Identification and control techniques for multicore scheduler design

Tesi di:
Matteo Carini
matr. 750193

Relatore: Prof. Alberto Leva
Correlatore: Ing. Federico Terraneo

Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Anno accademico 2013 - 2014

Desidero dedicare questo lavoro di tesi alla mia mamma Giuseppina.
Vedere coronato il mio percorso di studi è sempre stato un suo grande desiderio,
spero che l'impegno per realizzare questo lavoro possa, almeno in parte,
compensare l'affetto che non sono riuscito a darle.

Mi mancherà sempre e la porterò sempre nel mio cuore.

Ringraziamenti

Desidero ringraziare Alberto Leva e Federico Terraneo per tutte le idee, la disponibilità e il supporto dedicato alla realizzazione di questa tesi;
Questo documento è stato realizzato e impaginato attraverso LaTeX su un sistema GNU/Linux. Gli algoritmi realizzati in questo elaborato saranno prossimamente rilasciati gratuitamente con licenza open source.

Contents

List of Figures

List of Tables

Abstract

This thesis is part of a long term research line whose purpose is to apply the principles of the systems and control theory to the *design* of software systems [14]. This research is motivated by the fact that a significant class of computing system problems found in computing systems at large and, more in detail, within operating systems, are *de facto* control problems, even though, traditionally, are solved directly through algorithms or heuristics.

A relevant and prominent example is the problem of task scheduling in operating systems, for which a fully control-based scheduler was designed, implemented and profiled [17, 15]. It was shown that such a design approach results in a simpler implementation with respect to many classical ones, and permits to analyze and assess the results formally.

This thesis extends the quoted work in a previously unexplored direction, a control-based approach to schedulers designed for multi core architectures. To use control-design principles in a multicore context, first of all, it's necessary to develop the *model* of the process which will be controlled. So, first part of the thesis is focused on a preliminary analysis about how to model tasks in multi core context. Tasks have been divided into classes, characterized by similar task behaviors from system resources point of view, and then each class has been modeled.

Since there are no physical principles available to write models, they have been obtained from data, profiling, for each class, a representative application. Models have to well represent all the tasks belonging to the same class, so they have been developed to represent a family of similar, but different, applications, united by some basic behaviors.

In the second part of the work, it's presented a multi core scheduler simulator development. This simulator is useful to have a plain, controlled environment where control structures can be implemented, tested and tuned, without external influences.

Sommario

Questa tesi fa parte di una linea di ricerca di lungo periodo il cui fine è l'applicazione dei principi della teoria dei sistemi e del controllo alla *progettazione* dei sistemi software [14]. La ricerca è motivata dal fatto che una vasta gamma di problemi che si incontrano nell'ambito dei sistemi software, e più in particolare, dei sistemi operativi sono in relatà dei problemi di controllo, anche se tradizionalmente risolti direttamente tramite algoritmi o euristiche.

Tra questi problemi, un esempio rilevante è quello dello scheduling nei sistemi operativi, per il quale una soluzione interamente basata sulla teoria del controllo è stata progettata, implementata e comparata con soluzioni più tradizionali [17, 15], dimostrando che un tale approccio progettuale dà come risultato una maggiore semplicità rispetto a molti algoritmi classici e, soprattutto, permette di analizzare e valutare i risultati ottenuti formalmente.

In questo lavoro si è proposto di estendere l'approccio basato sulla teoria del controllo a scheduler progettati per architetture multi core. Per utilizzare i principi di progettazione proposti della teoria del controllo in un contesto multicore, prima di tutto è necessario sviluppare il *modello* del processo che sarà controllato. Così, la prima parte della tesi è focalizzata su un' analisi preliminare su come modellare i task in un contesto con più unità di calcolo indipendenti, ciascuna dotata di una cache locale. I task sono stati suddivisi in classi, caratterizzate da comportamenti (dei task) simili dal punto di vista delle risorse di sistema e, successivamente, ogni classe è stato modellata. Le classi individuate sono distinte sulla base del tipo di workload (periodico, non periodico, che necessita di interazione con l'utente o con altri risorse, ecc), sulla base del tipo di interazione con l'utente e, infine, sul grado di reattività richiesto al sistema, individuando complessivamente, quattro classi:

- task con deadline periodiche (o task periodici)
- task con una sola deadline e scarsa necessità di interazione con l'utente (o task batch)
- task senza deadline ma interattivi, che richiedono una buona reattività del sistema (task priority based)
- task attivati dal verificarsi di un evento (task ebent based)

Per ciascuna di queste classi è stata poi considerato il modo in cui i task possono interagire con il sistema (task I/O bound, cpu bound o memory bound), importante per sviluppare uno scheduler che possa tener conto dell'impatto dei task (e delle eventuali migrazioni) a livello di cache e di utilizzo di cpu o di altre risorse (come il disco, le periferiche ecc.).

Poiché non ci sono principi fisici su cui basarsi per scrivere i modelli, essi sono stati ottenuti a partire dai dati, profilando, per ciascuna classe, l'applicazione più rappresentativa (alcune sono applicazioni di uso comune, altre sono state scritte ad-hoc).

I modelli devono rappresentare bene tutti i task che appartengono alla stessa classe, quindi non sono stati sviluppati per fare un buon fitting sulle tracce di esecuzione di uno specifico task, ma devono rappresentare una famiglia di diverse tracce di esecuzione, accomunate da alcuni comportamenti fondamentali (i.e da un certo modo di utilizzare la cache o la cpu, dal grado di reattività richiesto al sistema, ecc..). Per ottenere questo, è stato scelto di scrivere modelli che fossero strutturati come dei filtri formatori. In questo modo, il modello può essere analizzato (e validato) anche nel dominio delle frequenze, presentando, nel contempo, una struttura semplice e facile da mettere a punto.

Nella seconda parte dell'elaborato, è presentato lo sviluppo di un simulatore in grado di testare algoritmi di scheduling per sistemi multi core. Il simulatore è utile perchè mette a disposizione un ambiente controllato dove le strutture di controllo possono essere implementate, testate e messe a punto, senza l'influenza di scheduler già esistenti o di altri moduli del sistema operativo. Inoltre, il simulatore è stato progettato per implementare e simulare facilmente sia scheduler single core che scheduler multi core, e per fornire simulazioni in breve tempo, realizzando così un framework flessibile e facile da usare.

Part I.

Principles of control based scheduling

This part of the thesis introduces the scheduling problem, evidences its relevance in the context of computing systems, and discusses some classical solutions proposed in literature. After a brief introduction, the discussion concentrates on explaining why control-based schedulers are a particularly effective way to approach the evidenced problems. Subsequently, moving the focus to the specific subject of this work, the chapter addresses possible ways to exploit control-theoretical principles to extend the scheduling solutions proposed for the single core case, to a multicore situation.

1. Introduction

1.1. The scheduling problem

The term 'scheduling' refers to a wide class of problems [9], very different from one another in terms of structure and complexity. Quite expectedly, therefore, many authors have attempted a systematization and a taxonomic classification of the wide range of methods that have grown up over the past 30 years, relating the desired taxonomic axes to those problems [19].

Such a huge effort has produced many interesting results, which allow today to classify many scheduling models in a systematic way, and to unify some algorithmic approaches. However, for the most difficult problems encountered in the scheduling context, it has not yet been possible to indicate a single approach that is somehow "uniformly" preferable. Many works have in fact suggested the use of heuristics, rather than enumeration algorithms, approximation algorithms and so on [12, 3, 11].

A possible way to define scheduling problems is "those decision problems where the factor of importance is time", viewed as a resource (potentially scarce) to be allocated "optimally" in certain activities. Generally, the goals of scheduling are

- maximizing the throughput of the entire system, minimizing the time that the resource is unused;
- attempting to order and regulate resource requests so as to minimizes the ratio of service time (i.e., the time to serve a request) and time "turnaround" (the amount of time that elapses between the instant at which the request is generated and the time when the request is satisfied);
- avoiding undesirable phenomena like starvation or the "eternal waiting" for some requests, experienced in certain conditions;
- giveing system user the perception that multiple requests are met at the same time;

To attain such goals, for example by following combinatorial optimization approaches, in the literature a wide range of algorithms have been developed. Here follows a brief description of some classical algorithms and of some implementations.

A classic example: Round-robin algorithm **Round-robin (RR)** is one of the most widely adopted algorithms for process and network schedulers. According to the most common meaning and use of the term, time slices are assigned to each process

in equal portions and in a circular order, handling all processes without priority (this is also known as "cyclic executive").

Round-robin scheduling is simple, easy to implement, and starvation-free. It can also be applied to other scheduling problems, such as data packet scheduling in computer networks. The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn. In order to schedule processes fairly, a round-robin scheduler generally employs time-sharing, giving each job a time slot or quantum (its allowance of CPU time), and interrupting the job if it is not completed by then. The job is resumed next time a slot is assigned to it. In the absence of time sharing, or if the quanta are large with respect to to the sizes of the jobs, a process that produced large jobs would be favored over other processes.

For example, if the time slot is 100 milliseconds, and job1 takes a total time of 250 ms to complete, the round-robin scheduler will suspend the job after 100 ms and give other jobs their time on the CPU. Once the other jobs have had their equal share (100 ms each), job1 will get another allocation of CPU time and the cycle will repeat. This process continues until the job terminates, and needs no more CPU time.

Job1 = Total time to complete 250 ms (quantum 100 ms).
First allocation = 100 ms.
Second allocation = 100 ms.
Third allocation = 100 ms but job1 self-terminates after 50 ms.
Total CPU time of job1 = 250 ms

Another approach is to divide all processes into an equal number of timing quanta such that the quantum size is proportional to the size of the process. As one can guess from the brief description given, classical RR scheduling algorithm is starvation free, and it is simple, but in general, it does not take into account the task needs as they change over time. hence, if a task has some constraint related to the completion of its workload, RR is not able to guarantee compliance with this constraint and, in general, scheduling policy is fixed or, anyway, has difficulty adapting to the needs (often time-varying) of the task pool to be scheduled.

Following are some examples that show how, trying to overcome the problems just mentioned following the combinatorial approach, results obtained are known to work but presents some flaws[5].

Two Linux schedulers: Complete Fair Schedule and Real Time Round Robin
Two examples of scheduler implementations which follow, in some way, a combinatorial approach, can be found in the Linux kernel; Round Robin algorithm cannot be implemented as it is on a production system, because the fair version (where CPU quantum is fixed and equal for all the tasks) is highly inefficient, and also the version where CPU quantum size is proportional to process size is not enough: if there is a large number of small tasks which need to complete execution within a certain instant, RR does not give guarantees.

A first way to overcome this problems, was the idea to implement a certain number of task queues, each managed by RR dispatcher [1] and to assign to each queue a certain priority level. This solved some problems, but introduced also new ones; to guarantee starvation-free property and to respect task priorities, long-term schedulers have to take into account all the tasks present in all the queues to decide which is the next task to execute. This results in a scheduler having temporal complexity of $\Theta(n)$ where n is the number of tasks present in the system. Such an order of complexity is a quite big problem when number of tasks increase, because a not negligible portion of CPU time is wasted to decide which is the next task to execute.

A big step forward has been made with introduction of the so-called Complete Fair Scheduler (CFS) [1]. The main objective of CFS was to realize a scheduler able to guarantee fairness, with a temporal complexity lower than $\Theta(n)$. To do that, CFS uses a red-black three data structure; in the leaves are stored tasks; the task in the first leave on the right is the task which will execute next, in the last leave on the left there's the task which has just been executed, task's priorities are somehow coded inside the three [2].

In some sense, CFS moves the scheduling problem complexity from time to data structures, in fact, if RR (and its version with Priority Queues) was simple and clear, CFS has a good temporal complexity (in the order of time needed to make an access in an ordered tree, so it is about constant), it uses a complex data structure to access the tree, and maintain it balanced. The algorithm is however quite complex (no criticism intended, needless to say), for example making it not immediate to follow the process by which time slices are calculated. Moreover, CFS guarantees fairness, but it is not particularly able to correctly manage soft and hard real time tasks, in fact, in the Linux scheduler, a priority RR module is still present to manage real time tasks, implemented exploiting priority queues.

1.2. Control Designed Scheduler: Control theory approach

Lately, some literature works proposed to design scheduling algorithms based on the principles of the systems and control theory, that is, task schedulers entirely constituted of a feedback control structure, not of some already functional scheduler with some loops closed around it [17]. In the reference just quoted it was shown that such a design approach has a number of advantages, the major ones being:

- tendentiously simpler scheduling algorithm with respect to "classical" (i.e., not control-centred) ones;
- inherent self-adaptation (in the sense the term is given by the computer science community) to the effective system load, as the presence of a loop makes the control (scheduling) action reacts in a state-dependent manner (in the sense, conversely, of the control jargon);

[1] The scheduler module that takes care of the scheduling in the short term

- possibility to analyze and assess the results formally;

Moreover, in [14] it is recognized that the control structure proposed for task scheduling has in fact a general validity for many resource allocation problems. After all, as mentioned in the 1.1 section, a CPU scheduler must manage, possibly in an optimal manner, a scarce resource (the CPU time of course). Then, it is expected that once the model of individual tasks is developed, one can design a control system that deals with managing the assignment of time slices to the various tasks, and that, exploiting feedback, it is able to do it in a "good" way.

To do it, however, a central fact is that task models are needed *in the control theory sense*, that is, in the form of dynamic systems. Recognizing this leads to identify and dynamically model the core phenomena that rule the controlled object (here, the tasks). The models resulting from such an approach are typically simpler than those obtained by just looking at the computing system from outside, as a black-box entity. The former in fact undoubtedly require more effort and insight on the part of the analyst, but on the other hand inherently avoid the main pathology of the latter, i.e., the necessity of describing the mentioned core phenomena plus a lot of software layers that are not sgrictly part of the object to be controlled.

In [15] it has been shown that on single core architectures, task models are simply time delays, leading to a simple control structure (based on PI blocks) and a simple scheduling algorithm. Since this approach has given such good results on single core architectures, it is interesting to extend the same control-based approach to multicore architectures, designing a Multicore Scheduler based on control theory principles.

1.3. Control Designed Scheduler: Multicore Scheduler Design

On a multi core architecture, in addition to dealing with core-level time slicing, a scheduler must manage the migration of tasks between the different cores [4]. As such, in this context, a task model cannot be a simple delay anymore.

In a multicore context, in fact, there are parallel execution flows, and tasks compete for resource access. There is no determinism and time invariance, because task behavior is time dependent and differs a lot from one class of task to another one, just think about how different is the behavior of the interrupt handler of the mouse from the behavior of a simulation software. Furthermore, load balancing is needed, and for that purpose it is necessary to know what (in terms of resources) a task needs, and when.

The first part of this work focuses on finding out task models suitable to design a Multicore Scheduler following a control theory approach.

The second part is conversely centered on implementing a simulator for **both** single core and multicore control-designed scheduling, in order to have a clean controlled environment where different scheduler structures and different scheduling policies can be tuned and tested in reasonable time, and with no external interferences (existing schedulers and operating system in general).

Considering finally the subject of task models, these can be written starting from principles (i.e., models of physical object or phenomena) or be obtained from data through an identification process; of course, also a mix of the two procedures can be used. In our case, tasks are not a physical object, thus we cannot write model starting from principles, and models have to be obtained from data.

Naturally, it is impossible (and also useless) to write a model for each possible executable task. Hence a preliminary work is needed, as it is necessary to divide tasks in classes, in such a way that tasks belonging to the same class have similar needs from the system's point of view. A possible classification is sketched out below.

- Tasks with periodic deadlines: a certain work load must complete in a certain time span, repeated over time).
- Tasks with a single deadline: tasks go into execution, and after the deadline they expire (hopefully, having completed their workload).
- Tasks with no deadlines: this is the typical situation for interactive tasks, such as desktop applications.
- Event-triggered tasks: this is the case of many services, for example the mouse driver.

Since it is interesting to know how a task impacts on both core and local core cache utilization [21], for each of the classes listed above, it is interesting to further differentiate cases where the tasks are CPU bound, memory bound, or I/O bound. Correspondingly, models which have to be identified from data, should not fit on execution traces of a single specific task, but should represent the whole class of tasks to which the task belongs. Note that task classes have been here described with synthetic high level models which try to be simple and, at the same time, try do describe a wide group of different situations (but with common characteristics) belonging to the same class.

In the following, the mentioned approach to task modeling and scheduler design, are applied to the particular context of this work.

2. State of the art

2.1. Single core scheduler

Some references previously quoted, following the same control-centric approach of this work and specifically devoted to task scheduling [17], are based on a "bare-physics" model of the tasks' behavior. At the beginning of the time span (or round) between a scheduler intervention and the subsequent one, some of the N tasks that are present in the operating system pool, are allotted a CPU time slice or *burst*; at the end of the round, those tasks have used a certain amount each of CPU time, not necessarily equal to their burst. Extending to the entire pool of tasks, this is simply translated in the difference equation

$$\tau_t(t) = b(t-1) + \delta b(t-1)$$

$$\tau_r(t) = \sum_{i=1}^{N} \tau_{t,i}(t) r$$

$$\tau(t) = \tau(t-1) + \tau_r(t-1)$$

where t counts the scheduler interventions, τ_t is the actually used CPU time, b the burst, summations are over the pool, τ_r is the time between two subsequent scheduler interventions (no matter how many tasks were allotted a nonzero burst and in which order), and finally τ is the system time. The disturbance δb accounts for any action on the phenomenon other than that of allotting b, such as for example anticipated CPU yields, delays in returning the CPU whatever the cause may be, and so forth.

A suitable structure to control the system as just modeled, is that shown as block diagram in figure 2.1. Block $R_t(z)$ is devoted to computing the task bursts b in such a way that each task's CPU time usage τ_t follows the corresponding set point $\tau_t^o(t)$. This set point is obtained by partitioning the measured round duration τ_r according to the (possibly time-varying) vector $\alpha(t)$, the elements of which sum to one. An idle task can be introduced to manage the case in which the total CPU share request is less than one, but this is totally straightforward and has no relevance here.

Coming back to the scheme, the bursts are additively corrected by the burst correction b_c output by block $R_r(z)$, so that τ_r follow its set point τ_r^o. This is important because if, for example, some task gets blocked, $R_t(z)$ can still keep the CPU time usage for the others, but the round duration set point is lost. The discrete-time control in figure 2.1 is single-rate, and the rate is dictated by the outer level, i.e., the scheduler performs its computations once per round; the discrete time index t thus counting the scheduler's interventions.

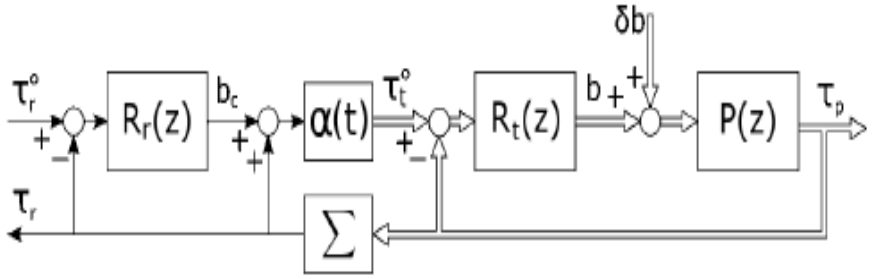

Figure 2.1.: Single core scheduler block schema

2.2. Parallel execution and core switching

In a multicore context, any single core scheduling approach is not sufficient to address the encountered problems, because there are parallel execution flows, and tasks compete for resource access [4]. Tasks cannot be modeled as simple delays any more, and determinism and time invariance are no more a property inherently enjoyed by the devised models, because not only the tasks' behavior is time dependent, which is clearly true also in the single core case, but in addition said time variance affects the scheduling problem, for example by altering the degree of parallelism of a task and/or the communication needs of its threads. Furthermore, and along the same reasoning just sketched, load balancing (or more in general, load distribution management) has to be performed, making it necessary to know what a task needs (in terms of resources) and when.

In a multicore context, one has to observe the coexistence of two different temporal scales:

- a single-core time scale, where the granularity magnitude has the order of the single task time slice,
- and a system time scale, concerned mainly with task migrations that apparently occur on a longer horizon.

The consequences of such a remark, that by the way falls very naturally into the concept of "dynamic separation" typical of many control design methodologies, are numerous and relevant. To mention just the major ones, a context switch (local to one core) has a time and resource cost that is far smaller than that of a migration, whence the need for a bandwidth separation as neat as possible between the two control systems, that are designed along very different performance requirements and costs. Also, when dealing with task migration one has to take into account memory issues, most frequently in the form of of cache misses required for the moved task to have its data available in the local cache of the arrival core. Moreover, memory bound tasks

behaviors are quite frequently encountered, and one would like to migrate this kind of task as few times as possible. At the same time, however, it is not desirable to have all memory bound (and cache critic) tasks on the same core. Finally, on multicore architectures, one would also like to avoid thermal issues: in this respect, a "fair" task-core allocation policy can balance the workload among different cores, in order to avoid thermal gradients, and favor a better performance at physical level.

2.3. Extending control design to a Multicore scheduler

It is clear even from the short discussion above that a multicore scheduler needs a different approach with respect to a single core one; as such, it is legitimate to wonder whether or not the same control design approach, which has given good results for single core scheduler, could be profitable for multicore scheduler too.

As will emerge in the following, the answer seems to be definitely affirmative; however, it is necessary to develop and complete it with other new elements: for example, the task classification dealt with in 1.3 is intended to be the basis for the development of a control structure that manages load balancing.

In any case, control-compatible metrics are needed that characterize how the task impacts on the system performances, since (as already discussed) models cannot be written starting from "principles". It is thus necessary to develop a solution to obtain data from which to start to identify task models. These aspects are discussed in the following parts of this work.

Part II.

Identification for multicore simulator and task profiling

This part of the thesis is divided into two chapters: the first one refers to task profiling, describing the related problems that were faced, and the techniques adopted; the second one is dedicated to the discussion of how task models, suitable for the considered research context, were obtained from data.

3. Task Profiling

In the previous part it has been stated that tasks models are needed. Now it will be explained how task profiling and models identification have been approached. The first sections explain how many and which task classes have been defined; then, which metrics have been defined to characterize task's needs from a system point of view. The last section contains a description of the tools used and how the profiling experiments have been set up. All experiments were run on a Linux machine, equipped with a 3.10.1 kernel.

