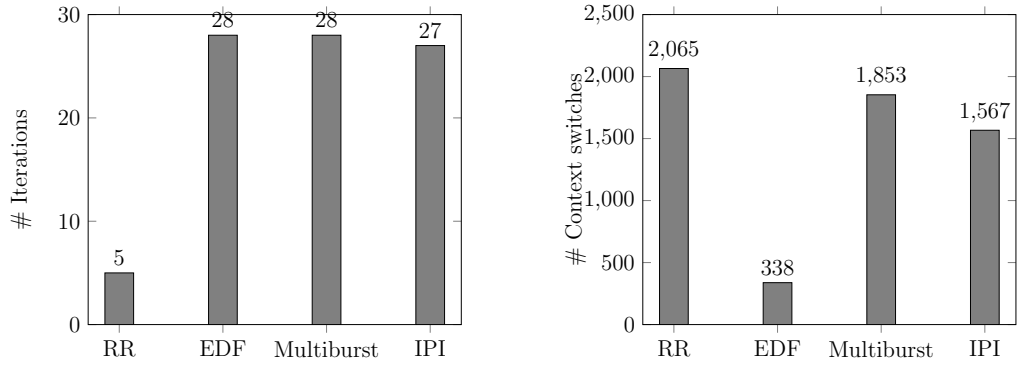

Figure 4.3: Hartstone Benchmark - Test 1

figure 4.3, the Multiburst and EDF exhibit the most stress resistance and they are followed by I+PI and RR respectively. Note that this test is the most extreme case, where the period of one tasks differs so much with respect to the other and that is one of the motivations supporting the introduction of Multiburst scheduler in this thesis. In fact, Multiburst is designed with the aim of introducing a configurable trade-off between being responsive and performing fewer context switches which is properly demonstrated by this

[1] In this context, an iteration is referred to a window of time in which the characteristics of the task pool (i.e. number of the tasks, their frequencies and workloads) remain constant.

test in the sense that it also outperforms EDF. Although I+PI reached the same number of iterations as EDF, but as it will be demonstrated in section 4.3, it can not tolerate the situation dictated by this pool very long and will miss many deadlines as soon as the first one is missed.

4.2.2 Benchmark two - balanced workloads

The second benchmark uniformly scales the frequencies of all tasks by 1.1, 1.2, and so forth till the first miss, and has the property of maintaining a balanced workload between the tasks. In this case, the RR scheduler handles

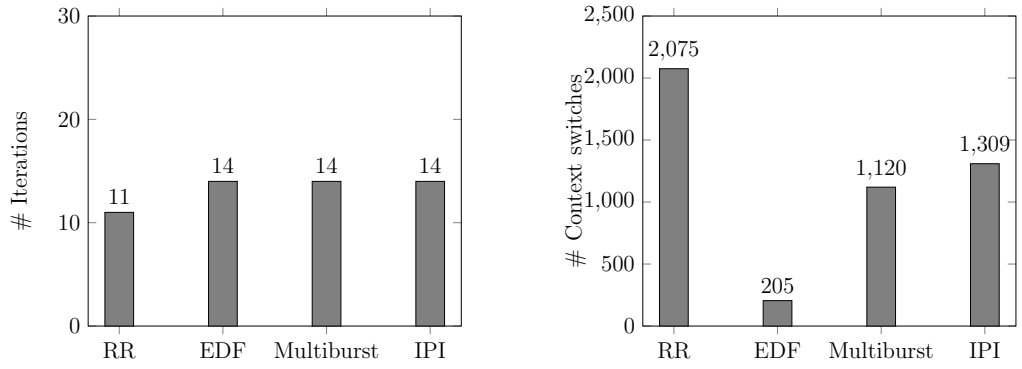

Figure 4.4: Hartstone Benchmark - Test 2

the tasks significantly better than that of the first test in the sense that the number of iterations without a deadline miss is closer to the other three that perform similarly. This is due to the fact that the pool is balanced in workloads and thus, matches better the fixed quantum applied by the RR Scheduler. Multiburst spends fewer context switches compared to that of I+PI but achieves the same performance of EDF. See figure 4.4.