3.1. Task classification

3.1.1. Performance metrics

For our purposes, it is necessary to define some metrics which can characterize the relevant system performances. From a qualitative point of view, some task performance metrics have been outlined:

- Task execution time, (how much time a task is being executed by a core)
- Task sleep/wait time (how long a task stays in sleep/wait status)
- Cache references
- Cache misses

The long term objective is to develop an optimum load manager; hence, it is necessary to understand what kind of system load will be produced by a given task [6]. The combination of the task execution time and its sleep/wait times allows to estimate its burstiness, which allows to understand its behavior from the CPU point of view: is it a periodic task? is it a batch task? How does it behave from the CPU use point of view?

From the load manager point of view, it is desirable to know if it is possible to improve performance migrating one or more tasks from a core to an other. As an example, migrating a task because it is making a significant amount of memory allocation, and so it is doing a lot of page faults, changes nothing from the memory subsystem point of view, as after the task is relocated it will restart making page faults on the other core (and, moreover, it will make cache misses too, because the task's data are not present in the local cache of the new core). So, there's no reason to relocate tasks on page faults basis, but it could be a good idea to use task migration for cache optimizations, as if there are two or more cache-heavy tasks on the same core, migrating

one of them could improve system performance. In this way, conflict misses can be partially reduced and the cache miss rate can be improved. This simple example motivates the selection of cache misses and cache references as representative metrics, as they represent how much the cache subsystem is stressed by a certain task.

3.1.2. Task classes

Having identified, in a qualitative way, suitable performance metrics, it is clear that their optimization cannot be addressed if not by means of some task pool characterization. Starting from performance metrics, we have therefore tried to qualify a significant set of task classes.

Four classes have been identified:

- Tasks with periodic deadlines (periodic tasks)
- Tasks with a single deadline (batch tasks)
- Interactive tasks (priority tasks); they have no deadlines and usually have a high degree of interactiveness with the end user (i.e. text editors, file managers etc)
- Event triggered tasks (event based tasks): they have no deadlines and usually need to be executed after that an event (or a group of events) has occurred (i.e. the mouse driver, or other event based services).

Periodic tasks: tasks which have a periodic behavior fall in this class: i.e. video players, audio players, periodic communications etc.

Batch tasks: tasks which have low interactivity and CPU/memory intensive load: simulators, scientific calculus software, model checkers etc.

Priority tasks: this class includes tasks which do not require a lot of CPU, but that require it frequently and with low latency: text editors and desktop applications are the most representative tasks.

Event based tasks: tasks which require system resources after that an event (or a group of events) has occurred. Usually they need a good system responsiveness: these task class may include kernel threads activated in response to an interrupt.

3.2. Profiling experiments

There are not physical principles suitable to write task models, so it is necessary to setup appropriate experiments to obtain suitable data sets. There are different possibilities to obtain data:

- Task instrumentation

- Nominal characterization: static characterization, assigns the target (cumulative) values of the metrics to the task

For the purpose of multicore load balancing it is important to know the temporal trends of the various performance metrics exploited in section 3.1.1, so nominal characterization is not a good choice. It thus is necessary to select representative samples of applications for each task class, and profile these tasks.

3.3. Profiling tools: Performance Counters subsystem

For profiling, tasks to be profiled are needed and profilers are needed too. Moreover, on modern computer systems, instruction throughput is very high, so there's a huge amount of hardware events (like for example cache misses) which occur in a very short time. To make application profiling, we need a profiler which is as lightweight as possible light, both in terms of memory and CPU occupation, to not risk interfering with the task to be identified. To decrease the computational overhead of the profiler and thus improve the quality of the gathered data, only events related to the metrics outlined in section 3.1.1 will be measured. In addition, to record data without missing sample points, a profiler has to be executed with a high priority.

The first profiler tried was perf tool, included in the linux-tools package. It is based on performance counters, which are a hardware somponent designed to aid application profiling by presenting a set of special registers, that can be configured to count the occurrence of a predefined set of events. The perf tool tool gives plenty of possibilities about event monitoring, but it gives only cumulative data, i.e. it merely reports the number of events that occurred during the entire lifetime of an applications instead of reporting their dynamics, so it resulted useless for this specific profiling intent, that is reconstruction of temporal transients.

Although it was clear that perf tool was not suitable, performance counters could still be used for task level profiling, but a dynamic profiling library needed to be developed, that periodically reads performance counters and logs the returned value.

Performance counters are special hardware registers available on most modern CPUs. These registers count the number of certain types of hardware events: such as instructions executed, cache misses suffered, or branches mis-predicted without slowing down the kernel or applications. The Linux Performance Counter subsystem provides an abstraction of these hardware capabilities. It provides per task and per CPU counters, counter groups, and it provides event capabilities on top of those. It provides 'virtual' 64-bit counters, regardless of the width of the underlying hardware counters. Performance counters are accessed via special file descriptors. There's one file descriptor per virtual counter used.

The special file descriptor is opened via the `perf_event_open()` system call, returning the new file descriptor. It can be used via the normal VFS system calls: `read()`

can be used to read the counter, `ioctl()` function can be used for special actions on the counter (i.e. enable/disable or reset counters). `fcntl()` can be used to set the blocking mode, obviously, these file descriptors cannot be written by user applications. Multiple counters can be kept open at a time, and counters can be polled.

Observation 3.3.1. Glibc does not provide a wrapper for this system call; it must be called using `syscall()`. This performance counters abstraction is a Linux specific solution, not available on other systems. So, this profiling library is not portable (without modifications) on other operating systems. If someone wants to use this library on a system different from Linux, it is necessary to find out which is its specific performance counter abstraction and modify the profiling library, using the proper system call.

Performance counters subsystem supports very extended data structures, that will be only partially reported (to improve readability). features useful for the purpose of this thesis will be outlined and commented.

The first function used is `perf_event_open()`:

```
int perf_event_open(struct perf_event_attr *attr, pid_t pid, int CPU, ...)
```

- `pid` is process pid (intended for a per-process profiling);
- `CPU` is CPU identifier; it has to be different from -1 only if a per-CPU profiling is intended; it goes from 0 to MAX_NUM_CORE-1;
- `attr` is a pointer to a `perf_event_attr` structure: this is one of the most important data structures, describing specific counter parameters.

Below is partially reported `perf_event_attr{}` data structure:

```
struct perf_event_attr {
  __u32     type;         /* Type of event */
  __u32     size;         /* Size of attribute structure */
  __u64     config;       /* Type-specific configuration */
  .
  .
  .
  __u64   disabled   : 1,   /* off by default */
  inherit            : 1,   /* children inherit it */
  pinned             : 1,   /* must always be on PMU*/
  exclude_kernel     : 1,   /* do not count kernel space events */
  exclude_hv         : 1,   /* do not count hypervisor space events */
  exclude_idle       : 1,   /* do not count when idle */
  .
  .
  .
  inherit_stat             : 1,   /* per task counts */
  enable_on_exec           : 1,   /* next exec enables */
  exclude_callchain_kernel : 1,   /* exclude kernel callchains */
  exclude_callchain_user   : 1,   /* exclude user callchains */
 }
```

`type` is an unsigned integer; it points out three possible types of events:

- `PERF_TYPE_HARDWARE`: are hardware events, measured directly by PMUs [1] registers. This kind of events is very reliable; hardware measures are very fast and can get a large amount of events occurred in a very small time. Direct access to PMUs is one of the most pleasant features of performance counters Linux subsystem. Their limit is the number of PMUs present on the CPU (four on the machine used for this work).
- `PERF_TYPE_SOFTWARE`: are events that are measured by a software wrapper; are less fast and precise than hardware events, on the other side, number of events measured is not limited by number of PMUs presents on chip.
- `PERF_TYPE_TRACEPOINT`: trace point events; generally are operating system related, i.e. scheduling events, kernel events etc. They are measured in a similar way with respect to software events, with similar pros and cons. Here they are not analyzed in detail, because another, more precise, tool has been used to get kernel events.

`size` is perf_event_attr structure size.

[1] Performance Monitoring Unit, are special hardware registers present on every core.

`config` is event specific configuration field: it is encoded inside performance counters subsystem by macros (for software and hardware events) or by specific numeric id, found in kernel debug file system (for tracepoint events). In particular they can be found in /sys/kernel/debug/tracing/events subdirectories.

`disabled` indicates whether performance counter is active or inactive; when a counter is disabled, it maintains its counting value.

`inherit` indicates whether children processes inherit or not father's performance counters (yes by default).

`pinned` indicates whether event is permanently present on PMU for counting or not; when pinned is 0, cumulative sampling mode is activated and counter is incremented after that a certain number of events have occurred: this introduces cumulative sampling and we want to avoid it, so, in this work, pinned is always 1.

`exclude_kernel`: 1 indicates to do not count events in kernel space (1 by default). this field will remain 1, profile is intended to detect task behavior in user space.

`exclude_hv`: 1 indicates to do not count events in hypervisor space; this field is useful only for profiling in virtual machines environments. in this work it will always be 1.

`exclude_idle`: 1 indicates to do not count events when task is idle

`inherit_stat` indicates whether profiling is per-task or per-CPU. 1 means per task profiling (default).

`enable_on_exec`: if it is 1, indicates that counter will be automatically enabled on the next `exec()` execution (1 by default); it is especially useful if profiler is a stand alone executable and we want to profile a command passed by command line.

`exclude_callchain_kernel`: if 1 it exclude kernel callchains from event counting (useful for tracepoint events, 1 by default).

`exclude_callchain_user`: if 1 it exclude user space callchains from event counting (useful for tracepoint events, 1 by default).

3.4. Profiling tools: trace-cmd

In this work, it was considered that profiling only ad-hoc applications could be not sufficient to give an idea of what really happens, from a system point of view, when a common use application is executed. Common use applications, in fact, have some characterization, but are not 'exactly' periodic, batch, event based or priority, more-

over, application characterization could be time variant. So it was decided to profile common use applications belonging, in principles, to each of the task classes spotted. Fearing that the approach based on performance counters could not be sufficient to catch the large amount of events generated by a common use application, which usually has also a heavy graphic user interface, an other tool has been used: trace-cmd tool [20]. This tool is based on ftrace and adopts a timestamp based approach, furthermore it provides fine filter options, directly on the events to record or on the events reported.

A little extract from trace-cmd man page: "trace-cmd is a user interface to Ftrace. Instead of needing to use the debugfs directly, trace-cmd will handle of setting of options and tracers and will record into a data file.

The record feature of trace-cmd uses the system call "splice". Splice allows the user space to move pages directly from the Ftrace ring buffer into a file or network without ever needing to go through userspace. This move is a zero copy algorithm. A page is removed from the ring buffer and sent directly to the destination. No copy is required making the record extremely fast.

This method is even faster than mmap. mmap is quick to get the data from the kernel to userspace, but if the data has to be saved to a file or to the network, then a copy is still required to send the data from userspace back to the kernel.

The trace data format file records all the necessary data to move the data file to other machines and be able to read it there. The endianess and long size is also recorded such that big endian trace data files can be read from little endian machines, and vice versa."

After recording data, and generating `trace.dat` file, it is possible to furthermore filter data and generate a text file. Generally, exploiting trace-cmd record capabilities, only events of our interest were monitored (as sched_sleep, sched_wait and sched_wakeup). Unfortunately, time stamp approach is suitable to measure execution times (ton) and sleep/wait times (toff), but it is less well suited to the measurement of cache references and cache misses; we'd like to know how much cache miss and cache references are made by a task in a millisecond, trace-cmd tells us when a cache miss or a cache reference are made. Anyway, for ton and toff measures, it is an helpful tool; after that data have been recorded, exploiting trace-cmd report and some other filters, a time ordered log file has been generated and finally, through a perl script, timestamp values have been extracted and put in a row ordered csv file; each row has this format:

$$sleep/wait\ timestamp, wakeup\ timestamp$$

. As it can be seen, trace-cmd is a efficient low-level tool, which relies on kernel debug file system. In this way a fast reliable tool is at our disposal, at least, we can measure ton (effective execution time) and toff (sleep/wait time) also on high stressing tasks.

Hereafter we'll see that it is possible to get ton and toff profiling measures both with trace-cmd (for ton and toff) and dynamic profiling library, but, in general, data obtained by trace-cmd have a better quality and are better suited to the identification procedure; that is reasonable, because trace-cmd time stamps have nanosecond resolution. Cache measures are obtained by dynamic profiling library, which has,

anyway, millisecond resolution.

3.5. Performance profiling dynamic library

Once the Performance Counters subsystem has been briefly introduced, performance profiling dynamic library can be described. In this work, a dynamic library and a stand alone profiler [2] have been developed, both in c++ language.

Dynamic library has a multithreaded structure and a very simple interface:

`void start_profiler(pid_t pid)` this function starts profiling; profiling is intended per-task, so pid is process identifier of the task who has to be profiled.

`void stop_profiler()` this function stops profiling and writes samples on row-by comma separated values file.

Library structure is quite simple and centered around performance counters, described in section 3.3: substantially `start_profiler(pid)` creates a new thread, inside thread are initialized

- a locker
- a mutex variable
- an array of four performance counters

Performance counters are here detailed:

- CPU clock cycles counter (hardware counter)
- Context switches counter (software counter)
- Cache references counter (hardware counter)
- Cache miss counter (hardware counter)

They are all set up to do profiling in user space only (excluding kernel, hypervisors and callchains), to make a per-task profile and to inherit counters by child processes. Including directly the library inside application code, enables profiling invoking `start_profiler(pid)` and `stop_profiler()` functions directly from code. Parameter `pid` is the process identifier of the task to be profiled. When `stop_profiler()` is called, profiling is stopped and data are written in `profile.csv` in the present working directory.

Profiling library reads counters in polling and uses a singly linked list structure to save profiling samples; each millisecond, profiling samples are saved into an array of four elements (one element for each counter) and this array is saved in a list node. When profiling stops, all data saved in the list are written in `profile.csv` file. Data

[2] which, anyway, uses dynamic profiling library

are written off line to avoid library from generating a lot of I/O interrupts, heavy to manage (we want a profiling sample **every** millisecond). Here follows description of the singly linked list data structure and a sketch of profiling algorithm (in particular a sketch of the thread function called by profiling algorithm):

```
 // Global section

 // Globally define number of performance counters managed
 #define NCOUNT = 3

 // Global boolean flag, true if profiling should be stopped
 static bool quit = false;

// Global flag,
// it prevents multiple start_profiler(pid) invocations
 static bool started = false;

// Global mutex init, it'll protect quit flag
 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Global profiling thread variable;
// it'll point to thread function a run time
 pthread_t thread;

// Global array of NCOUNT perf_event_attr structures
 static struct perf_event_attr default_attrs[NCOUNT];

// Global array of NCOUNT file descriptors
 static int fd[NCOUNT];


// Define a linked list of profiling samples

 struct prof_entry
 {
  /* Array of NCOUNT unsigned long long integers,
     used to record a single profiling sample;
     one sample for each performance counter.*/

  unsigned long long dataStack[NCOUNT];

  //Pointer to the next list node
  prof_entry* next;
 };
```

Algorithm 1 start_profiler(pid) thread function

Require: `pid` is a valid process identifier
Ensure: `profile.csv` is a row-by file, containing profiling data related to process identified by `pid`; each row has four values, separated by a comma.

```
...
single_count; // Used to read a single performance counter file descriptor
data[NCOUNT]; // array intended to record a single profiling sample
prof_entry* currdata = (prof_entry*) malloc(sizeof(prof_entry)); // currdata is a new list node;
prof_entry * newdata; // newdata is an other new list node
prof_entry * head = currdata; // head is list's head pointer and points to currdata
ofstream prof; // prof is on output stream
new_nice = -20 // High priority, ok under Complete Fair Scheduler module
nice1 = nice(new_nice) // Try to set high priority for profiler execution;
if nice1 != new_nice then
    return exit(-1); // Error, nice fail, abort
end if
...
// default_attrs[] init; every element is a pointer to a struct perf_event_attr{ }; each struct is configured to init a performance counter file descriptor
for (i = 0; i < NCOUNT; i++) do
    fd[i] = sys_perf_event_open(&default_attrs[i], pid, -1, -1, 0);
    if fd[i] < 0 then
        return -1 // Error on perf_event_open( ) syscall
    end if
end for
while 1 do
    usleep(1000) // Sleep for 1 ms
    for for(i = 0; i < NCOUNT; i++) do
        res = read(fd[i], &single_count, sizeof(unsigned long long));
        if res != sizeof(unsigned long long) then
            return exit(-1) // Error on reading performance counter, exit
        end if
        data[i] = single_count; // Store counting values into data[] array
    end for
    ...
    ...
    // Store data[] into current list node and refresh list pointers to prepare new sample recording
    for (i = 0; i < NCOUNT; i++) do
        ioctl(fd[i], PERF_EVENT_IOC_RESET, 0); // Perf counters reset
    end for
    PthreadLocker lock(mutex); // Lock on the mutex
    if quit == True then
        started = False;
        for (i = 0; i < NCOUNT; i++) do
            ioctl(fd[i], PERF_EVENT_IOC_DISABLE, 0);
            close(fd[i]);
            // If quit == True, disable perf counters and close all file descriptors
        end for
        prof.open("profile.csv", ios :: out); // Open profile.csv file in writing mode

        // Scroll linked list from the head
        ...
        // Write data sample row by row into profile.csv file
    end if
end while
```

Now follows `void stop_profiler()` algorithm:

Algorithm 2 stop_profiler()

Require: $start\_profiler(pid)$ has been called in the past
Ensure: $quit =$ `True`
1: $PthreadLocker\ lock(mutex)$; // Get lock on $mutex$ protecting $quit$ flag
2: $quit =$ `True` // Set $quit$ `True`
3: $pthread\_join(thread, NULL)$; // Wait for $thread$ completion

As it can be noticed reading these two algorithms briefly sketched, profiling library is very simple and meant to be light and fast, in fact, profiling every millisecond force to pay high attention on programming language selection, algorithm logic and implementation details. Library has been developed in c++; in this way profiling directly invoking profiling functions can be made in c and c++ applications.

3.5.1. Stand alone profiling module

In this work, have been profiled also common use applications; modify source code of big and complex applications is difficult and error-prone, so, a stand alone profiling module has been developed: it exploits dynamic profiling library exposed in 3.5, and it has, in principle, the same conceptual structure: absolute path of the application to be profiled has to be passed by command line when launching stand alone module; stand alone profiler creates performance counters, initializes them (but it do not enable counting); then it creates a child process. Father process calls `start_profiler()` and then will wait for child termination; after child has terminated, it calls `stop_profiler()`. Child process does an exec, loading code of the application to be profiled. Since `enable_on_exec` flag is 1, when exec is completed, counters are automatically enabled.

Here follows a sketch of the stand alone module.

Algorithm 3 main(int argc, char* argv[])

Require: $argc >= 2$
Ensure: 1 is returned if everything was ok, 0 otherwise.
1: **if** $argc < 2$ **then**
2: $printf(...Program \quad usage...)$;
3: **return** 0
4: **end if**
5: $pid\_t\ pid = fork()$; // Creates a child process
6: **if** $pid > 0$ **then**
7: $start\_profiler(pid)$; // We are in father; start profiling on child process
8: $waitpid(pid,\ NULL,\ 0)$; // Wait for child process completion
9: $stop\_profiler()$; // Stop profiling
10: **else**
11: // We are in child; now execute command passed by command line
12: $execvp(\text{""},\ (char**)argv)$; // Make a dummy execvp for efficiency reasons
13: $argv$++; // Skip process command name
14: $execvp(argv[0], argv)$; // Execute command passed by command line
15: **return** 1
16: **end if**
17: **return** 1

Observation 3.5.1. This library is not meant to profile more applications in parallel execution (simply because with one, we use three of the four PMUs available on profiling machine), but for more complex and powerful systems, is easy to extend library for parallel profiling.

3.6. Profiling setup

It has been already exploited that for a good profiling work a fast and light profiler is needed; similarly, profiling environment has to be set up to be as cleaner as possible, so that other tasks and system itself do not interfere or, at least, interfere as little as possible. In fact, we cannot eliminate current system scheduler and its impact on task profiling, but graphical server and graphic user interface can be (and should be) eliminated, improving profiling quality. Each experiment has been run in a very base Linux environment: here there's a brief list of what has been **eliminated** (with reference to a common Linux based system):

- No user graphic interface (kde/gnome)
- No graphic server (xorg) and graphic applications (dolphin/konqueror/okular/firefox etc..)
- No network managers (nm,wicd, etc..)
- No print servers (cups, etc..)

- No shell/frameworks different from bash.

A system with this kind of configuration is quite fast (from CPU point of view) and light (from memory point of view; it occupies an amount of about 150-200 MB in ram).

A little specification about profiling setup from task and system point of view: in 3.1.2 we have spotted a number of classes, representing principals task behaviors on a computing system. Since performance metrics are related to how tasks use CPU and memory, it is interesting to further characterize task classes, from system resources point of view. From CPU, memory and user interactions point of view, a task could also be characterized as:

- Task CPU bound
- Task I/O bound
- Task memory bound

For each of the four task classes (periodic, batch, priority and event-based), have been written and profiled ad-hoc applications CPU bound, I/O bound and memory bound, except for some unrealistic or redundant combinations: event based and priority tasks, as defined in section 3.1.2, are intrinsically I/O bound, periodic tasks can be I/O bound, but they are better represented by media players, so it is not necessary write new ad-hoc applications, a batch task hardly is I/O bound, and so on. For each pool, some ad-hoc applications have been written, varying some parameters, as period in periodic tasks, programming language in batch CPU bound tasks or number of memory access for memory bound tasks. In the following sections, more details will be provided.

3.7. Profiling results

3.7.1. Periodic tasks

CPU bound

CPU bound ad-hoc periodic tasks are here analyzed: code will not be explained in detail, because is not particularly useful for work's purpose. Task structure is quite simple: using `timer_fd()` interface, which creates file descriptor based timers, a periodic timer is created; main function executes two nested cycles: outer cycle is on the number of periods we want to profile; for each period, a cycle of algebraic operations is executed (inner cycle). When inner cycle has terminated, `wait_period(& infos)` function is called and task wait for timer expiration; in this way, the number of outer cycle iterations is the number of periods executed (and profiled), inner cycle, instead, represents single period workload, so we can tune :

- Period duration

- Number of periods
- Workload intensity

When the task ends inner cycle iterations, this in fact results in a blocking `read()` on the timer (it goes in wait status).

Algorithm 4 Periodic_main() template

```
1: pid_t pid = getpid(); // Get process pid
2: start_profiler(pid); // Start profiling current task
3: int i, j, workload // workload is number of inner cycle iterations
4: int period // Task period (in milliseconds)
5: int periods // Number of periods to do
6: double p, t, n // Support variables
7: struct periodic_info info; // Support structure
8: make_periodic(period, &info); // Init periodic timer
9: while (i < periods) do
10:   i++; // i starts from 0
11:   for (j = 0; j < workload; j++) do
12:     ...
13:     ...
14:     // Inner cycle, performs algebraic operations
15:   end for
16:   wait_period(&info); // Support function; makes task waiting for periodic
      timer expiration
17: end while
18: stop_profiler(); // Stop profiling and terminate
```

Here there are some plots referred to some of the profiled variables:

Figure 3.1, shows that there's a periodic trend of CPU clock cycles; for every rising/descending front there's a context switch and, sometimes, this context switch is recorded with some delay (compared to rising/descending fronts), an indication that a parallel real time profiling is difficult, even in a light, well setup environment.

In figure 3.2 there is a detail of CPU cycles reported in figure 3.1. Task period should be 4 seconds; we can see that effective period is a bit different; this remarks that Complete Fair Scheduler module has some serious trouble about granting time deadlines; it is ok about fairness, but it is not able to satisfy precise real time constraints (as exact scheduling periodicity of a given task). That is not the only effect of Linux scheduler on task profiling; other effects will be analyzed more precisely in the next figures.

Figure 3.3 spots an other profiling problem: some times (especially on high workload tasks) application throughput is high and a lot of events happen in the same time. Since real parallelism is limited (on testing machine we have two independent cores), sometimes it is not possible to read and record all events of interest in a single millisecond, so some profiling samples present cumulative measures. In this specific

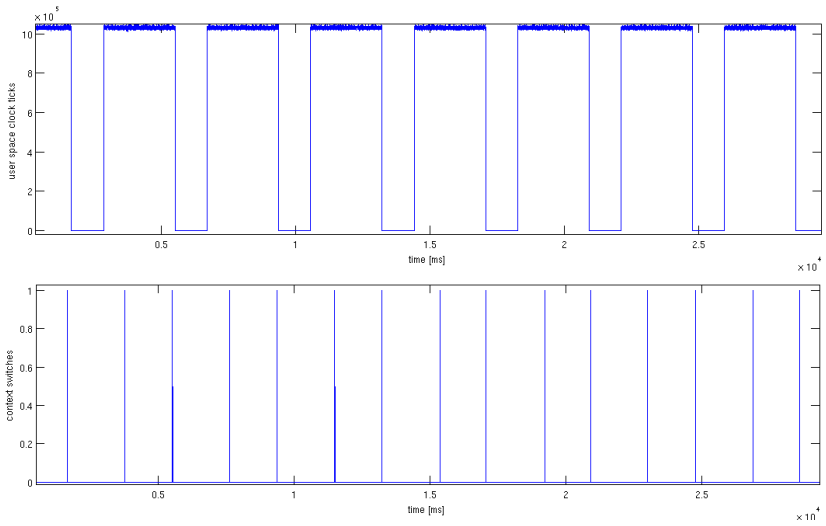

Figure 3.1.: CPU clock and context switches of a periodic task

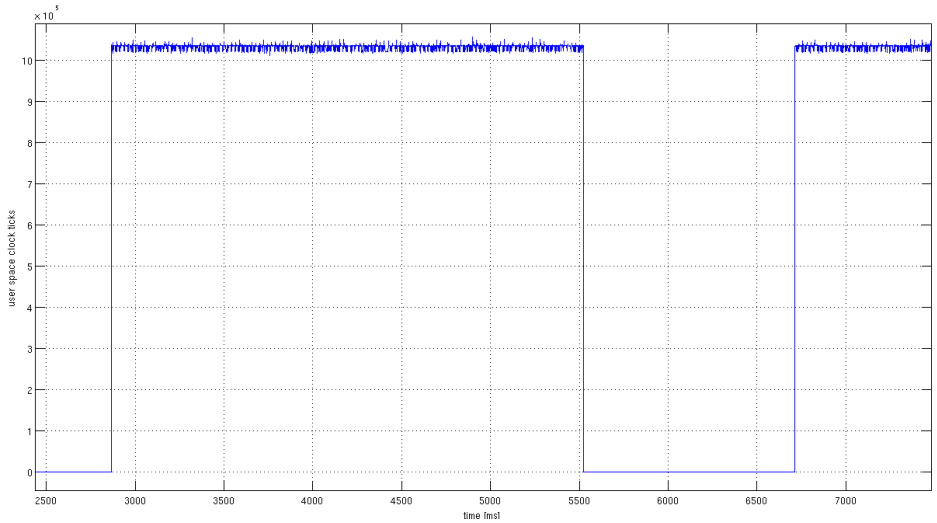

Figure 3.2.: Effective period of a 4 seconds periodic task

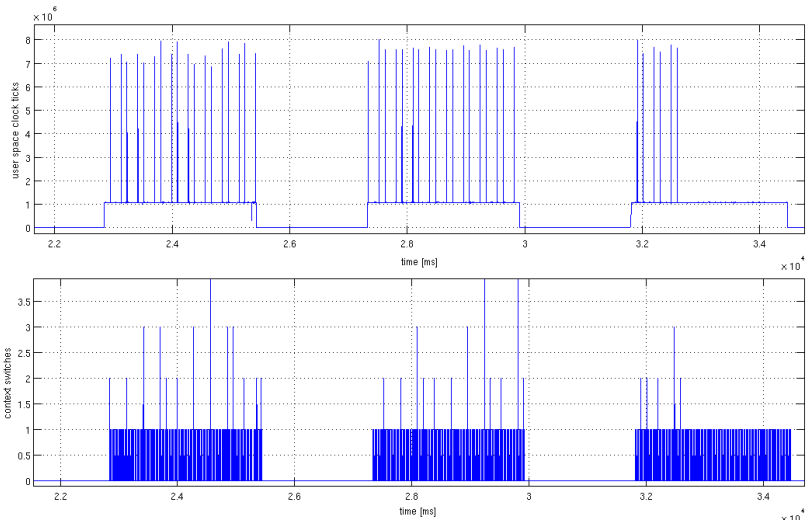

Figure 3.3.: Cumulated context switches problem on a high workload 4 seconds periodic task

example, some times there are two or three context switches in the same millisecond (which is quite unfeasible). The same problem is attributable to very high peaks on CPU cycles. This problem is quite difficult to completely eliminate from high workload experiments (even using Real Time scheduler module), so, in this cases we content ourselves of 'good enough' measures, filtering out what is patently implausible.

Moving on to analyze cache miss and cache reference, an other unexpected (and unfortunately general) problem comes out:

Figure 3.4 shows a detail of context switches and cache references plot, spotting that Linux scheduler and its 10 ms periodic scheduler tick, distorts some measures. In fact these measures, which were made before compiling a new kernel with 1 ms periodic scheduler tick, show that there is not a cache reference sample for every millisecond, but a cumulated sample, recorded every 10 ms, when scheduler ticks is raised and a reschedule happens. Simplifying, even if task is executing, reading from performance counters needs a system call and a context switch, which cannot be performed every millisecond, but it is bound to 10 ms scheduler tick periodicity, because that is the minimum rescheduling granularity. To resolve this problem, which is general and affects also all other kinds of tasks, it was necessary to compile a new Linux kernel (version 3.10.1), setting scheduler tick periodicity at 1 millisecond.

Periodic tasks analysis has been exploited to remark some problems faced during profiling operations, some specifically related to periodic tasks (as effective task period), other more general, as cumulative context switches (related to all high workload tasks) and measures distorted by 10 ms periodicity related to scheduler tick (very general problem, affects all tasks).

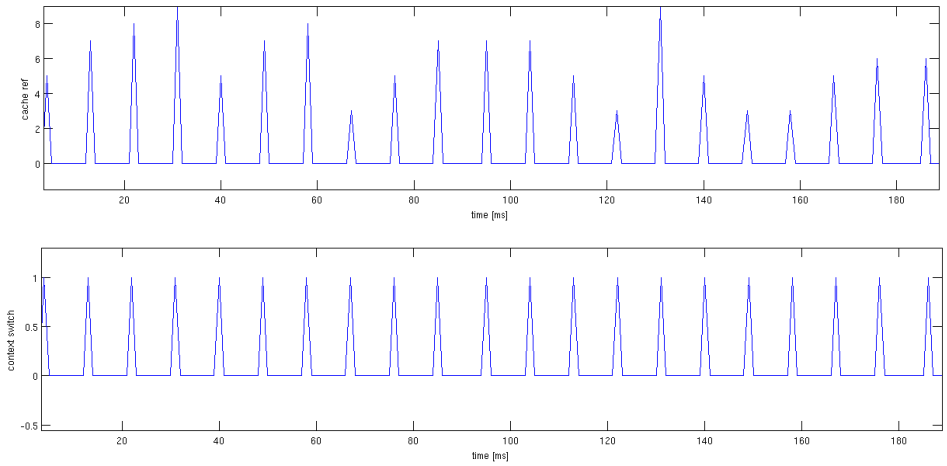

Figure 3.4.: 10 ms scheduler tick influences cache references dynamic

In next figure 3.5 is reported detailed cache references and cache miss plot related to a piece of execution of a periodic task profiled on a kernel with 1 ms scheduler tick periodicity. Cache reference dynamic has no more a "periodic" aspect. From now on, all experiments reported have been made under a 1 ms scheduler tick Linux kernel.

It can seem strange that cache miss are fixed to zero, but reader should remember that this profiling is related to a CPU-bound periodic task; as it is sketched in algorithm 4, this kind of task uses a very little amount of memory (a couple of counters and a couple of numeric variables). Its data are quickly loaded into memory and are very small, so cache miss are quite unlikely to happen. watching plot 'in the large' as it is showed in figure 3.6 however it can be noticed a little amount of cache miss, scattered along task execution.

Memory bound

Now we are going to take a look over memory bound periodic tasks; task structure is quite simple and similar to CPU bound one. The base idea is to create a periodic task which makes a large memory allocation in the heap, at the very beginning of its execution, and then makes, periodically, random memory accesses (read or write accesses).

```
// Global section, support structure definition
  typedef struct{
  long int field[8000];
  } bigMem;
```

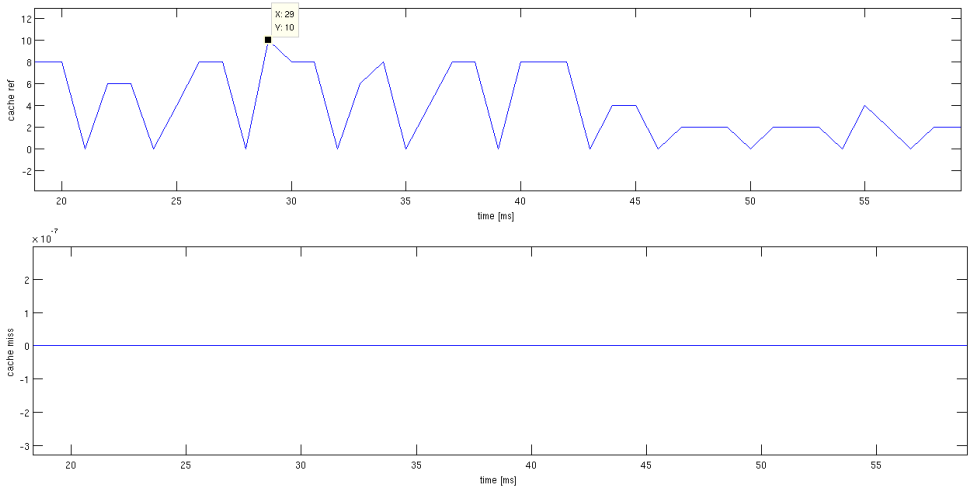

Figure 3.5.: Cache references plot with 1 ms scheduler tick

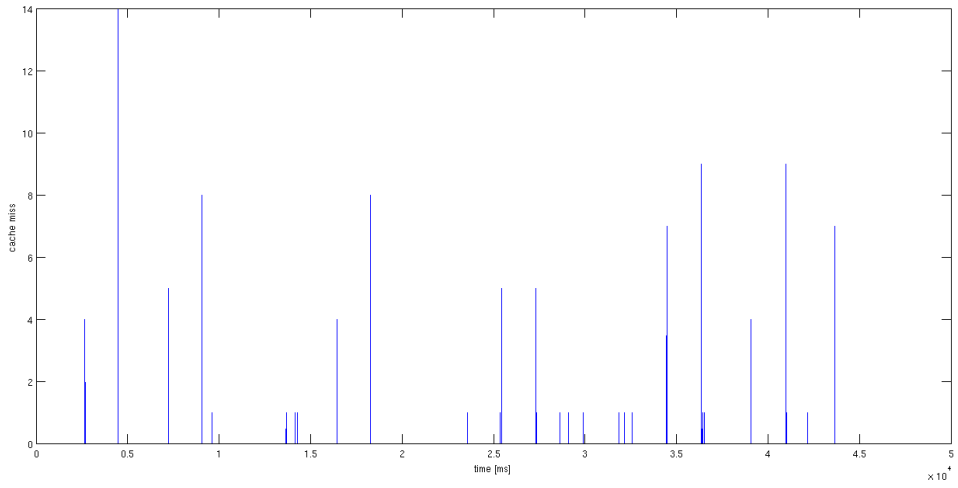

Figure 3.6.: Cache miss plot

$bigMem$ is an array of 8000 $long\ int$ variables; it is the base data structure: each variable in the array has 8 Bytes size (testing machine is a 64 bit machine), so each $bigMem$ variable has a size of 64000 Bytes (that is about 64 KB). in algorithm 5 we declare an array of 1000 pointers to $bigMem$ data structure; in this way, the overall data structure exploited has a size of about 64 MB. The overall data structure can not be contained in cache all together. Simulation of a memory intensive, periodic application is carried on making random read/write accesses to the data structure.

In this kind of task we can tune:

- Period duration ($period$ variable)
- Number of periods to execute ($periods$ variable)

Algorithm 5 Memory Bound periodic_main() template

```
1: pid_t pid = getpid(); // Get process pid
2: start_profiler(pid); // Start profiling current task
3: bigMem * data[1000]; // Array of 1000 bigMem pointers
4: int rnd1, rnd2; // rnd1 and rnd2 are random numbers to make memory access

5: int period // Task period (in milliseconds)
6: int periods // Number of periods to do
7: int j, k // Support counters
8: long int read, write = 42 // Support variables
9: struct periodic_info info; // Support structure
10: for (i = 0; i < 1000; i++) do
11:    data[i] = (bigMem*) malloc(sizeof(bigMem)); // Make a malloc of bigMem
       size for each element of data[] array
12:    memset(data[i], 0, sizeof(bigMem)); // Init memory area to 0
13: end for
14: srand(time(NULL)); // Seed random generator
15: i = 0;
16: make_periodic(period, &info); // Make task periodic
17: while (i < periods) do
18:    i++;
19:    for (j = 0; j < 10000; j++) do
20:       rnd1 = rand_lim(1000); // 10 random accesses for each bigMem element;
21:       // Potentially, make 1 access for every element of bigMem.field[]
22:       for (k = 0; k < 8000; k++) do
23:          if (rnd1 % 2 == 0) then
24:             rnd2 = rand_lim(8000); // if rnd1 is even, do a random write
25:             data[rnd1] → field[rnd2] = write;
26:          else
27:             rnd2 = rand_lim(8000); // Do a read
28:             read = data[rnd1] → field[rnd2];
29:          end if
30:       end for
31:    end for
32:    wait_period(&info); // Wait for period to expire
33: end while
34: stop_profiler(); // Stop profiling and terminate
```

On the next figures, we'll have a look at periodic memory bound task plot; this task has a 6 seconds period and makes 80 millions memory accesses in every period:

in figure 3.7 we see that, also for memory bound periodic tasks, there's a periodic trend in CPU clock cycles plot; context switches are coherent with clock cycles plot. Effective execution time (ton) and sleep/wait time (toff) are inferred from CPU cycles plot; when CPU cycles are greater than a threshold, a rising front is recorded; when

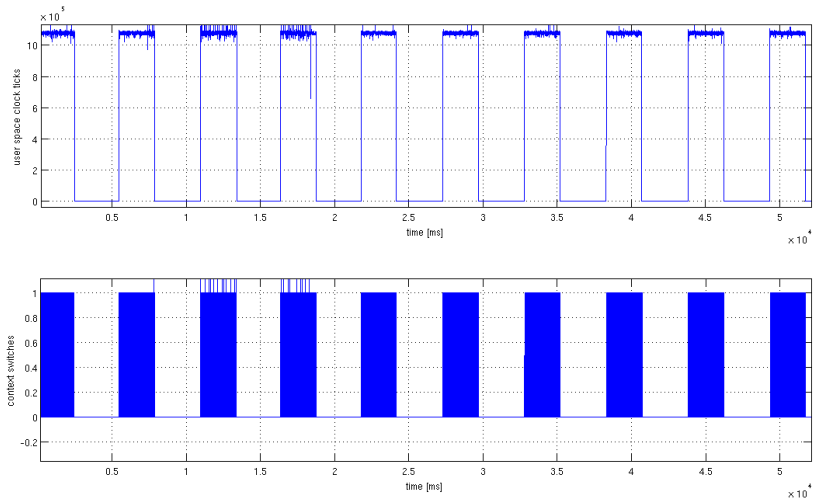

Figure 3.7.: Periodic memory bound CPU cycles and context switches

CPU cycles are lower than threshold, a descending front is recorded. If we use an index i, $desc(i)$ is the i-th descending front and $rising(i)$ is the i-th rising front, ton and toff are calculated with the following formulas:

$$ton = desc(i) - rising(i)$$

$$toff = rising(i+1) - desc(i)$$

This is based on the assumptions that a rising front always precede a descending front, so the first rising front and the first descending front spot the first execution time, whereas, the second rising front and the first descending front, spot the first sleep/wait time. These are the reasons of the index i in the formulas.

What is really interesting in memory bound tasks is cache reference and cache miss plots; they have a square wave like trend, similar to clock cycles trend, but, in the piece of time where task is in execution, CPU clock cycles have a 'noisy' behavior, but everything fluctuate around a mean value, instead, cache miss and cache reference signals have a very remarkable high frequency component, which is quite evident in figure 3.8.

More or less, this is what we expected, in fact, this task makes about 80 millions memory accesses in each period, and memory area accessed is much bigger than cache size, so an high level of stress related to cache subsystem, is a natural consequence; this stress is observed in the form of high frequency components in cache miss and cache references signals. In figure 3.9 and figure 3.10 are respectively reported ton and toff of 150 periods related to a periodic memory bound execution

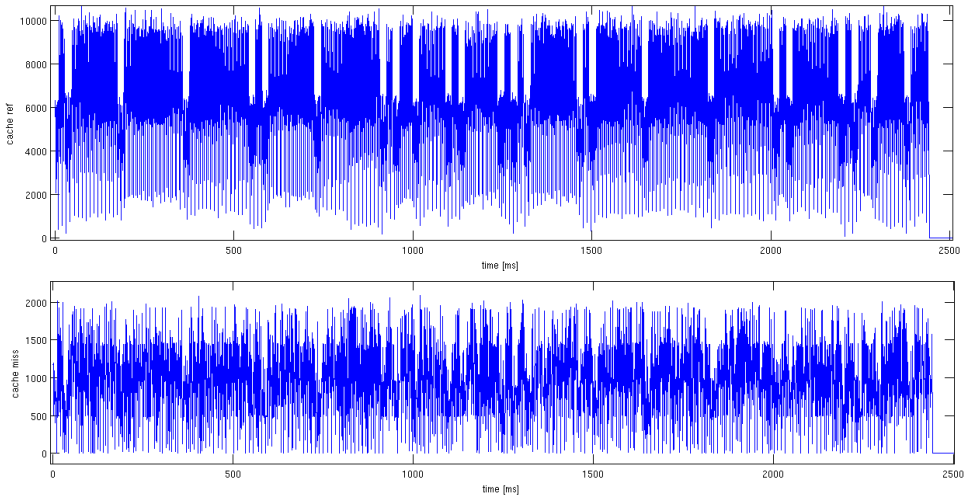

Figure 3.8.: Cache miss and cache reference detail of an execution fragment

trace.

Common use periodic applications: Mplayer

As it was believed that profiling only ad-hoc tasks, could not be sufficient, it was decided to profile common use applications to. For periodic task class, a video player is considered a sufficiently representative application, both from CPU and memory point of view. In this section is reported profiling results of a video streaming playback benchmark performed with Mplayer. Video streaming is Big buck bunny, played at 24 fps and no audio.

Observation 3.7.1. Video players stress system a lot, both from CPU and memory point of view. Initially there was concern that it was not possible to have a good enough profile. Luckily, quite good profiling results have been collected. This probably because Mplayer is implemented in order to have direct access to memory, without going through xorg. In this way, dynamic profiling library is able to read from performance counters every millisecond, avoiding cumulative data readings. Naturally, to profile Mplayer (and similarly for all the others common use applications), it has been used stand alone profiling module, avoiding to modify source code of the application and recompiling it. To make sure that data profiles were good, we profiled Mplayer also with an other tool: trace-cmd, explained in section 3.4 and then a cross-check was made between ton and toff inferred by CPU cycles retrieved from performance counters and the ones directly measured by trace-cmd.

Here are reported profiling data collected with CFS scheduler module and real time scheduler module.