4.2.3 Benchmark three - unbalanced workload growth

In addition to the first benchmark which evaluates the performance on that is unbalanced in the first place, it is in general important and also relevant to the purpose of this work that the scheduler be capable of properly managing

asymmetric workload growth leading to a totally unbalanced set of workloads. The third benchmark increases the workload of all tasks by 1 KiloWhet at a time resulting in an unbalanced workload increase. Again, EDF reaches the

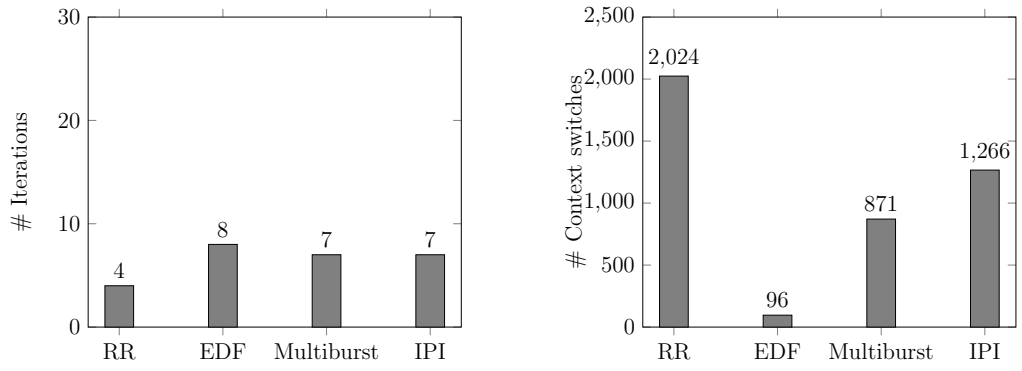

Figure 4.5: Hartstone Benchmark - Test 3

best performance while having the minimum number of context switches. The Multiburst and I+PI schedulers are very close to EDF through their variable burst sizes and more considerably more context switches. Using looser round duration, Multiburst spends fewer context switches than I+PI to reach the same performance.

4.2.4 Benchmark four - task pool growth

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. In this case, only Multiburst reaches EDF with fewer context switches compared to two other schedulers.

In conclusion, the RR scheduler shows the worst performance due to adopting fixed-size quantum (i.e. burst) and having the same amount of context switches regardless of characteristics of the pool at runtime in all the cases. This is a good example that demonstrates scheduling algorithms with lower complexity orders and thus with cheaper context switches does not necessarily produce better results in practice. The two control schedulers,

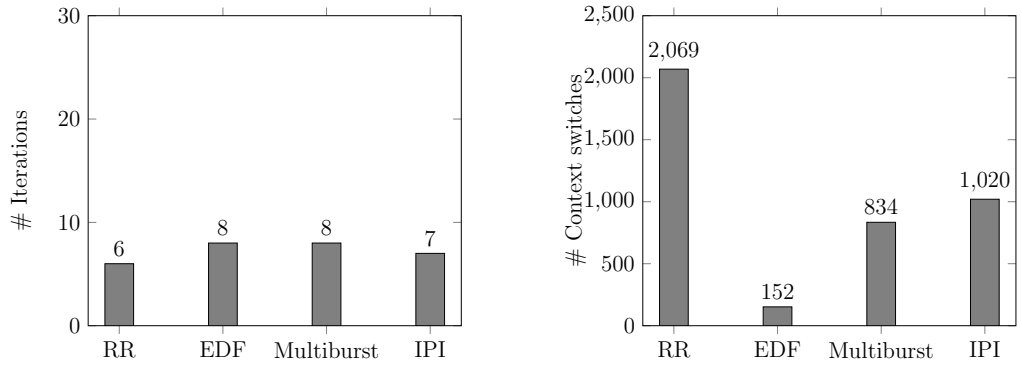

Figure 4.6: Hartstone Benchmark - Test 4

on the contrary, support adaptive bursts and amount of context switches by design which is completely evident from the experiments.