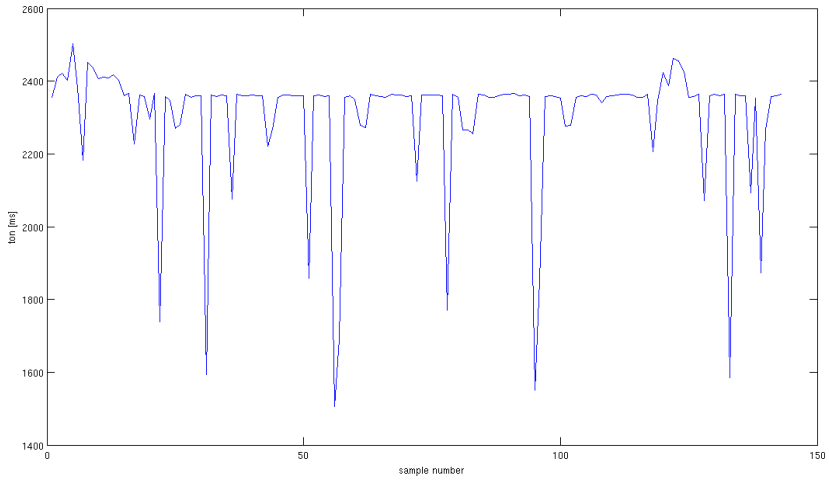

Figure 3.9.: Ton periodic memory bound task

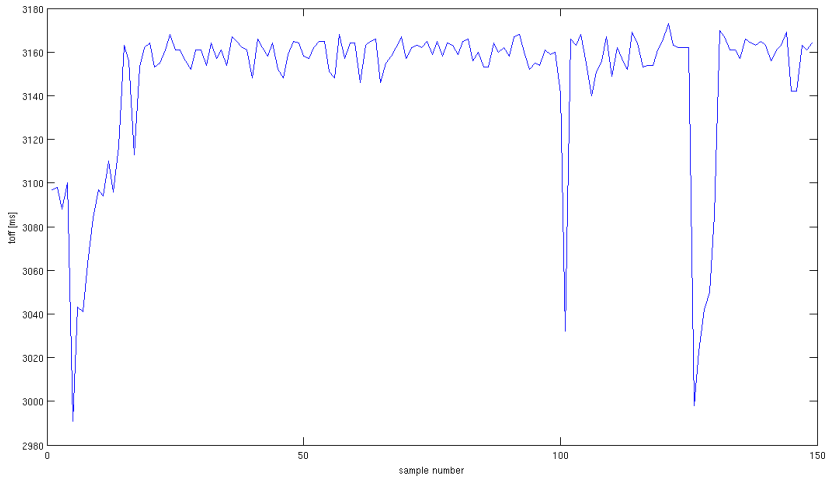

Figure 3.10.: Toff periodic memory bound task

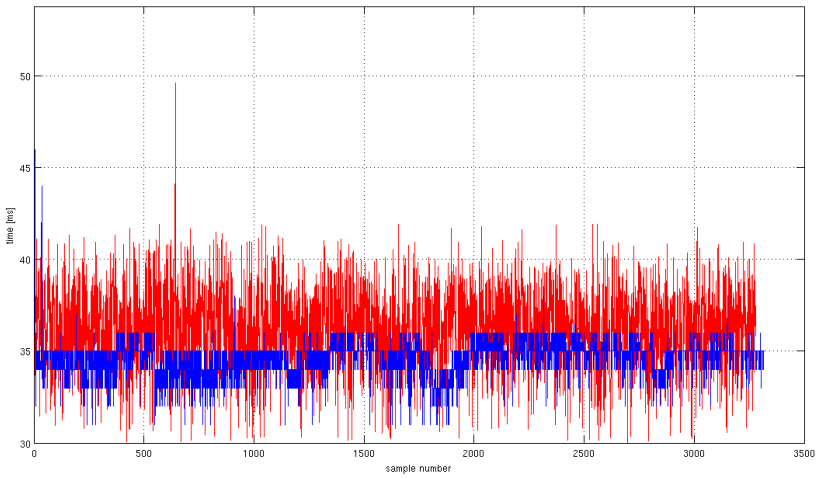

Figure 3.11.: Toff measured on two mplayer executions of the same video stream: red trace is trace-cmd data set, blue one is profiling library data set

Profiling with Complete Fair Scheduler Figure 3.11 reports toff measures of two mplayer executions of the same video stream; red trace is related to toff measured by trace-cmd tool; blue trace is the one related to toff inferred by CPU clock cycles measured by performance counters. Both signals have very evident high frequency components, symptom that CFS module has some troubles to support decoding at 24 fps and profiling at 1 ms at the same time. Anyway, in both signals there's a quite evident mean value, around which measures oscillate; for blue trace is about 35 ms, for red trace is about 36 ms.

Figure 3.12 reports ton related to two different Mplayer executions of the same video stream (remark: these executions are the same from which toff has been extracted). Here overlying is more difficult, so it is reported a zoom on the first 550 samples; however there are high frequency contributes around a mean value in both signals. That is ok, because ton depends on scheduler module and on the complexity of the frame which has to be decoded, so some variability, even on the mean ton value was expected. Anyway, both mean values are about 2 ms (reasonable, if we think that kernel was compiled to have 1 ms periodic scheduler tick).

Considering that measures are related to two different mplayer executions, results are not so bad and we can conclude that video player (or at least Mplayer) are effectively periodic tasks and, moreover, data collected by performance counters are usable, and suitable to identificate a model.

Figure 3.13 reports cache references and cache miss plots. Nothing in particular needs to be remarked; plots are reasonable, they resemble a chain of pulses, and cache reference are always much bigger than cache miss.

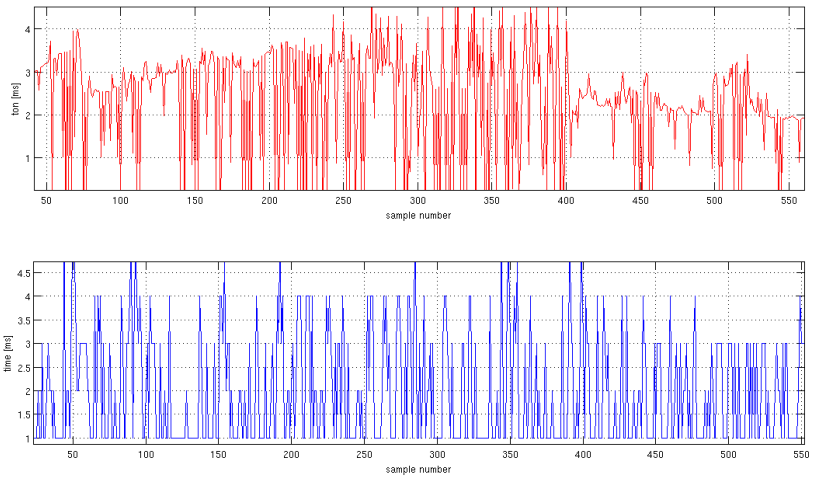

Figure 3.12.: Ton measured on two mplayer executions of the same video stream

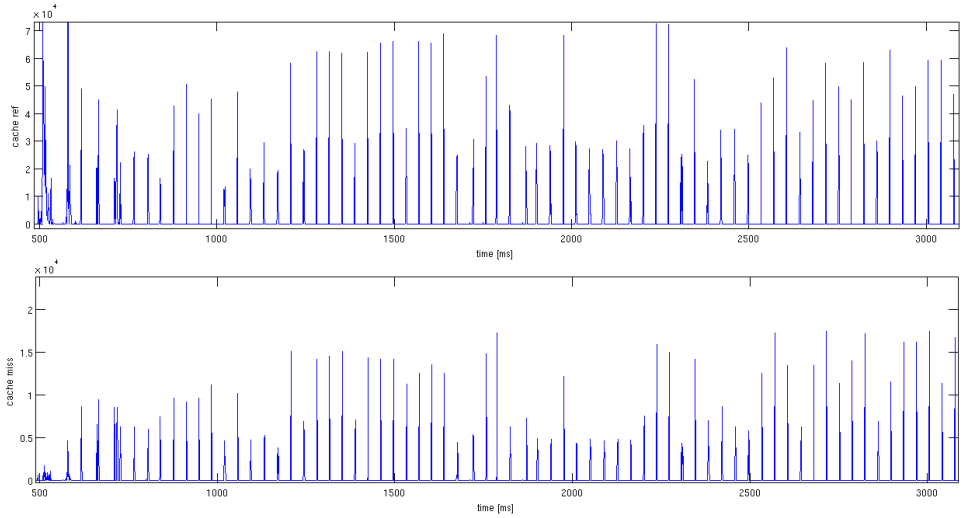

Figure 3.13.: Cache reference and cache miss related to about first 3000 samples of mplayer execution

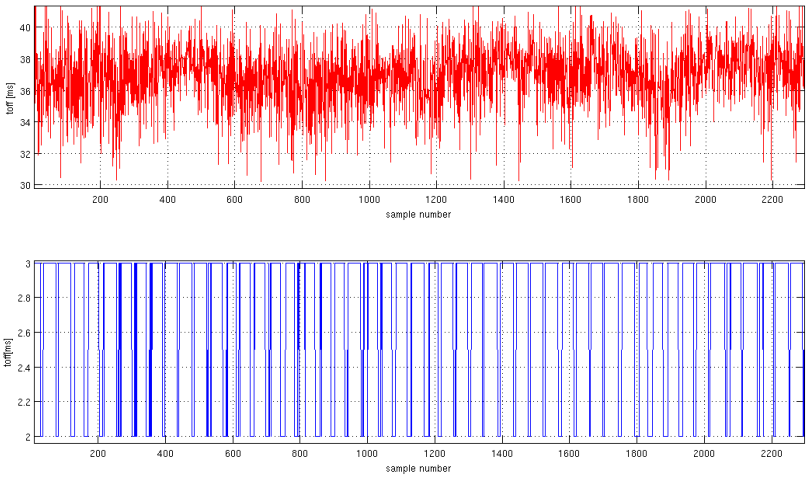

Figure 3.14.: toff related to Mplayer execution under Real Time module:red trace is related to trace-cmd data set, blue trace is related to profiling library

Profiling with Real Time module Since profiling under Complete Fair Scheduler module remarks a quite high variability on data collected, execution under Real Time Round Robin module of Linux scheduler, has been forced. Maximum priority has been assigned to Mplayer execution (99, in a scale from 1 to 99).Real Time module should ensure respect of the periodicity of video playback (in this case 24 fps). Here are reported obtained results.

in figure 3.14 we see toff profiled using trace-cmd (red signal) and toff profiled using performance counters (blue signal). In this case trends are quite different each from the other. toff measured by trace-cmd is always around 35 ms, whereas, watching at blue signal, we see a very precise square wave, oscillating between 2 and 3 ms. The suspicion is that, with performance counters, we catch some dynamics winch are strictly related to scheduler and that are not caught by trace-cmd, as subsequent reschedules of mplayer itself: the idea is that, having an high priority in this execution trace, Mplayer is often rescheduled anyway; when it doesn't need to be executed, it undergoes immediately preempt. Probably trace-cmd ignore these reschedule of the same task, giving us more user-space related data, while performance counters catch every single low level scheduler interruption (and that could be the reason for such a precise square wave). in fact, in figure 3.15 there's a context switch every millisecond.

In figure 3.16 dynamics are substantially similar to the ones represented in the previous figure; data collected by trace-cmd and the ones collected by performance counters are different, but here values are more similar. This could confirm that there are some discrepancies on detecting chain of reschedule related to the same task (Mplayer).

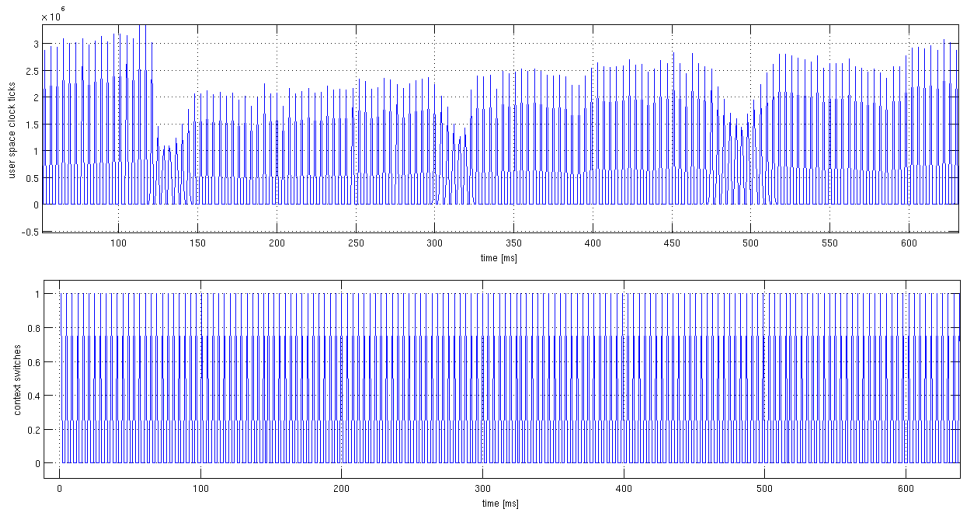

Figure 3.15.: CPU cycles and context switches of related to mplayer video playback under Real Time module

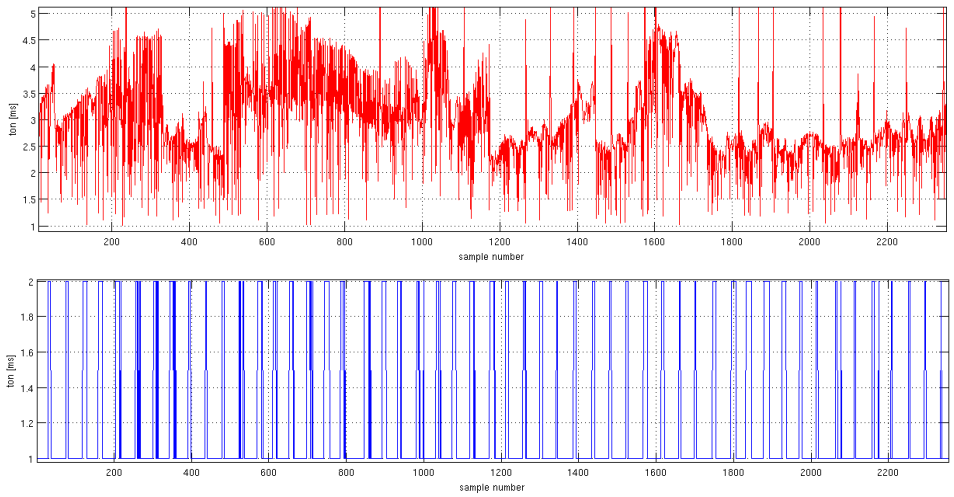

Figure 3.16.: ton related to two different Mplayer execution under Real Time module

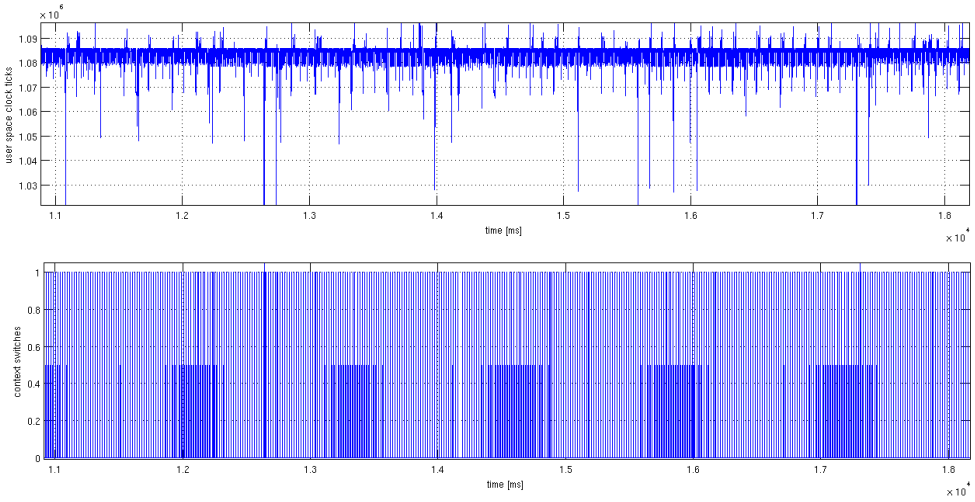

Figure 3.17.: CPU cycles and context switches of a batch task implemented in C

3.7.2. Batch tasks

Now analysis will be focused on the batch class, starting from batch, CPU bound tasks.

CPU bound

Batch CPU bound tasks are very simple, from the code point of view; they are just a cycle with $O(n^2)$ complexity, where n is big enough to have a sufficiently long profile. To improve work readability, algorithm is not reported. In particular, batch CPU bound tasks have been implemented using c language and python language, pointing out quite different results, especially from cache point of view. Here follows profiling results:

Figure 3.17 shows CPU cycles and context switches; results are more or less what we expected; CPU cycles settle to a mean value and there is a sequence of context switches.

More interesting is cache reference and cache miss plot, especially if compared to cache miss and cache reference plots of a batch task written in python.

Figure 3.18 reports cache references and cache miss of a batch application written in c; figure 3.19 reports cache references and cache miss of an application written in python. Cache miss have usual random trend. What is very different is cache references plot. In c application, cache references plot is similar to a saw tooth, and that is not strange, some times, task makes memory allocation and some times it does not. In python application, cache references has a saw tooth trend, as reported in fig 3.20, but peaks get larger and larger over time. That could be explained remembering

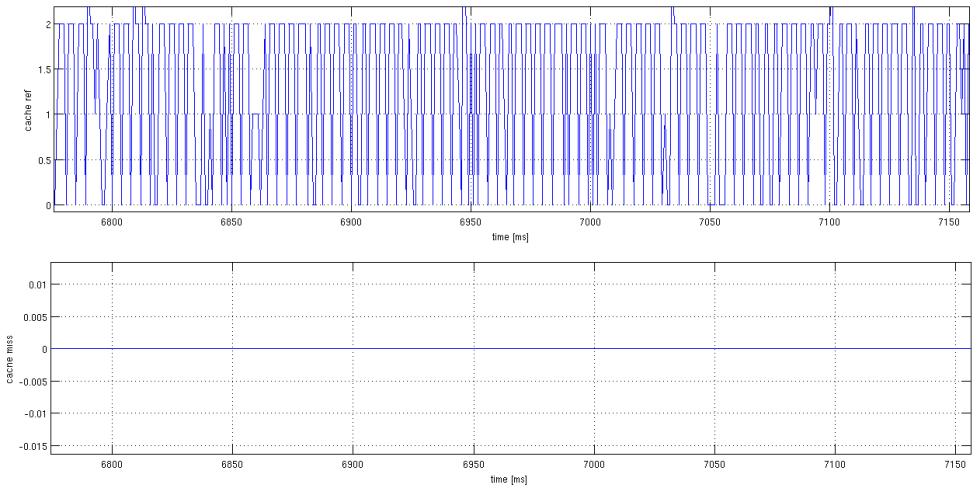

Figure 3.18.: Cache references and cache miss of a batch task implemented in C

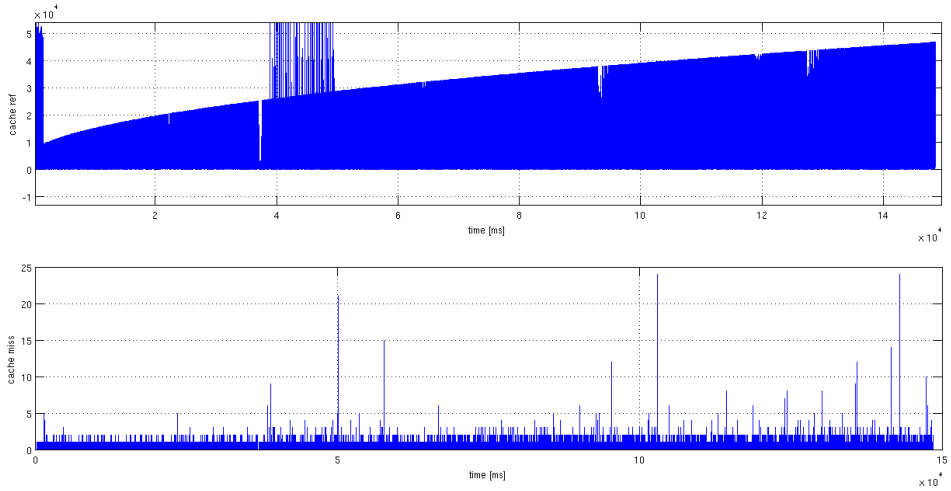

Figure 3.19.: Cache references and cache miss of a batch task implemented in Python

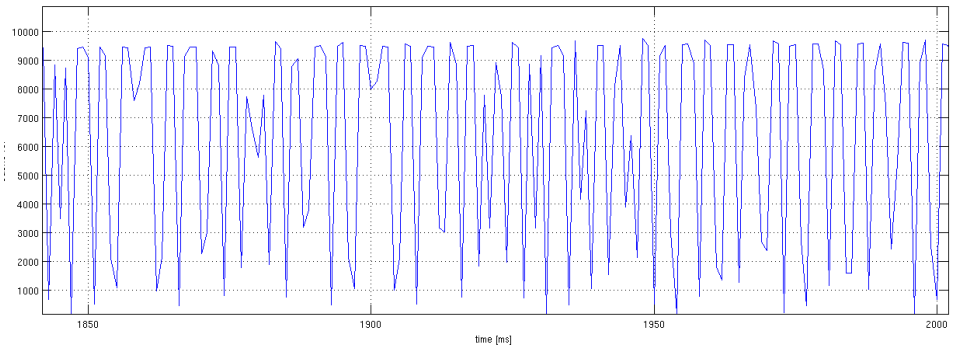

Figure 3.20.: Cache references detail of a batch task implemented in Python

how a python task is executed. Python is an interpreted language, characterized by dynamic typing and dynamic binding. When a python application is executed, an instance of python shell is invoked, then shell executes line by line python code. In python, every object is implemented as a dictionary, in this way is possible to add methods, attributes or other data structures to a certain instance of a certain class a run time. When we make profile, *de facto* we are profiling python shell while is executing our python application's code, and that is why we see cache references grow over time. While execution gradually proceeds, shell dictionaries grow in dimensions (especially if we are executing for cycles) and that is why, in cache, larger and larger memory allocation are done. On the other side, if we are lucky, that memory allocation should help on reducing cache miss.

I/O bound 'batch'

To profile batch I/O bound tasks, ad-hoc applications have not been written; indeed, in this case, the concept of a task batch has been slightly relaxed. In general, a batch task is associated to an application which starts working and, if no external interferences occur, continues execution until termination, so, generally, a batch task is not I/O bound. This time, a little relaxing has been done, including an application of common use: ls. ls usually starts and executes until the entire contents of the directory is shown, so, from this point of view, is a batch task, but, if the directory is big and full of files (as /usr/bin directory) ls has to wait for the disk controller to extract data, fill disk cache and then starts sending extracted data to ls, so, from this point of view, it can be considered I/O bound or, anyway, an interesting case. Exper-

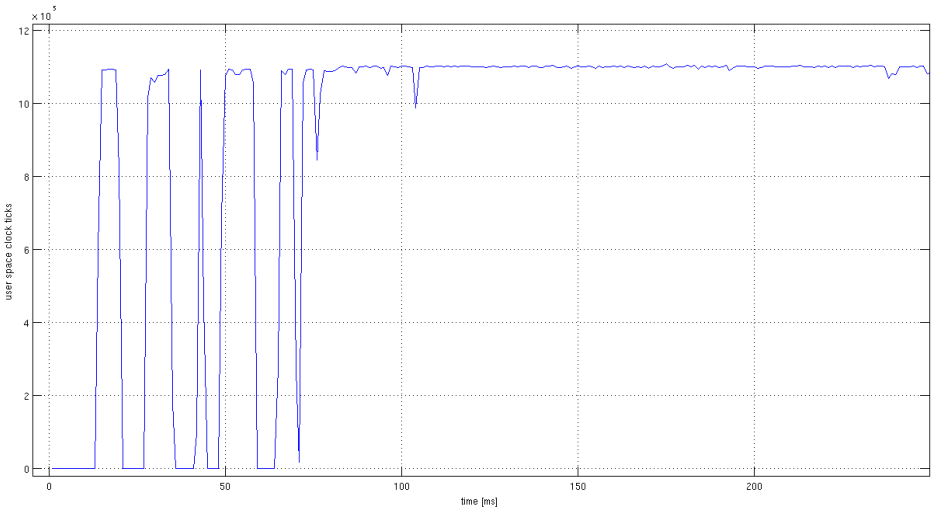

Figure 3.21.: First samples detail of CPU cycles related to ls profiling

iments have been conducted exploiting stand alone profiling module and launching $./tperf \quad ls -la /usr/bin$ bash command in a clean light environment, as it is exposed in section 3.6. ls profiling was carried on in an environment with no user graphic interface, so it was not necessary to force real time scheduler module to obtain good results. $/usr/bin$ contains all the executables (or symbolic links to the executable) present on the machine and launchable by users, so it is a quite big directory, and execution is sufficiently long to allow significant profiling. CPU cycles and context switches plot are not very different from the ones related to CPU bound tasks, so they are not entirely reported to avoid redundancy. What is quite interesting is the very first samples, reported in detail in figure 3.21

At the beginning of the execution, task waits, than executes a bit and then waits again. This can be explained by the procedure of disk caching. $/usr/bin$ is a quite big directory, and $ls \ -la$ retrieves data from file system and then alphabetically orders data and shows it. In the first phase, disk put data blocks in its cache, then the data is retrieved, ordered and shown. Wait time that is visible at the beginning of the execution, is probably due to disk caching: task waits for disk putting all data blocks in disk cache and then starts retrieving data.

in figure 3.22 is reported a zoom on the first 1000 samples. Cache miss trend is interesting, in fact, during disk caching phase, as it can be expected, there are high cache miss peaks, also due to the fact that ls own code and data have to be loaded in core local cache.

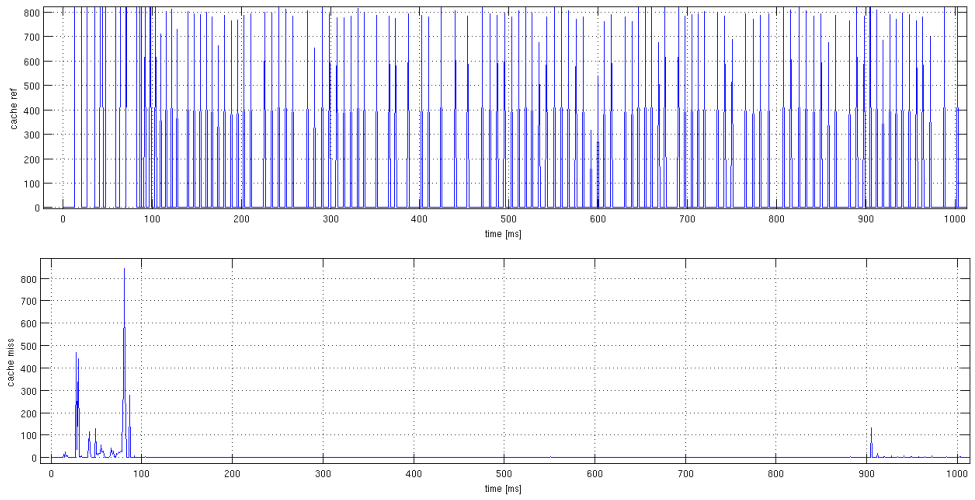

Figure 3.22.: First samples details of cache references and cache miss related to ls profiling

Memory bound

Batch memory bound ad-hoc application are based on the same data structure of the periodic ones, shown in 3.7.1. For readability reasons, the structure of the code is not listed, anyway, task is composed by a $O(n^2)$ cycle, where, at each cycle iteration, two random numbers are extracted, addressing a $bigMem$ element and accessing one of its internal elements for a read or a write. In figure 3.23 first 1000 samples of CPU cycles plot is reported; in figure 3.24 first 1000 samples of cache miss and cache ref trends are reported.

3.7.3. Priority tasks

In this work, what is intended with priority task term are all that applications which have an high level of interactivity with user; we can identify as priority tasks, applications such as text editors, photo editors, and other desktop applications. Since writing a text editor or a photo editor from scratch is quite complicated and not particularly useful for the purpose of this work, priority task profiling has been carried out only on common use applications, in particular, the original intent was to profile some text editors, as kate and kile. These applications are intrinsically I/O bound and both memory and CPU intensive, so no further distinctions have been made.

Problems Initially, it was thought that with Real Time scheduler module and trace-cmd, could be possible to have a good profile, but soon it was evident that prob-

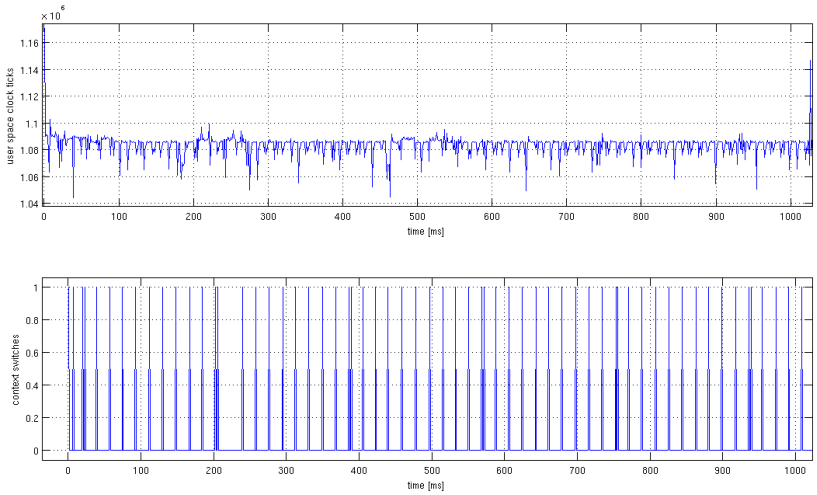

Figure 3.23.: First samples details of CPU cycles plot of a batch memory bound task

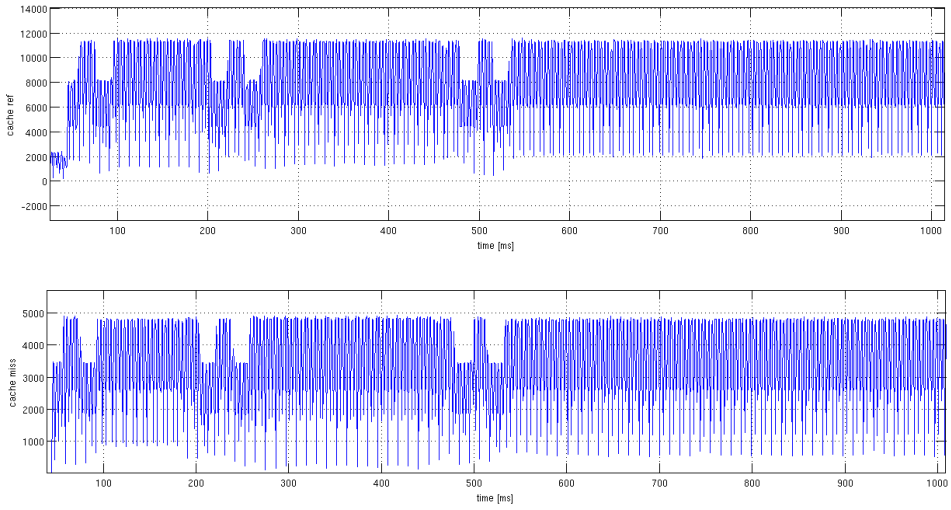

Figure 3.24.: First samples details of cache references and cache miss plot of a batch memory bound task

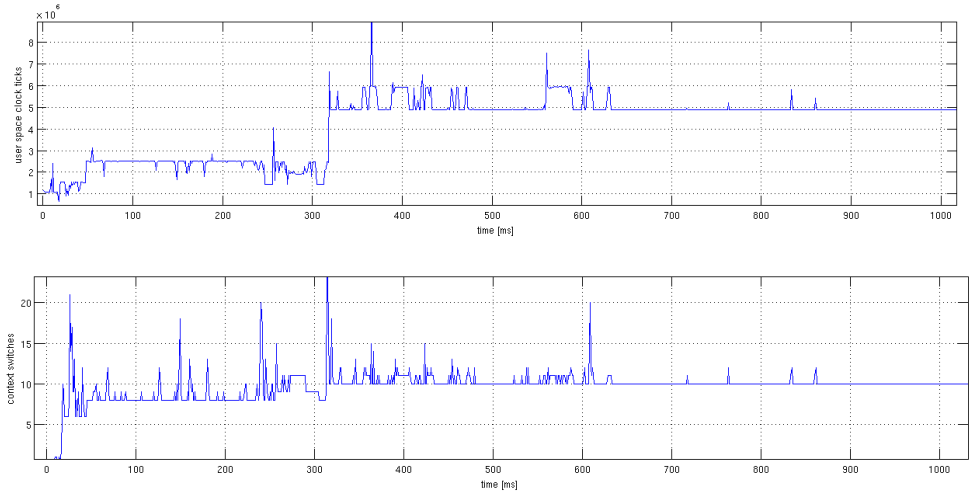

Figure 3.25.: First samples detail related to kate editor, profiled with real time module

lems related to cooperation with graphical interface were worse than expected. Performance counters are not fast enough to profile CPU clock cycles and cache every millisecond. In fact, while using the real-time scheduler module, context switches recorded for each sample are far too many, figure 3.25 shows an example.

Context switches per sample are always much bigger than one; data collected are not reliable.

Also profiling through trace-cmd did not give good results: probably it has a multithreaded implementation and the large amount of events to record creates synchronization problems, so much that, durations of some sleep/wait and of some execution times resulted negative; figure 3.26 shows an example.

Profiling Kile gave even worse results; problems are always similar to those that afflict kate, but more pronounced, so plots are not reported, to improve readability. Probably these text editors are not implemented 'puncturing' xorg and making reserving resources directly from the kernel, so, the amount of events generated is too large to get caught by profiling tools at our disposal.

At this point, to carry out profiling of at least a text editor, it has been chosen a command line text editor: nano. Here are reported some results of executing nan owith no graphical interface, under Complete Fair Scheduler module.

In figure 3.27 is reported a detail of CPU cycles and context switches related to a nano execution. During this execution, a short text has been written and, at the end, buffer has been closed without saving. it is quite evident that even writing fast, task spends most of its time waiting for keyboard input. For this kind of tasks, what is critical is responsiveness, not amount of CPU cycles reserved by system; simplifying, they need little CPU, but they need it quickly.

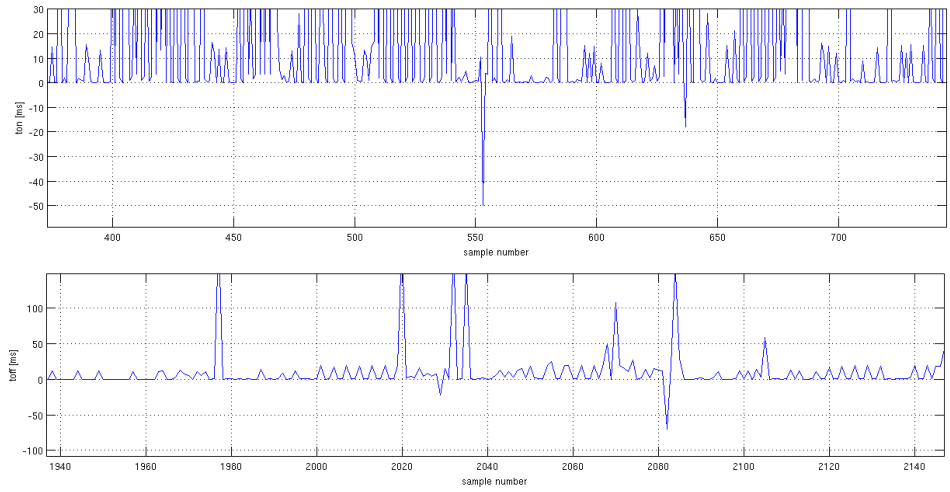

Figure 3.26.: Some negative durations, collected profiling kate with real time module

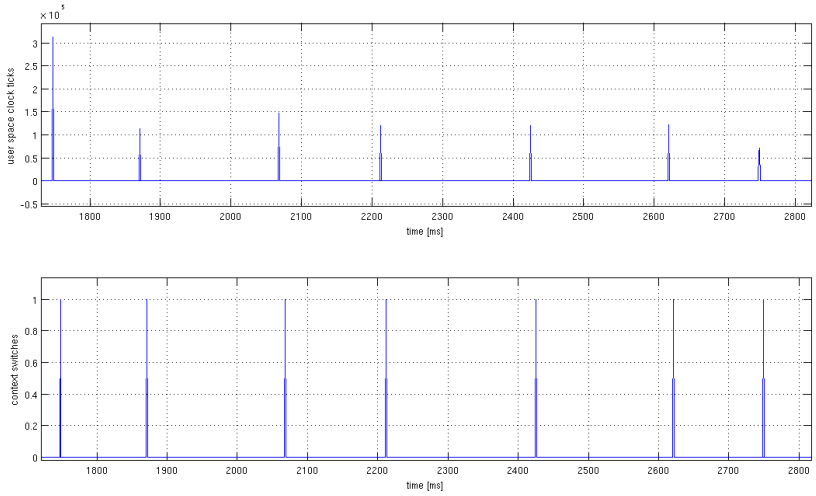

Figure 3.27.: A detail related to nano CPU cycles and context switches profile

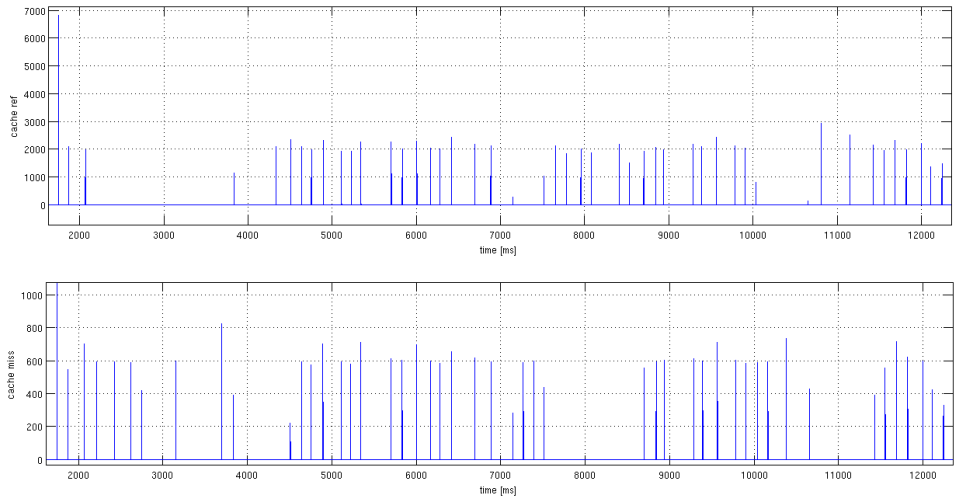

Figure 3.28.: A larger detail related to nano cache references and cache miss profiles

On the other side, even a simple and light editor as nano, need a not inconsiderable amount of memory; in figure 3.28 larger fragments of cache reference and cache miss plots are reported.

3.7.4. Event based tasks

In this section event based task profiling is presented. Event based task are considered all those applications inherently asynchronous, where processing starts as a result of an external event, that could be an interrupt, a remote request or something similar.

Observation 3.7.2. Note that, with this conceptual approach, traditional client-server mechanism can be comprehended in the event based paradigm.

CPU bound

Under event based paradigm a lot of different applications could be included, from the interrupt handler to a client-server application. However, there are some common factors:

- computation starts after that an external event has happened, so, external events frequency is a critical factor
- computation can be short and light (i.e. mouse interrupt handler) or long and heavy (i.e. a ftp request), in both cases, what is more desirable is responsiveness.
- often, event based applications have a parallel structure

In view of these considerations, memory bound and CPU bound distinction has been made. I/O bound case has not been explicitly considered because event based tasks are intrinsically I/O bound. however, it was built a simple client-server application (but not a classic socket server), and the server has been profiled.

CPU bound ad-hoc application has been written with this structure:

- multithreaded: main function is substantially a cycle; for every cycle iteration, a new thread is spawn.
- periodic: it can seem wrong, but we have to simulate in some way that event inter arrivals are not too close. In this way, $main()$ function has been made periodic, 1 second period.
- at each cycle iteration, a pseudo random number n is calculated; then a tread is spawn and n-th prime number is calculated by thread function; $main()$ waits for thread function completion.
- thread function executed is a CPU bound function: it calculates n-th prime number with a $O(n)$ complexity algorithm (it uses successive divisions to compute n-th prime number). When thread function ends, n-th prime number is returned to $main()$ and it waits for period expiration, to go ahead with next cycle iteration

Here are reported algorithms sketches; algorithm 6 presents thread function sketch; that is the function used by new thread, spawn to calculate the n-th prime number by successive divisions.

Algorithm 6 Event based compute_prime() template

Require: arg is a pointer to a integer casted to void
Ensure: a void pointer to the n-th prime number is returned

```
1: void* compute_prime(void* : arg) // Thread function signature, to conform with
   standards, argument is a void pointer and function returns a void pointer to
2: {
3: int n = *((int*) arg); // put *arg value into an int variable
4: while 1 do
5:    int   factor;
6:    int   is_prime = 1;
7:    for (factor = 2; factor < candidate; ++factor) do
8:       if (candidate%factor == 0) then
9:          is_prime = 0; // If factor divides candidate, candidate is not prime
10:         break; // candidate is a global variable
11:      end if
12:      if is_prime then
13:         if --n == 0 then
14:            return (void*)candidate // Decrement n, if it is 0, we have the n-th
               prime
15:         end if
16:         ++candidate // Increment candidate
17:      end if
18:   end for
19:   return NULL
20: end while
21: }
```

algorithm 7 presents a sketch of $main()$ function. it is quite simple, basically it starts profiler, set period (1 second) and initializes a quite high random number (from 0 to 4000) and a bias (bias is about 3000); after that a cycle begins; we have a cycle iteration for each simulated external event to process (for identification, at least 500). At each loop iteration, candidate restarts from 2, a new thread is spawn and executes thread function, passing as parameter the n-th prime number to compute. prime to be computed is put in `which_prime` variable, and is calculated as a sum of bias and random number extracted. Bias has been introduced to avoid too fast computations due to very low numbers randomly extracted (i.e. 2,5,10 etc..)

Algorithm 7 Event based main() template

```
1: pid_t pid = getpid();
2: start_profiler(pid); // Start profiler
3: .
4: .
5: int * prime; // Pointer to a int, where thread function put result
6: int which_prime, rnd1;
7: int steps = 500; // Number of events to simulate
8: int bias = 3000; // Bias to calculate not too small numbers
9: int i = 0; // Counter
10: unsigned int period;
11: .
12: .
13: // Make main() function periodic, exploiting make_periodic() function, already
    introduced
14: while (i < steps) do
15:   rnd1 = rand_lim(4000); // rnd1 is a random number between 0 and 4000
16:   which_prime = bias + rnd1;
17:   pthread_create(&thread, NULL, &compute_prime, &which_prime); // Spawn
      a new thread
18:   pthread_join(thread, (void **) &prime); // Wait for thread to complete, and
      get the result
19:   .
20:   .
21:   // Print n-th prime number
22:   wait_period(&info); // Wait for period expiration
23: end while
24: return NULL
```

Figure 3.29 reports CPU cycles and context switches; figure 3.30 reports cache reference and cache miss plots. Event based tasks plots resemble, in some way, priority tasks plots, exposed in section 5.4.4. The main difference is event based tasks regularity; this because we made them periodic, to ensure a bit of waiting, from an event to the other; moreover, probably prime number to compute are all quite similar or small (nevertheless, a random generator has been used) and task needs a quite short time to complete computation (about 200 ms). This kind of tasks has been reported for completeness of exposition, but they did not result very useful during identification phase (from a certain point of view, they are a particular case of periodic tasks). Instead, client-server application, both CPU and memory bound have been more useful during identification.

In priority tasks, sleep/wait time are more variable; when user is writing or doing something, they are quite short, but sometimes, when user stops, we have quite big peaks (2 or 3 seconds).

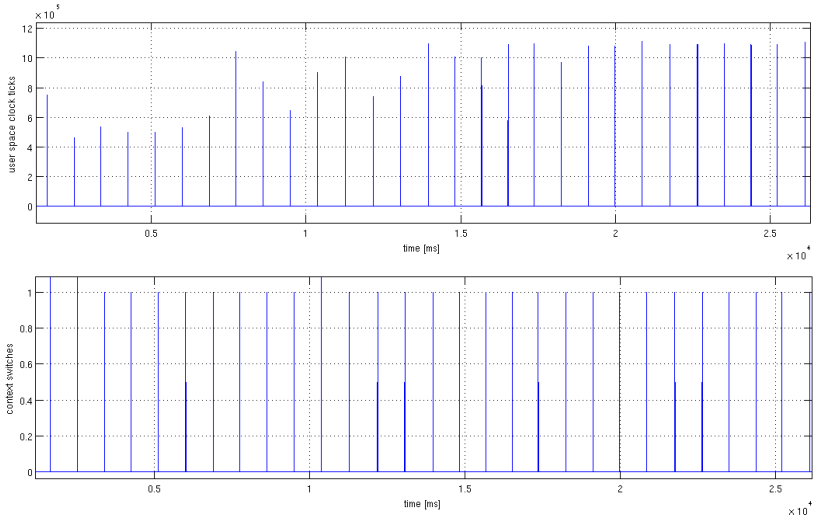

Figure 3.29.: CPU cycles and context switches plot of an event based task

Anyway, figure 3.31 shows comparative plot between event based task and a priority task. Event based toff is the red trace; nano toff is the blue one.

In the next figures is presented what happens in a very simple client-server application; Server waits to get a number n by command line (standard input); when number is inserted, a new thread is spawn and the $n-th$ prime number is calculated and printed on the shell (standard output).

In figure 3.34 is reported a brief plot comparison of sleeping/waiting time related to nano (blue trace) and our event based interactive task (red trace). Now, event based task toff fluctuates much more, than the toff measured on the event based task CPU bound reported in figure 3.31. Indeed, now, toff depends substantially on how much time passes between the insertion of a number and the next by the user, and sometimes it could be a long time.