4.3 Extended Hartstone benchmark

As the literature shows and the results of Hartstone benchmark suggest, the EDF scheduler is the optimal solution when considering hard real-time problems, however, it suffers from domino effect as discussed in chapter 1. To show the ability of control-based approaches (I+PI and Multiburst) to handle the scenario where EDF is not the best candidate - i.e. real-time systems where deadline misses are not totally fatal but fewer failures are better appreciated - Hartstone has been extended to address the situation where the required processor utilization transiently exceeds unity. This is a typical situation that can happen in soft real-time systems where, for cost reasons, the CPU is not scaled to handle the maximum predicted workload, but rather it is dimensioned in between the average and maximum workload.

This extended Hartstone benchmark consists of running the system for a total duration of 120s as follows:

- In the first 30 seconds the workload is kept constant at 48% CPU utilization.
- During $t = [30, 45]$ the workload is increased to 120% and kept constant.

- After $t = 45$ seconds, the workload is decreased again to 48% until 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) along with the number of context switches per second to appreciate the ability of the schedulers to avoid unnecessary context switches.

Figure 4.7 displays the number of deadlines missed and figure 4.8 reports the number of context switches performed per second by each scheduler.

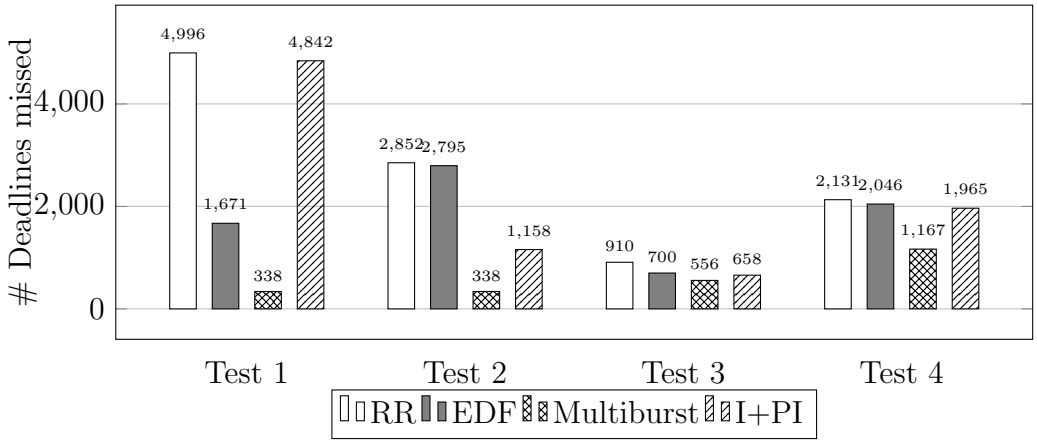

Figure 4.7: Extended Hartstone - Deadline miss

The RR scheduler, as in the classic Hartstone, despite making significantly more context switches than the other two schedulers, performs worse in all four benchmarks. I+PI outperforms EDF in all four tests except the first one with the asymmetric pool, at the price of a minimal increase in the number of context switches with respect to that of EDF. While on the other hand, Multiburst spends slightly more number of context switches than I+PI, it misses the least number of deadlines compared to all the other sched-