Wait/sleep time could be elevated also for a priority task; figure 3.35 shows an example. In this figure a long wait is remarked. that is probably due to a long waiting by user, before going on writing text. In that case, editor goes in wait status and it is not executed until a new keyboard interrupt happens. When interrupt happens, it is processed and then execution goes on. Interrupts are usually not very heavy, from CPU point of view, but is a common desire that they are served quickly, to ensure responsiveness. This is the reason why priority tasks are not particularly CPU critical, but the small amount of CPU they need, should be given with low latency.

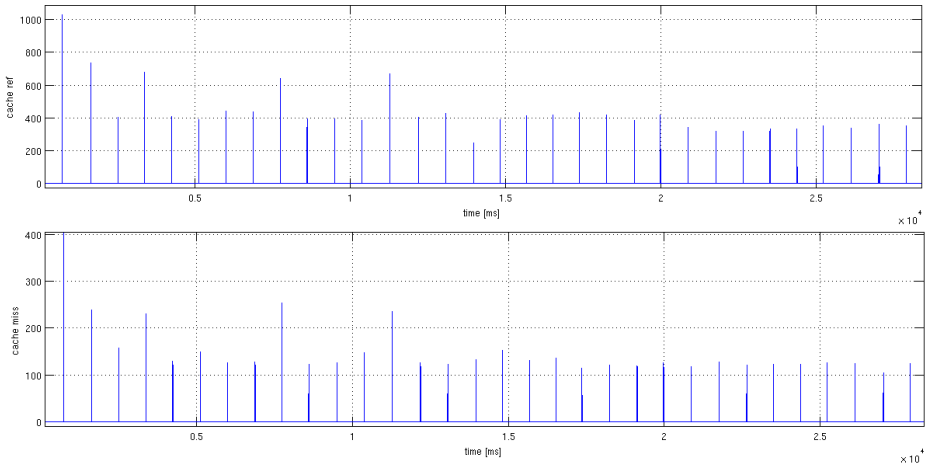

Figure 3.30.: Cache reference and cache miss plot of an event based task

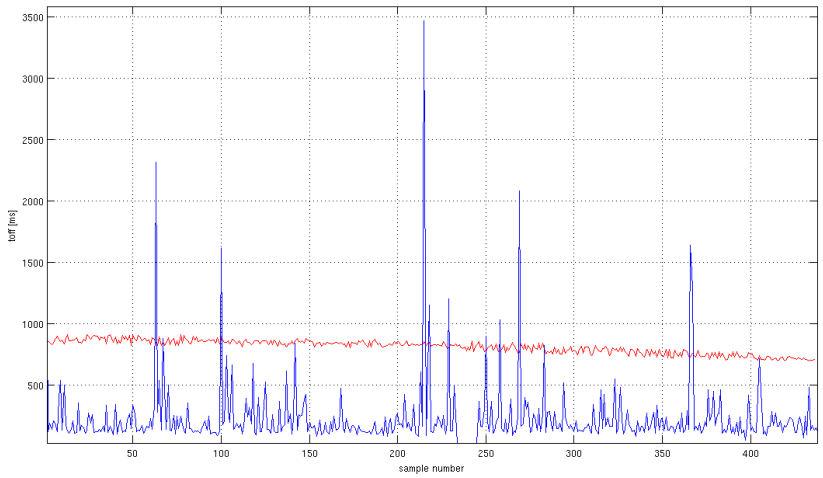

Figure 3.31.: Comparison between nano toff and event based toff; nano trace is the blue one, event based trace is the red one

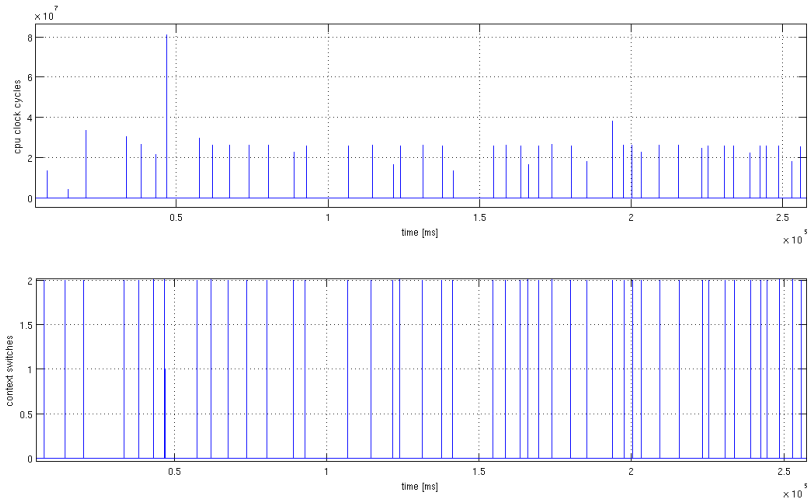

Figure 3.32.: CPU cycles and context switches plot of an event based interactive task

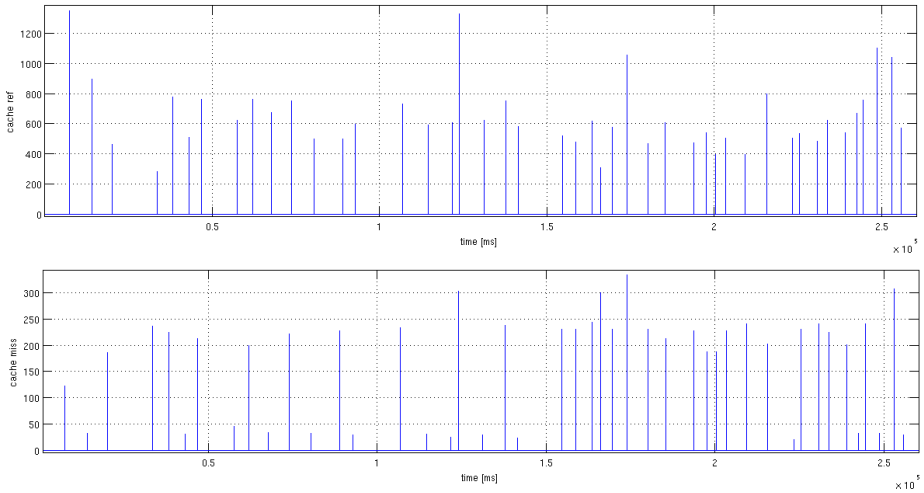

Figure 3.33.: Cache references and cache miss plot of an event based interactive task

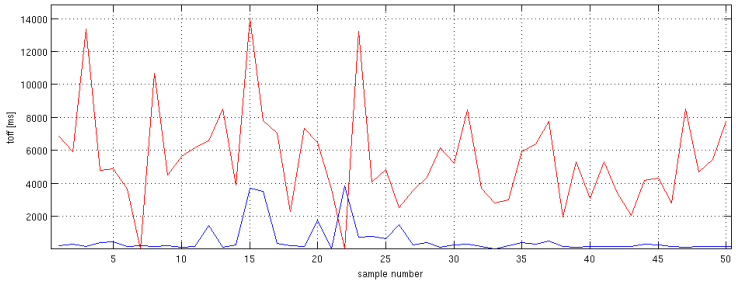

Figure 3.34.: Comparison between nano toff and event based server toff: nano trace is the blue one; event based task trace is the red one

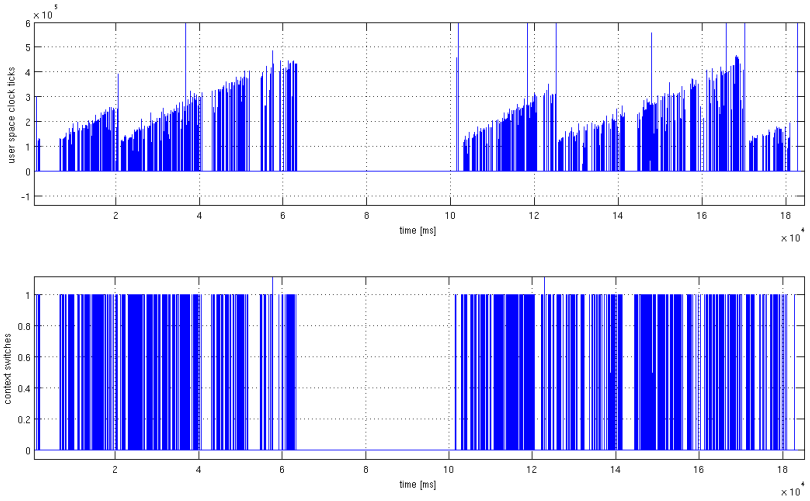

Figure 3.35.: CPU and Context switches plot of a nano execution with a long wait

Memory Bound

Memory bound tasks have the same structure of interactive event based application described in the previous paragraph; server waits for a number on a blocking $scanf()$; every time user inserts a number, server spawns a new thread. The main difference compared to the previous case, is that new thread allocates a lot of memory in the heap and then it does random read/write accesses in this memory area. in algorithm 8 memory bound thread function is sketched.

Algorithm 8 Event based memory bound thread function template

```
1: void* memory_scan(void* arg)
2: {
3: bigMem* data[1000]; // Declare an array of 1000 bigMem pointers;
4: // Each bigMem element contains an array of 8000 long int elements
5: int* ptr;
6: int rnd1, rnd2, j, i;
7: int write = 42;
8: int read; // Support integers
9: .
10: .
11: .
12: // Init data structure array to 0 and starts random generator
13: for (i = 0, i < 10000, i++ ) do
14:    rnd1 ← rand_lim(1000); // Do 10000 accesses; Put a random number between
       0 and 1000 in rnd1
15:    for (j = 0, j < 10000, j++ ) do
16:       rnd2 ← rand_lim(8000); // extract a random number between 0 and 8000;
17:       // If rnd1 is even, do a write, else do a read
18:       if (rnd1%2 == 0) then
19:          data[rnd1] → field[rnd2] = write;
20:       else
21:          read = data[rnd1] → field[rnd2];
22:       end if
23:    end for
24: end for
25: // Return a void pointer for portability and compilation issues, then terminate
26: }
```

Next figures report a detail of CPU cycles, cache reference and cache miss of a single input execution.

Event based memory bound task has been implemented, but, watching profiling results (and also from a conceptual point of view), it does not need a dedicated model; in fact, it is very similar to a batch application executed n times, where n is the number of input given by the user. From this point of view, a client-server memory bound

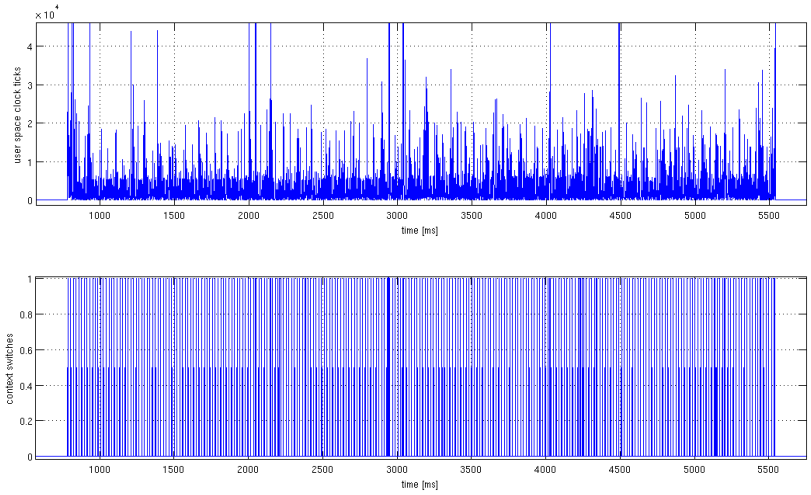

Figure 3.36.: CPU and Context switches plot of a single input execution

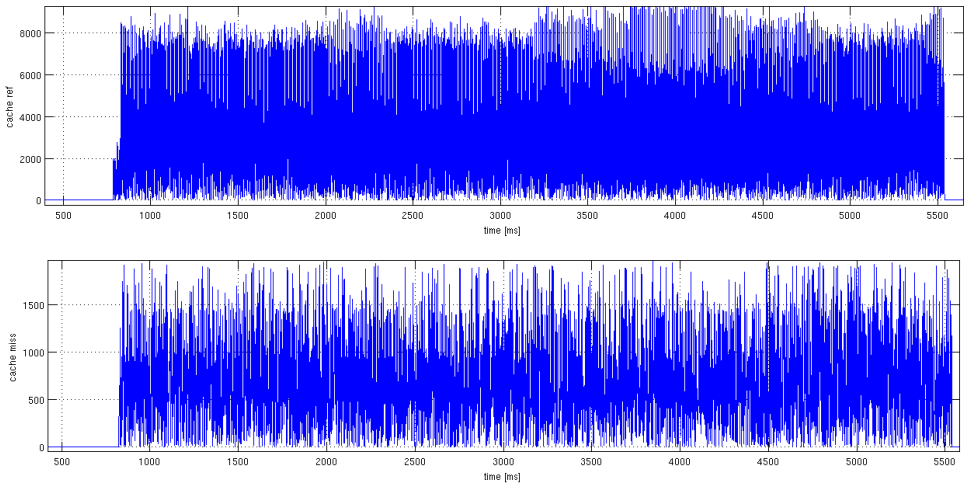

Figure 3.37.: Cache references and cache miss plot of a single input execution

computation, can be seen as a batch memory bound task which is launched every time that a user forwards a request. For this reason, in part 4, event based memory bound tasks are not explicitly modeled.

Data logging and results representation All profiling data has been saved in comma separated values files. These files have been imported, filtrated, plotted and analyzed using Matlab framework. Plotting data and observing it from both an high level and detailed point of view helped to understand task behavior and notice eventual experiment setup errors. For example, at the very beginning of profiling work, some tasks have been profiled without killing xorg but, plotting profiles, it turned out that context switches for sample were too high, so memory and computational system load had to be reduced. Matlab also helped us finding out that graphical text editors were not suitable for profiling, because plotting data retrieved by performance counters and trace-cmd tool pointed out some structural unavoidable problems. Last but not least, Matlab has a powerful system identification tool, which it was essential to carry out the identification work, discussed below. Profiling work has confirmed expectations about task behavior. From a qualitative point of view, cache miss, cache reference and CPU clock cycles (so also ton and toff), have the temporal trend expected, especially for common use applications.

4. Task model identification and validation

This chapter presents the work related to the identification of a task model, starting from data [16, 7] obtained during profiling (profiling details are described in section 3.7).

Work described here is particularly critical, because the quantities profiled are not produced by a physical system having a structure or, at least, a representation known a priori. Moreover, the same quantity (i.e. cache references) has a very different behavior, according to the different classes of the task analyzed. The models identified are meant to develop a control structure able to handle load balancing on multi core architectures, so they will be implemented in Multi Core Scheduler simulator, described in chapter 5, to favor the development of a suitable load balancing *controller*,exploiting, at the same time, work already present in literature to address scheduling on each single independent core.

These are some reasons why each model have to be general enough, not only to fit different execution traces of the same task, but also to fit execution traces related to different tasks belonging to the same class. This has led to validate the models in Fourier transform domain, verifying that Fourier transform of signals produced by identified models is similar to the one of profiled data or, in case of Input-Output models, that frequency response of the model is similar to the one of the data set.

This is not, however, an easy task, because, as it will be better explained later, data have a quite complex behavior, and results of identification attempts are not always clean and elegant as one would like them to be.
To achieve better results, profiling data have been windowed, cleaning them up from the initial set up phase, when task is executed for the first time (except ls profile); during this phase, very different tasks (even belonging to different classes) have a similar behavior (quite similar to batch tasks behavior), invoking syscalls, loading data and code on memory etc. This behavior, in many cases, is quite different from task's 'normal' one, so it has been removed, to don't polarize the identification process. Moreover, to identify a model, each data set has been detrended and divided into two parts: one part has been used to estimate model, the other one to validate the model, using cross validation approach, since original data set were big enough to allow this approach.

Here follows a table which summarizes which classes of profiled tasks have been analyzed and which quantities of interest (profiled signals) have been identified.

- **ok** means that the model has been identified;

Task class (application)	Ton	Toff	Cache miss	Cache references
Periodic (Mplayer)	ok	ok	ok	ok
Batch (C)	n.d.	n.d.	no	ok
Batch (Python)	n.d.	n.d.	ok	no
I/O bound (ls)	ok	ok	ok	ok
Priority (nano)	no	no	ok	ok
Event (interactive)	ok	ok	ok	ok

Table 4.1.: Summary table of identified models

- **no** means that the identification process was not successful (detailed reasons will be explained later in the specific task class section);
- **n.d.** means that the signal has no meaning for the specific task class (i.e. ton and toff have no meaning for a batch task which ideally, would execute with no interruption from beginning to termination);

4.1. Time on and time off identification

4.1.1. Toff identification for periodic tasks

To identify a model related to periodic task class, the data obtained by Mplayer profiling, exposed in section 3.7.1, have been used. This decision has been taken because video player is a common use application which has natural periodic behavior and that stresses both CPU and memory subsystem. Luckily, Mplayer implementation enables collection of good quality profiling data. Infact, ton and toff of a periodic application are roughly constant (their fluctuation depends mainly on the complexity of the reproduced frame), and this helps on achieving good identification results.
In figure 4.1 is reported toff periodogram [1] of two data series: toff inferred by CPU cycles measured by profiling library and toff measured by trace-cmd tool.

The behavior of the two data series in Fourier transform domain is quite similar, so, both have been used to make model identification. It has been held the model which presented the best fit. The state space model derived from trace-cmd data series has proved having an acceptable interpolation (in terms of estimated noise spectrum, according to the model block schema reported in figure 4.3).

Toff and ton data sets are time series, so, to identify suitable models, the approach represented in the block schema reported in figure 4.3 has been adopted. This model schema has been adopted in general for toff and ton identification, not only on the periodic tasks case.

Figure 4.4 shows power spectrum of data produced by model, compared to the power spectrum of validation data set.

[1] periodogram is the normed absolute square of the Fourier transform of the data.

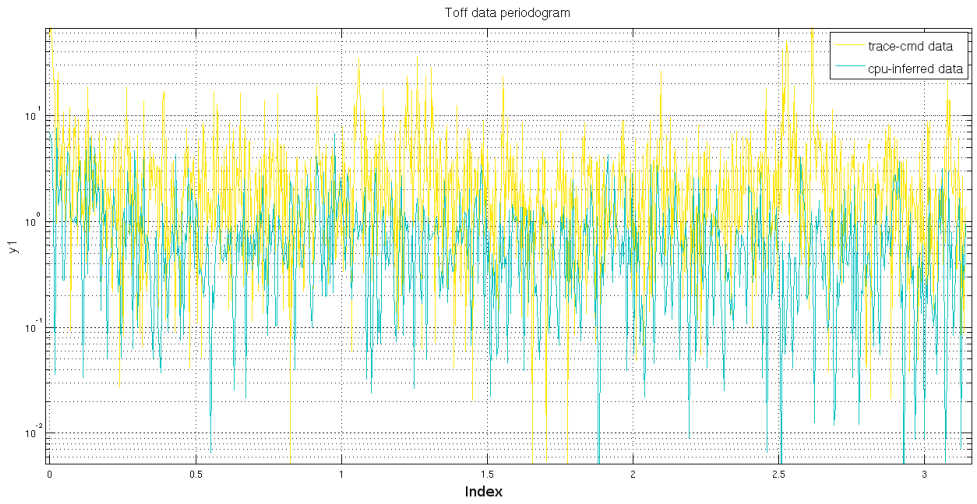

Figure 4.1.: toff periodogram of data inferred by CPU cycles measured by profiling library (cyan trace) and toff periodogram of data measured by trace-cmd tool (yellow trace)

Figure 4.2 shows the model output residuals autocorrelation; interval reported is 99% confidence interval, so residuals can be considered white.

For convenience, let's call the state model vector $x(t)$, the noise input signal $e(t)$ and the model output $y(t)$. where $e(t) = H(z)in1(t)$ is a filtered white noise (a colored noise). Noise spectrum is given by the formula

Definition 4.1.1.

$$\Phi_e(w) = \lambda \left| H(e^{jw}) \right|^2$$

and it characterizes the output data power spectrum.

The state space model has the following structure:

$$x(t+1) = Ax(t) + Ke(t)$$

$$y(t) = Cx(t) + e(t)$$

Where

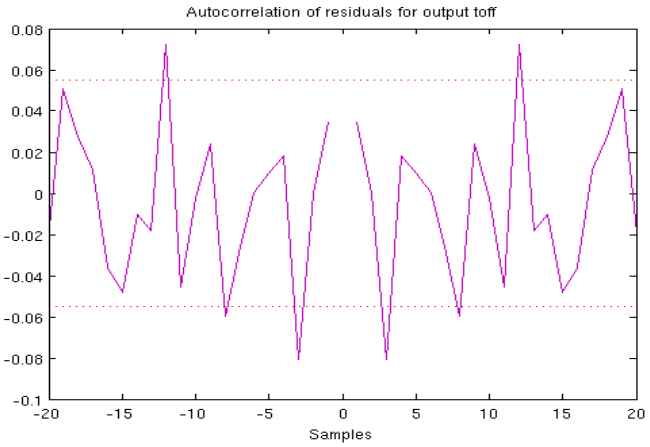

Figure 4.2.: off model output residuals autocorrelation

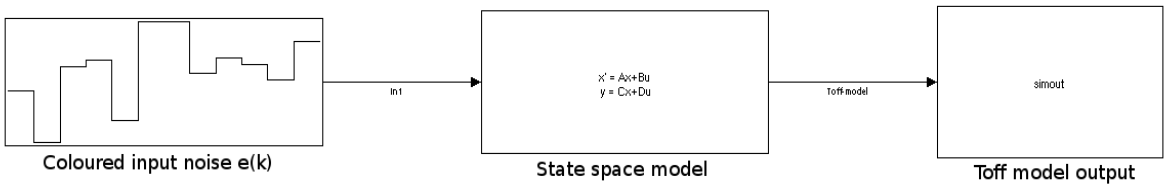

Figure 4.3.: toff state space model schema

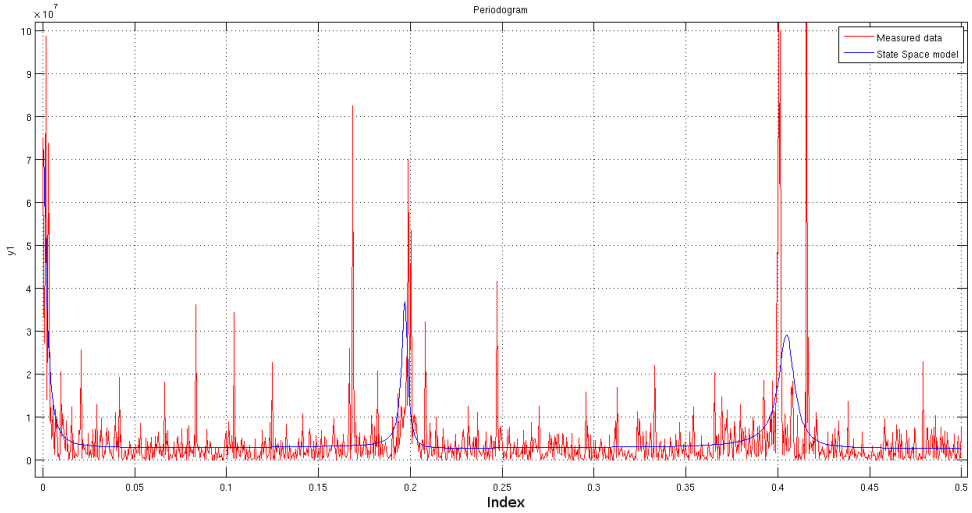

Figure 4.4.: toff state space model in Fourier transform domain

A =

	$x1$	$x2$	$x3$	$x4$	$x5$	$x6$
$x1$	0.98322	0.07196	0.04423	−0.02662	0.00458	0.02164
$x2$	0.04864	−0.82168	0.52474	−0.09456	−0.19205	−0.04238
$x3$	−0.00606	−0.41741	−0.57365	−0.58350	0.34966	0.11492
$x4$	−0.01785	−0.05423	0.31083	0.29636	0.89847	0.00856
$x5$	−0.01174	0.36117	0.42365	−0.74100	0.14495	−0.14128
$x6$	0.00177	0.00625	0.00900	0.01398	−0.00950	0.61084

C =

	$x1$	$x2$	$x3$	$x4$	$x5$	$x6$
$y1$	40914	−11612	−53998	30836	19946	−24821

K =

	$y1$
$x1$	$1.2238 * 10^{-6}$
$x2$	$-1.2096 * 10^{-6}$
$x3$	$1.1323 * 10^{-6}$
$x4$	$6.0076 * 10^{-6}$
$x5$	$-1.6404 * 10^{-6}$
$x6$	$2.8898 * 10^{-7}$

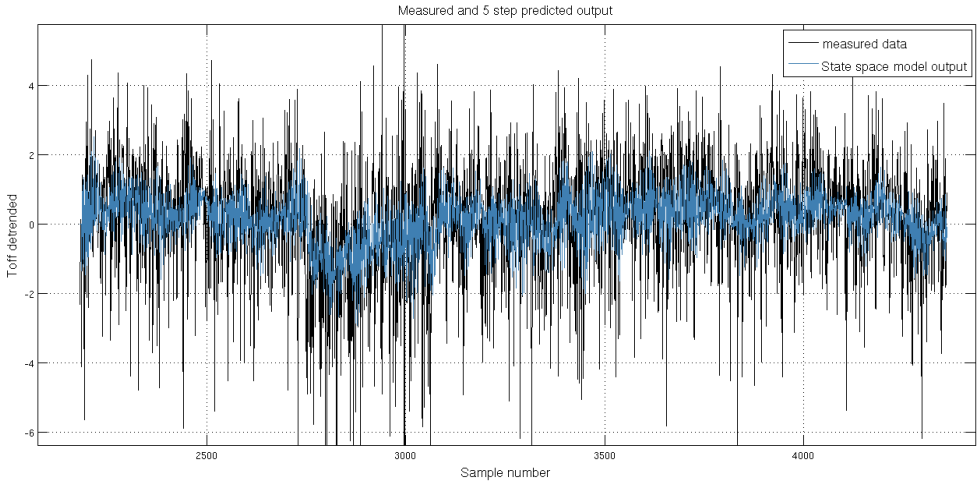

Figure 4.5.: toff state space model time fitting

Initial conditions:

x(0) =

$$\begin{array}{ll} x1 & -0.039941 \\ x2 & 0.010294 \\ x3 & 0.003551 \\ x4 & -0.028165 \\ x5 & -0.020146 \\ x6 & -0.002447 \end{array}$$

The model was estimated using Matlab identification toolbox (n4sid, a subspace algorithm [18]); it is a sixth order model, with an input **in1** and an output **toff**. it is a reasonable simple model and gives satisfactory results. Figure 4.5 shows time fitting of data produced by a five step predictor based on the state space model:

The model does not overfit on the single execution trace and the model output temporal trend is quite similar to the measured data temporal trend.

4.1.2. Ton identification for periodic tasks

Also for ton analysis, the data set chosen to carry out identification process was the one related to Mplayer profiling, for the same reasons explained in toff identification section. Ton are characterized by a 'more complicated' frequency content, substantially because ton related to a video stream playback depends on actual frame decoding complexity. To identify this model, it has been applied the same procedure used for toff model identification. Both data sets obtained by trace-cmd tool and by pro-

filing library were used to identify a model. Also in this experiment, data measured by trace-cmd have allowed us to obtain the best model, and it is again a state space model. It has been identified using Matlab identification toolbox (n4sid algorithm). Figure 4.3 shows block schema representation related to the model. For convenience, let's call the model state vector $x(t)$, $e(t)$ is a white noise input signal and $y(t)$ is the model output.

The identified model has the following structure:

$$x(t+1) = Ax(t) + Ke(t)$$
$$y(t) = Cx(t) + e(t)$$

where $e(t) = H(z)in1(t)$ is a filtered white noise (a colored noise). $e(t)$ spectrum is given by the formula 4.1.1 reported before.

A =

	$x1$	$x2$	$x3$	$x4$	$x5$
$x1$	-0.69886	0.64777	0.28810	-0.01953	0.02202
$x2$	0.12078	0.50144	-0.85654	0.02595	-0.02476
$x3$	-0.69773	-0.53241	-0.42441	-0.04451	0.02505
$x4$	-0.02842	0.02169	0.02916	0.30019	-0.93685
$x5$	-0.02972	-0.01010	0.00620	0.94781	0.31357

C =

	$x1$	$x2$	$x3$	$x4$	$x5$
$y1$	13.49	34.64	15.62	-13.29	23.58

K =

	$y1$
$x1$	0.00220
$x2$	0.00078
$x3$	-0.00348
$x4$	-0.00291
$x5$	-0.00051

Initial conditions:

x(0) =

$x1$	-0.00948
$x2$	-0.05615
$x3$	0.01578
$x4$	-0.00208
$x5$	-0.00128

Figure 4.6 shows power spectrum of the model output and of the validation data

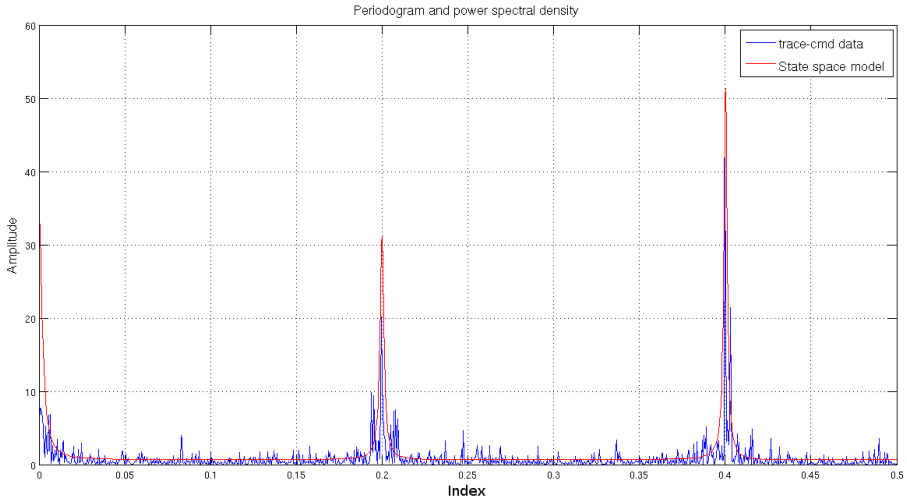

Figure 4.6.: ton state space model power spectrum

set; the model has a good Fourier transform domain behavior. Also time fitting results are quite good, reproducing temporal trend and avoiding overfitting. Figure 4.7 shows time fitting of the data produced by a five step predictor based on the state space model.

Figure 4.8 shows the model output residuals autocorrelation; 99% confidence interval is remarked. Residuals can be considered uncorrelated.

4.1.3. Toff identification for 'batch' I/O bound tasks

In this section, identification process is carried out for 'batch' I/O bound tasks, according to the considerations made in section 3.7.2. To represent this class of tasks, ls application has been chosen and profiled. Here are reported the model obtained by toff data set, in the next section the model obtained from ton data set will be discussed.

Figure 4.9 shows toff data power spectrum, compared to the ones of different signals, produced by some different models.

The state space model behavior in time domain is quite good. Figure 4.10 shows that the model represents quite well the main temporal trend spotted by validation data set.

The state space model has the usual structure:

$$x(t+1) = Ax(t) + Ke(t)$$

$$y(t) = Cx(t) + e(t)$$

where $e(t) = H(z)in1(t)$ is a filtered white noise (a colored noise). $e(t)$ spectrum is

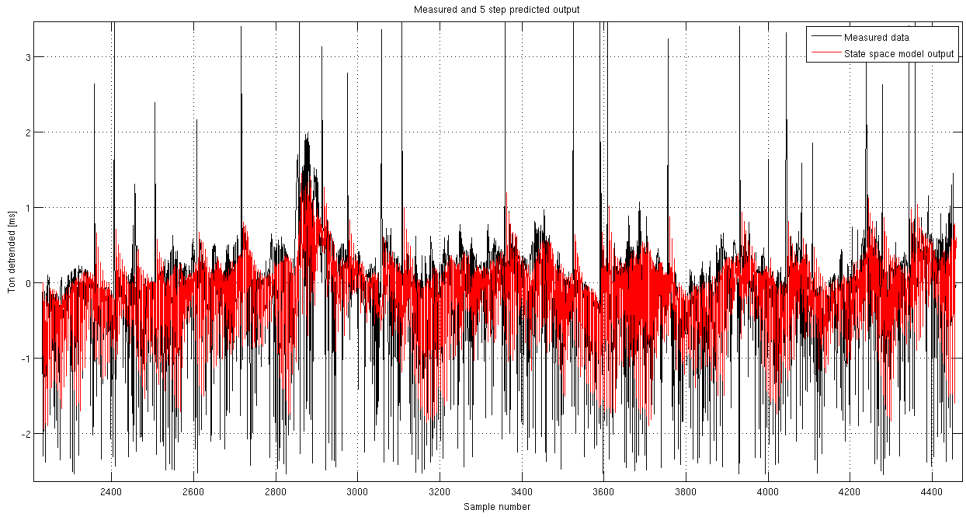

Figure 4.7.: ton state space model time fitting

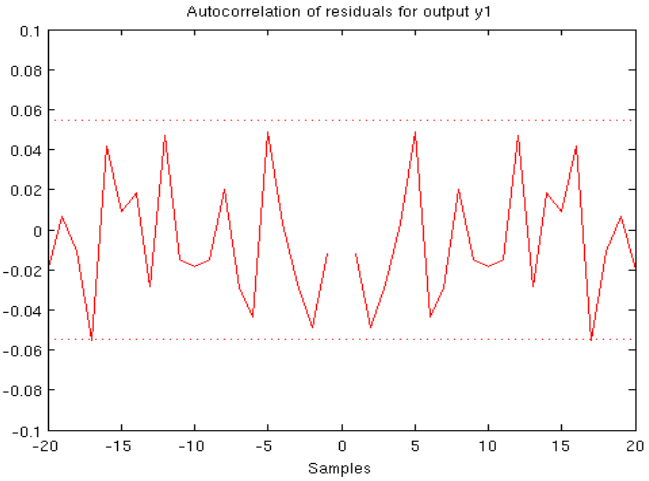

Figure 4.8.: ton model output residuals autocorrelation

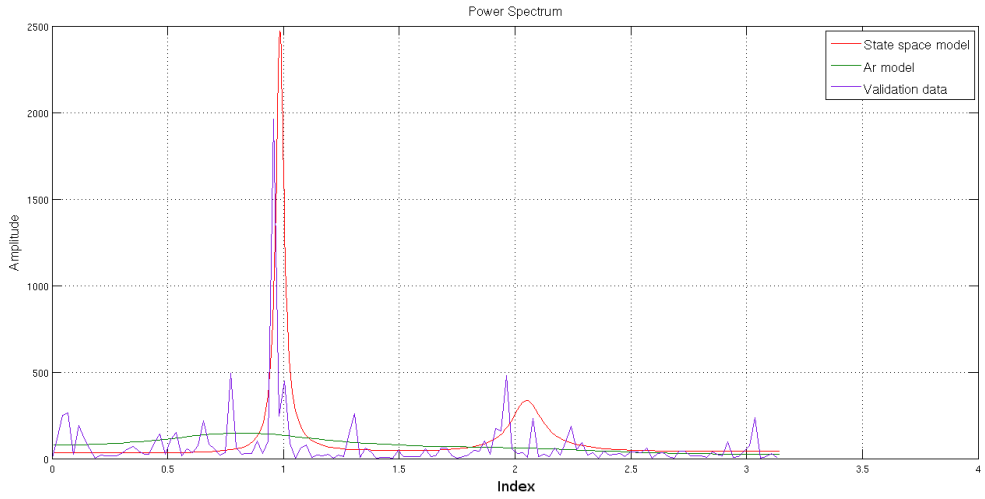

Figure 4.9.: toff data power spectrum of an ls execution trace compared to model ones

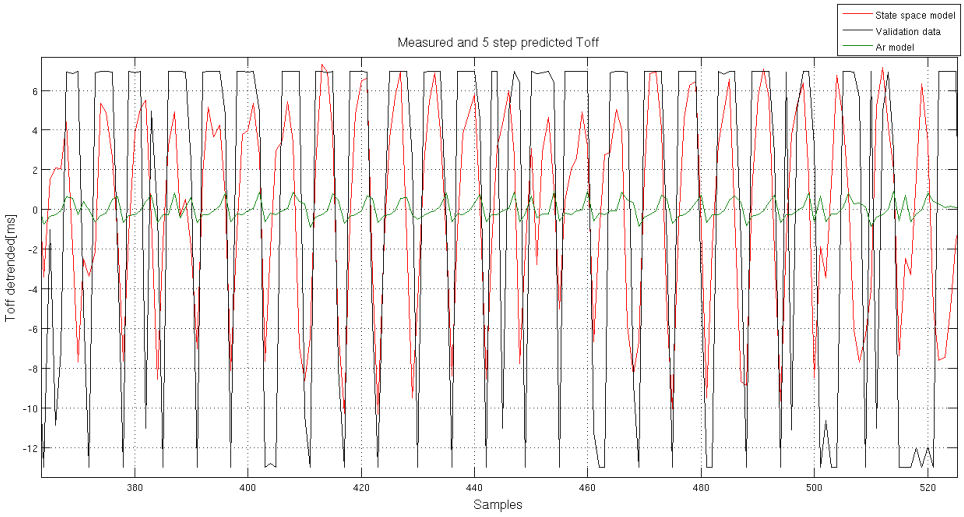

Figure 4.10.: toff time fitting related to the ar model and to the state space model

given by the formula 4.1.1 reported before.

Model matrices are reported here:

A =

	$x1$	$x2$	$x3$	$x4$
$x1$	0.56061	0.81571	−0.01586	−0.00153
$x2$	−0.81325	0.52590	−0.08115	0.04857
$x3$	0.00386	0.04736	−0.35871	0.78999
$x4$	0.00386	0.02195	−0.85367	−0.48741

C =

	$x1$	$x2$	$x3$	$x4$
$y1$	56.515	−78.095	−46.873	39.427

K =

	$y1$
$x1$	−0.001447
$x2$	−0.001698
$x3$	0.002889
$x4$	0.001221

x(0) =

$x1$	−0.056511
$x2$	−0.019984
$x3$	0.071438
$x4$	−0.013319

The model was estimated using Matlab toolbox (n4sid algorithm).
Figure 4.11 shows the toff model residuals autocorrelation. Residuals can be considered uncorraleted and prediction error is white with confidence level of 99% .

4.1.4. Ton identification for 'batch' I/O bound tasks

In this section identification of a ton model related to I/O bound 'batch' task class is performed. As explained before, the application chosen to represent this class of tasks is ls. As happened with other profiles, the data obtained by trace-cmd tool and the ones obtained by profiling library was quite similar, so it was chosen the model who produced the best fit in Fourier transform domain. Data set produced by trace-cmd, as in the previous cases, gave the best model. This happened quite often and it is easily explained by the fact that trace-cmd measured are made by time stamping, using kernel level high resolution clocks (which have nano second resolution). Hardware counters exploited by profiling library are fast, but, however, they have milli second resolution, so toff and ton data obtained using profiling library are inevitably

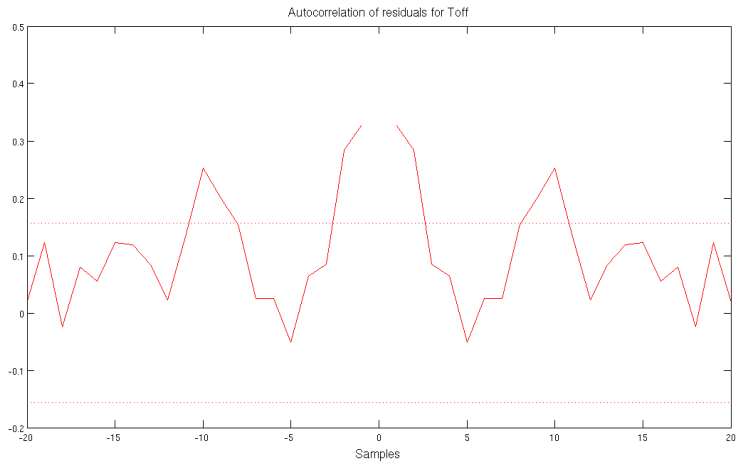

Figure 4.11.: toff model output residuals autocorrelation; 99% confidence interval

less precise.

The state space model estimated using Matlab toolbox has given acceptable results; figure 4.12 shows a comparison between the validation data power spectrum, and the model output signal power spectrum.

As in previous cases, this state space model represents a time series, so, the model structure is:

$$x(t+1) = Ax(t) + Ke(t)$$

$$y(t) = Cx(t) + e(t)$$

where $e(t) = H(z)in1(t)$ is a filtered white noise (a colored noise). $e(t)$ power spectrum is given by the formula 4.1.1 reported before. In the figure is compared power spectrum of $e(t)$ with the validation data power spectrum.

Model matrices are now exposed:
A =

	$x1$	$x2$	$x3$	$x4$	$x5$
$x1$	0.53128	0.80200	−0.02920	0.18018	−0.03100
$x2$	−0.75427	0.51491	0.25160	−0.20969	0.16584
$x3$	−0.07218	−0.15932	0.55650	0.72872	−0.16968
$x4$	−0.03076	0.11633	0.13265	−0.20048	−0.94512
$x5$	−0.18695	−0.05815	−0.65692	0.23111	−0.21707
$x6$	−0.03424	−0.07355	−0.10666	−0.04601	−0.29466
$x7$	−0.02414	0.04350	−0.03558	0.16315	−0.04546
$x8$	0.05694	−0.05682	0.12314	−0.18791	0.03534

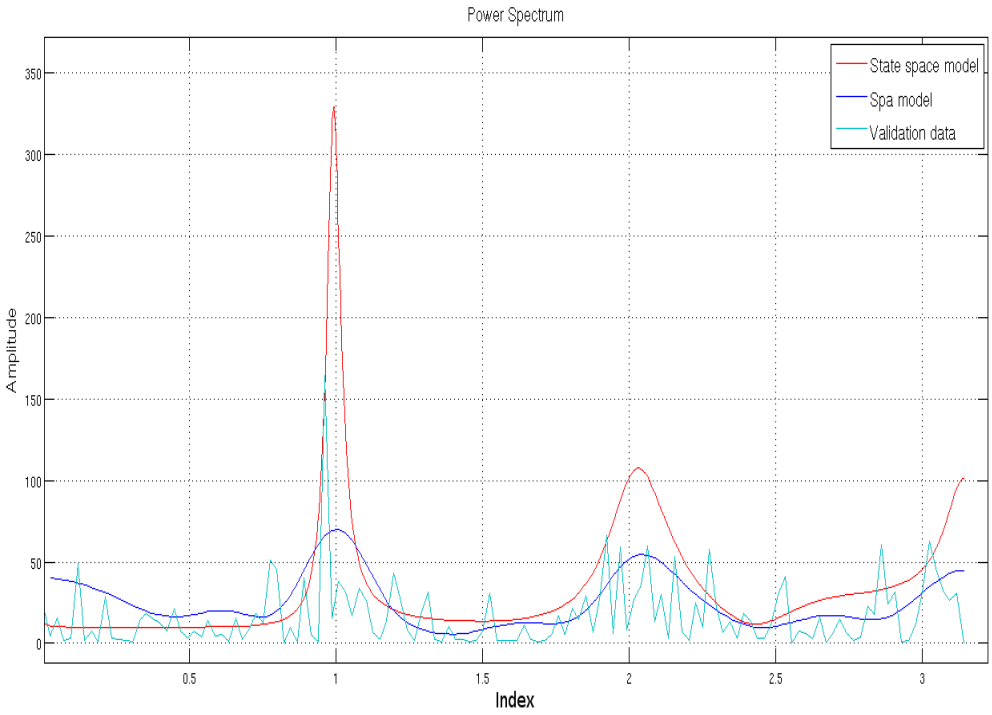

Figure 4.12.: ton data spectrum of validation data set compared to the one of the state space model

	$x6$	$x7$	$x8$
$x1$	0.02326	0.00384	0.02181
$x2$	$1.92e-5$	−0.08590	−0.15863
$x3$	0.12634	0.04092	−0.13397
$x4$	0.22278	0.09394	0.10903
$x5$	0.43725	−0.07265	−0.26296
$x6$	−0.83871	−0.10369	−0.19724
$x7$	−0.03537	−0.84307	0.36027
$x8$	0.10626	−0.50028	−0.41170

C =

	$x1$	$x2$	$x3$	$x4$	$x5$
$y1$	31.270	−30.812	21.713	−35.121	−0.514

	$x6$	$x7$	$x8$
$y1$	13.288	−3.5793	0.731

K =

	$y1$
$x1$	−0.003994
$x2$	−0.001888
$x3$	−0.002904
$x4$	0.002711
$x5$	−0.003557
$x6$	−0.003065
$x7$	0.001203
$x8$	−0.001213

x(0) =

$x1$	−0.099478
$x2$	−0.169450
$x3$	0.244040
$x4$	0.272980
$x5$	−0.092217
$x6$	0.184390
$x7$	−0.272850
$x8$	0.192680

Figure 4.13 shows the model's time domain behavior:

The identified model is able to grasp the major temporal trends that characterize the validation data set.