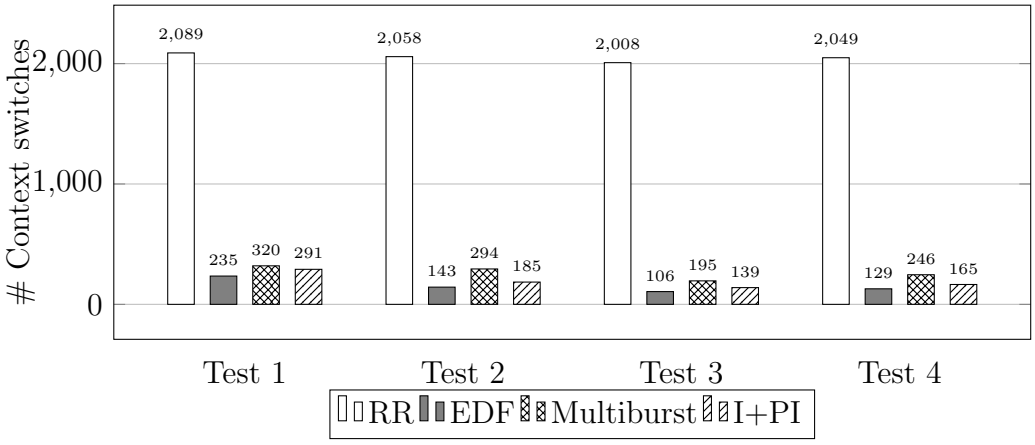

Figure 4.8: Extended Hartstone - Context switches

ulers and for all the tests. In fact, Multiburst is configured with the same round duration and task priorities (CPU shares) as I+PI but, as it reacts properly on periodic tasks with allowing them to have their burst partitioned into multiple ones during each round, it is better adjusting itself with the variations in the task pool parameters.

Moreover, it is noteworthy to remark that both control-based schedulers 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.

In conclusion, the results from both Hartstone and its extended version demonstrate that both control-based schedulers can perform truely close to the optimal solution and even better in some cases which makes them suitable candidates for both soft and hard real-time applications. Endowed with the amendments discussed so far, Multiburst allows better response times and adaptivity than its predecessor especially in the case of asymmetric pools.

Chapter 5

Conclusions

As a pursuit of the research trend wherein models rooted in control theory are exploited to enrich the design phase in the development of operating system components, this thesis provided a robust solution to the scheduling problem for both soft and hard real-time applications. The concept of tickless kernels and their advantages, particularly with respect to embedded and real-time systems, was introduced in this thesis. Then, an effective and simple mechanism for having a kernel operate in the tickless mode while being loosely coupled with the target hardware was designed and implemented on a real embedded kernel named Miosix.

Moreover, a control-based scheduler which relies on a tickless kernel was designed to improve the responsiveness of an existing scheduler named I+PI through redesigning with a different view of fairness in the whole problem. The responsiveness deficiency of I+PI was dealt in the new scheduler by reconsidering the actual definition of fairness used in I+PI and changing the way the control model is augmented with heuristic algorithms around it and consequently the results demonstrated better performance. In this regard one can deduct that

- Although one may argue that the use of theoretical approaches are there to avoid using heuristic algorithms which are highly error prone and difficult to analyze formally, having good heuristics alongside with

the models which confine them is effective and thus there should be a trade-off between these two elements when designing a system.

- When resorting to model a problem with control theory, one should make sure that there is a precise definition of elements (objectives, constraints and etc.) involved in the problem.

In fact, the inner loop of I+PI is there to guarantee fairness based on the false assumption that the tasks are always thirsty to get the CPU and did not distinguish between voluntarily and forced relinquishing of CPU. Then to alleviate the impact of the inner loop on responsiveness, the heuristic part of I+PI is fighting the inner loop back. Therfore, although the research work that introduced I+PI was a great and peculiar leap towards a new perspective in system design, this work completed it by considering these two major points and resulted in a more efficient algorithm.

5.1 Future works

The design pattern introduced in this work and related ones exploiting control theory to enhance OS design can be further expanded into many more problems in this field such as load balancing in multi/many-cores and principally anywhere that some shared resources have to be distributed according to particular criteria.

The new perspective of formally modeling and assessing the problems in operating systems may have opened the door to reconsider the qualitative definitions (such as that of fairness) and transform them into more precise and careful quantitative definitions. Thus, it is worthwhile trying to model different problems and performing examinations about how the current definitions will affect the performance of models in well-endorsed benchmarks.

Bibliography

[1] Akli Abbas, Malik Loudini, Emmanuel Grolleau, Driss Mehdi, and Walid-Khaled Hidouci. A real-time feedback scheduler for environmental energy with discrete voltage/frequency modes. *Comput. Stand. Interfaces*, 44(C):264–273, February 2016.

[2] Özgür Armağan and Leyla Gören-Sümer. Feedback control for multi-resource usage of virtualised database server. *Comput. Electr. Eng.*, 40(5):1683–1702, July 2014.

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

[4] Sanjoy Baruah and Joël Goossens. Scheduling real-time tasks: Algorithms and complexity, 2003.

[5] Vincenzo Bonifaci, Ho-Leung Chan, Alberto Marchetti-Spaccamela, and Nicole Megow. Algorithms and complexity for periodic real-time scheduling. *ACM Trans. Algorithms*, 9(1):6:1–6:19, 2012.

[6] Daniel Bovet and Marco Cesat. *Understan1ding the Linux Kernel*. O'Reilly, third edition, 2005.

[7] Miosix embedded operating system. http://miosix.org/.

[8] S. Siddha et al. Getting maximum mileage out of tickless. *Linux Symposium*, jun 2007.

[9] V. Srinivasan et al. Energy-aware task and interrupt management in linux. *Linux Symposium 2*, aug 2008.

[10] Gene F. Franklin, David J. Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Pearson, 6th edition, 2010.

[11] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Ottawa Linux Symposium*, 2006.

[12] S. Haldar and D. K. Subramanian. Fairness in processor scheduling in time sharing systems. *SIGOPS Oper. Syst. Rev.*, 25(1):4–18, January 1991.

[13] Victor Jiménez, Francisco J. Cazorla, Roberto Gioiosa, Mateo Valero, Carlos Boneti, Eren Kursun, Chen-Yong Cher, Canturk Isci, Alper Buyuktosunoglu, and Pradip Bose. Power and thermal characterization of power6 system. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 7–18, New York, NY, USA, 2010. ACM.

[14] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1:1–1:16, New York, NY, USA, 2016. ACM.

[15] Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms*. *Real-Time Systems*, 23(1):85–126, 2002.

[16] Martina Maggio, Federico Terraneo, and Alberto Leva. Task scheduling: A control-theoretical viewpoint for a general and flexible solution. *ACM Trans. Embed. Comput. Syst.*, 13(4):76:1–76:22, March 2014.

[17] Quest operating system. http://quest.bu.edu/.

[18] V. Pallipadi. Cpuidle - do nothing, efficiently... *Linux Symposium*, jun 2007.

[19] The I+PI scheduler. https://github.com/fedetft/miosix-kernel/tree/master/miosix/kernel/scheduler/control.

[20] Silicon Labs. *EFM32GG Reference Manual*, April 2016.

[21] STMicroelectronics. *Reference manual - STM32F205xx, STM32F207xx, STM32F215xx and STM32F217xx advanced ARM-based 32-bit MCUs*, February 2015.

[22] STMicroelectronics. *Reference manual - STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM-based 32-bit MCUs*, July 2015.

[23] STMicroelectronics. *Reference manual - STM32F100xx advanced ARM-based 32-bit MCUs*, June 2016.

[24] The Barrelfish Operating System. http://www.barrelfish.org/.

[25] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[26] Federico Terraneo. Control based design of os components. Master's thesis, Politecnico Di Milano, 2011.

[27] Linus Torvalds. Summary of changes from v2.6.6-rc2 to v2.6.6-rc3. Technical report, The Linux Fundation, April 2004.

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

[29] Feng Xia, Youxian Sun, and Yu-Chu Tian. Feedback scheduling of priority-driven control networks. *Comput. Stand. Interfaces*, 31(3):539–547, March 2009.