Finally, residuals autocorrelation has been investigated. It is reported in figure 4.14 highlighting 99% confidence interval.

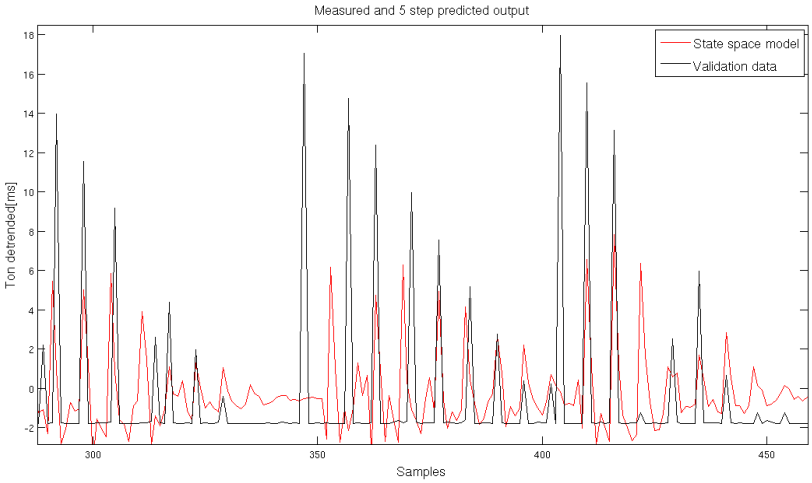

Figure 4.13.: ton state space model time fitting

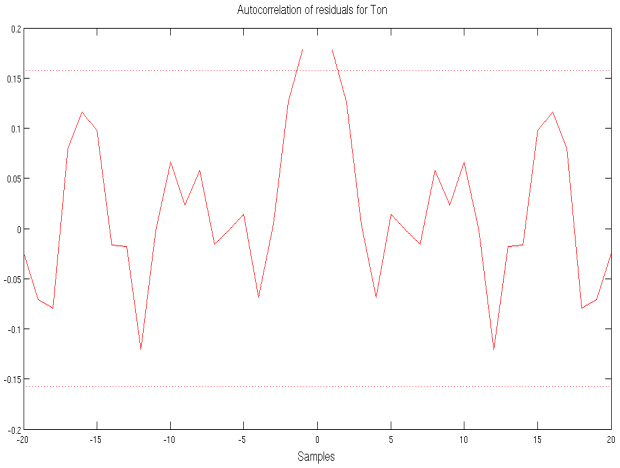

Figure 4.14.: ton autocorrelation of residuals, 99% confidence interval

4.1.5. Ton and toff identification for priority tasks and batch tasks

Unfortunately, ton and toff related to priority tasks depend on the time that user needs to insert single characters; this time depends on a lot of parameters that can hardly be estimated, as how long is the word, how far are single characters on the keyboard and on what user is thinking about. For these reasons (and maybe others), ton and toff processes related to priority tasks are not stationary, so no model can be identified with linear, time invariant approach. About batch tasks, instead, question is quite different; they are effectively preempted by scheduler, so ton and toff data sets related to batch tasks exist, but these data sets do not model task behavior, but scheduler behavior. In fact, a batch task, on an ideal machine would execute with no interruptions, until its termination, so, its toff model is zero, and its ton model is total task duration.

4.1.6. Toff identification for event based tasks

In this section is discussed identification of a toff model from event based task profiles. As discussed in section 3.7.4, event based tasks are intrinsically I/O bound and, usually, computation is quite light. Task used to represent this situation, waits a number n in input and then calculates $n-th$ prime number by a successive divisions algorithm. In this case, light computation is simulated inserting small numbers (but sometimes, some big ones are inserted, to simulate some heavier request to satisfy; this impacts mainly on ton model). A priori, also this model can be affected by the problem of non stationarity faced during priority toff model identification procedure. Probably here, time variance is not so heavy; number inserted are quite small and similar, and also time intervals elapsed are not so dramatically different. Anyway, modeling is not easy, in fact the best model identified is a first order model, symptom that some dynamics were missed, but, as figure 4.15 shows, the most important ones were modeled, leaving other dynamics in the noise model,

The model with the best fitting properties is derived from ton time series profiled by trace-cmd tool, and it has the usual structure:

$$x(t+1) = Ax(t) + Ke(t)$$
$$y(t) = Cx(t) + e(t)$$

where $e(t) = H(z)in1(t)$ is a filtered white noise (a colored noise). $e(t)$ power spectrum is given by the formula 4.1.1 reported before.

Model matrices are listed here:
A =

	$x1$
$x1$	0.92685

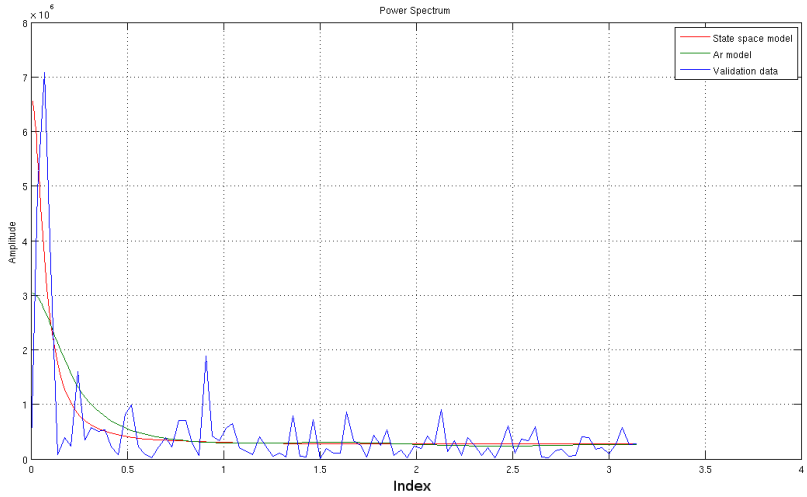

Figure 4.15.: toff spectrum of event based CPU bound task model compared to validation data set one

C =

$$\begin{array}{cc} & x1 \\ y1 & 7136.10 \end{array}$$

K =

$$\begin{array}{cc} & y1 \\ x1 & 3.4146e-5 \end{array}$$

x(0) =

$$\begin{array}{cc} x1 & 0.11832 \end{array}$$

The model is very simple, but, as related figures show, it fits data quite well both in Fourier transform and time domain.

In figure 4.15 is compared power spectrum of $e(t)$ with data validation one.

The next figure 4.16 reports time domain model fitting.

The main problems emerge user inserts big numbers. In the middle of the toff signal, in fact, are presents high peaks, followed by low toff values. High peaks correspond to sleeping time while task is waiting for the next input by keyboard; following low sleeping durations are due to execution preempt while task is attempting to calculate the $n-th$ (big) prime number (in fact low toff values correspond to high ton values in the ton data set). This can be seen as a non stationarity factor which is not well modeled by the state space model. However, the model is able to follow main temporal trend (confirmed by spectrum plot in figure 4.15) and that is what was

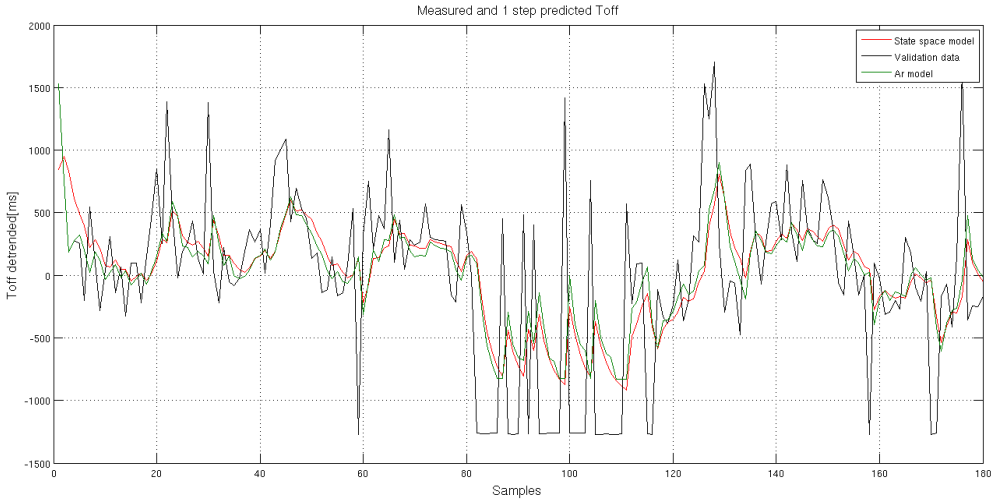

Figure 4.16.: toff produced by one-step predictor model compared to validation data set in time domain

desired.

Finally, residuals autocorrelation is plotted in figure 4.17 where 99% confidence interval is highlighted.

4.1.7. Ton identification for event based tasks

In this section a model for event based task ton is discussed. Some considerations made for toff are still valid: event based tasks are intrinsically I/O bound and, usually, computation is quite light. Task used to represent this situation, waits a number n in input and then calculates $n-th$ prime number by a successive divisions algorithm. Light computation is simulated inserting small numbers (but sometimes, some big ones are inserted, to simulate some heavier requests).

Variations in the computational load had a quite important impact on ton data series and, differently from toff data set, the resulting model is quite complex, but, from both spectral and time analysis, this model has a good behavior. Figure 4.18 shows spectral analysis and figure 4.19 shows temporal analysis.

The model structure, as usual, is:

$$x(t+1) = Ax(t) + Ke(t)$$

$$y(t) = Cx(t) + e(t)$$

where $e(t) = H(z)in1(t)$ is a filtered white noise (a colored noise). $e(t)$ spectrum is given by the formula 4.1.1 reported before. Matrices are reported above:

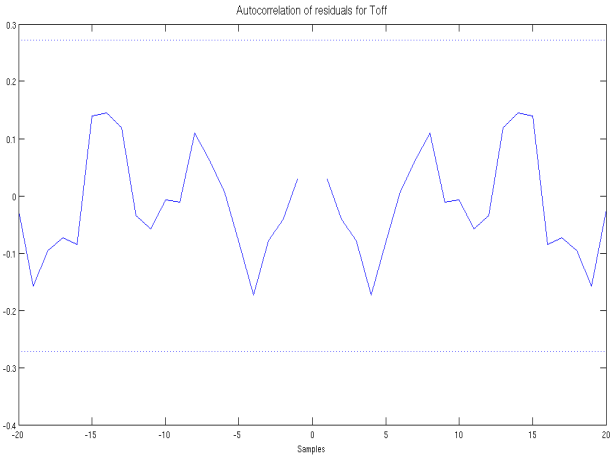

Figure 4.17.: toff residuals autocorrelation, 99% confidence interval

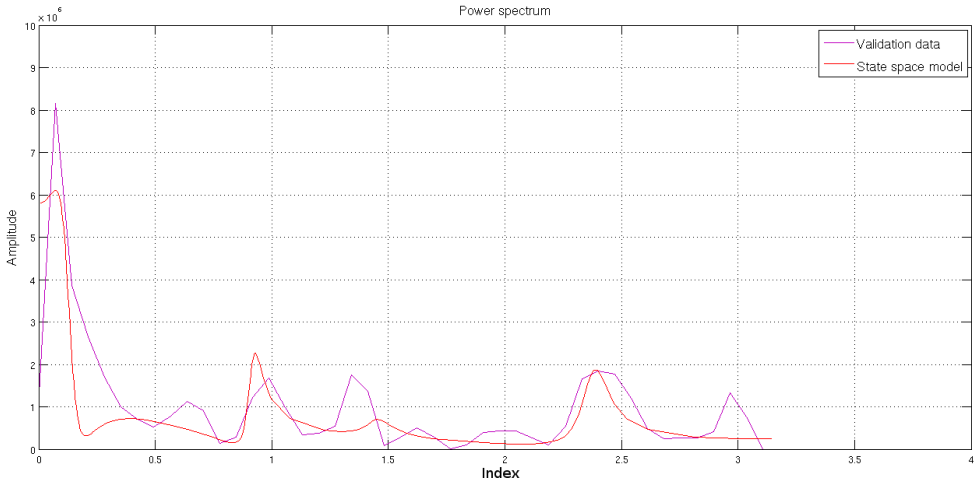

Figure 4.18.: power spectrum of the ton signal produced by the model compared to power spectrum of the validation data set

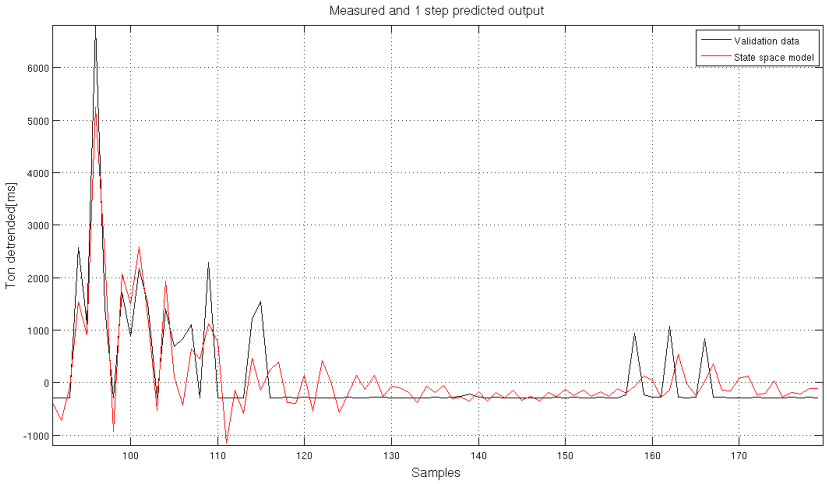

Figure 4.19.: ton produced by the one-step predictor built from the model, compared to the validation data set

A =

	$x1$	$x2$	$x3$	$x4$	$x5$
$x1$	0.87508	−0.22471	0.16716	0.15469	0.18617
$x2$	0.15253	−0.62780	−0.45092	−0.15294	−0.54898
$x3$	0.15048	0.31147	0.20683	0.68917	−0.49596
$x4$	−0.08876	−0.05538	−0.68026	0.59645	0.42169
$x5$	0.14428	0.52737	−0.42561	−0.24206	−0.34669
$x6$	−0.07523	−0.08609	0.07759	0.15190	−0.19648
$x7$	−0.00943	−0.04101	0.05043	0.02092	−0.02180
$x8$	0.07706	−0.00515	0.04119	−0.06441	0.31077
$x9$	0.00760	0.04994	−0.03774	−0.01103	−0.03090
$x10$	0.01093	−0.01302	0.01435	0.00025	0.07245

	$x6$	$x7$	$x8$	$x9$	$x10$
$x1$	-0.12540	-0.04383	0.15408	-0.05833	0.08658
$x2$	0.10382	0.03661	-0.04626	0.08462	0.02647
$x3$	0.22052	-0.00597	-0.00706	0.12733	0.04776
$x4$	-0.02566	-0.00699	-0.00328	-0.04156	-0.03650
$x5$	-0.29599	0.21665	0.36928	0.05127	0.17801
$x6$	-0.84705	0.01507	-0.38664	-0.04751	-0.19033
$x7$	0.14789	0.95801	-0.18225	-0.19783	-0.12335
$x8$	-0.23165	0.32668	0.08755	0.58654	0.21458
$x9$	0.06421	-0.03519	-0.59880	0.11225	0.70646
$x10$	-0.08863	-0.004797	0.12629	-0.68434	0.49459

C =

	$x1$	$x2$	$x3$	$x4$	$x5$
$y1$	-6851.7	-3748.2	2139.2	2337.2	-3227.1

	$x6$	$x7$	$x8$	$x9$	$x10$
$y1$	89.902	822.22	2561.3	1038.8	1776.9

K =

	$y1$
$x1$	$-3.2761e-5$
$x2$	$2.6593e-5$
$x3$	$2.7867e-5$
$x4$	$-7.4396e-6$
$x5$	$-9.2383e-6$
$x6$	$-3.6710e-6$
$x7$	$-5.1017e-6$
$x8$	$2.1865e-5$
$x9$	$-3.3267e-6$
$x10$	$6.7874e-6$

x(0) =

$x1$	-0.22398
$x2$	-0.24837
$x3$	0.14686
$x4$	0.16053
$x5$	-0.04963
$x6$	-0.08630
$x7$	0.07140
$x8$	0.33834
$x9$	0.03765
$x10$	0.13184

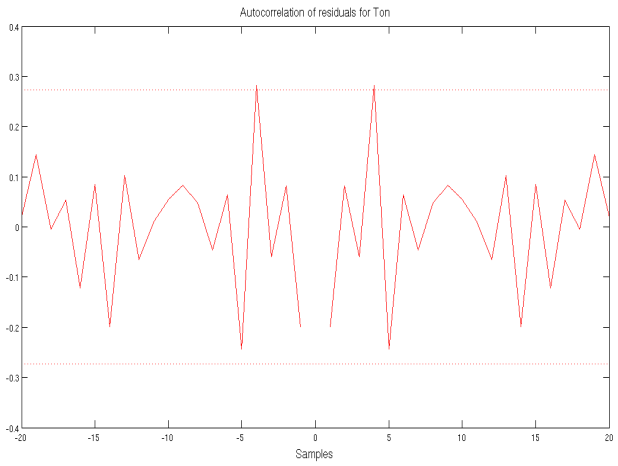

Figure 4.20.: ton residuals autocorrelation; 99% confidence interval

As usual, this model was estimated using Matlab identification toolbox (n4sid algorithm) applied on the data set profiled with trace-cmd. This is a ten-th order model; it is one of the most complex identified, a testimony that, indeed, modeling ton of this type of task is not easy.

Figure 4.20 shows residuals autocorrelation; 99% confidence interval is highlighted.

Residuals can be considered uncorrelated, error signal is white (with a 99% confidence level) and the model can be accepted.

4.2. Cache references identification

In this section, it will be discussed a model identification of cache reference signal. Identification procedure will be performed for all the different task classes exploited. It is important to remark that the models are not intended, as usual, to identify a single entity, but they must adequately represent a large class of tasks. Tasks belonging to the same class have common characteristics, from system behavior point of view, but, in general, they have different structure and implementation at application level. For example, Amarok audio player and Mplayer video player, are very different applications, but, from system point of view, they have similar needs: both should reach their target workload before a certain time deadline, and this is periodically repeated, on the base of audio or video stream played. They should then be modeled by the *same* model (or, at least, by two models having the same structure), even if a single Mplayer execution trace is very different from a single Amarok execution trace. This remark is valid for all the task classes (and subclasses) pointed out in this work. The cache references (and cache miss) models are identified from data sets produced by profiling library (described in section 3.5). It is important to remember that library

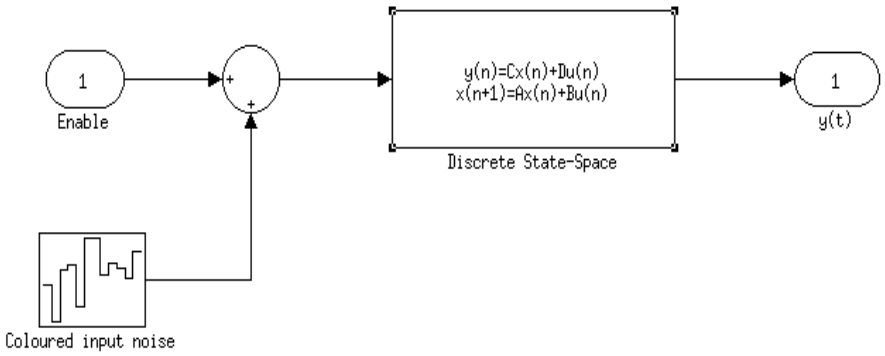

Figure 4.21.: Identification approach for cache reference signal

samples at 1000 Hz, so Nyquist frequency, for all cache reference (and cache miss) data sets is 500 Hz (except for those data sets where it's otherwise specified).

4.2.1. Cache references identification for periodic tasks

Mplayer is a quite good candidate to represent periodic tasks; it stresses system from both CPU and memory point of view. In this section, identification of a cache reference model will be discussed. To identify a model, an auxiliary signal has been built, the enable signal. The enable signal can assume only two values: it is one when task is in execution, zero otherwise.

The enable signal has been built from a threshold analysis over CPU clock cycles count signal; when signal is greater then the threshold, enable signal is one, otherwise is zero.

However, data are affected by noise, so the identification approach has to take account of this. A suitable cache reference model structure is represented in the block schema of figure 4.21.

Figure 4.22 shows the frequency response of the model, compared to the validation data one.

Both the state space model and the arx [8] model seem to have a good behavior; it has been chosen the arx model because it is simpler and it has a slightly better behavior both in frequency and time domain.

Figure 4.23 shows a time fitting detail, related to the arx model data set.

The model identified is an arx(4,3); its structure is simple:

$$A(q)y(t) = B(q)u(t) + e(t)$$

$$A(q) = 1 + 0.0355q^{-1} + 0.001662q^{-2} + 0.05611q^{-3} - 0.7448q^{-4}$$

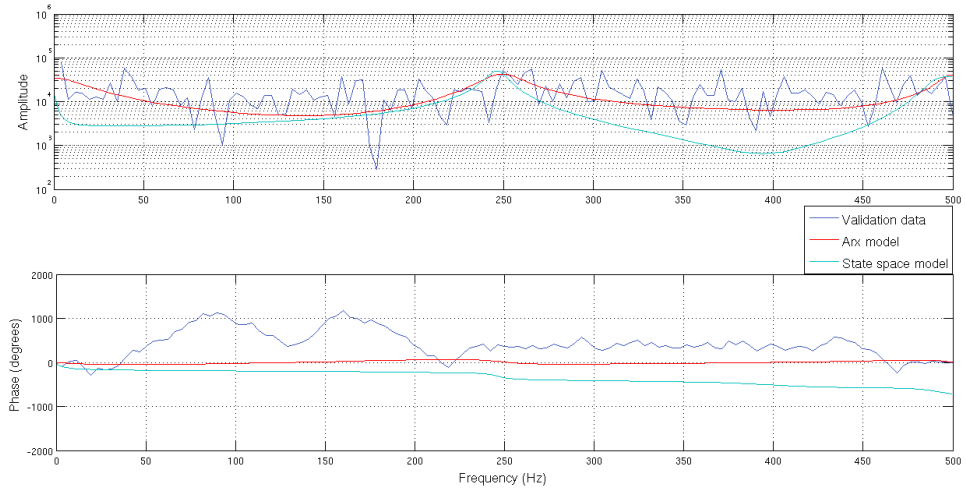

Figure 4.22.: Frequency response of validation data and identified models

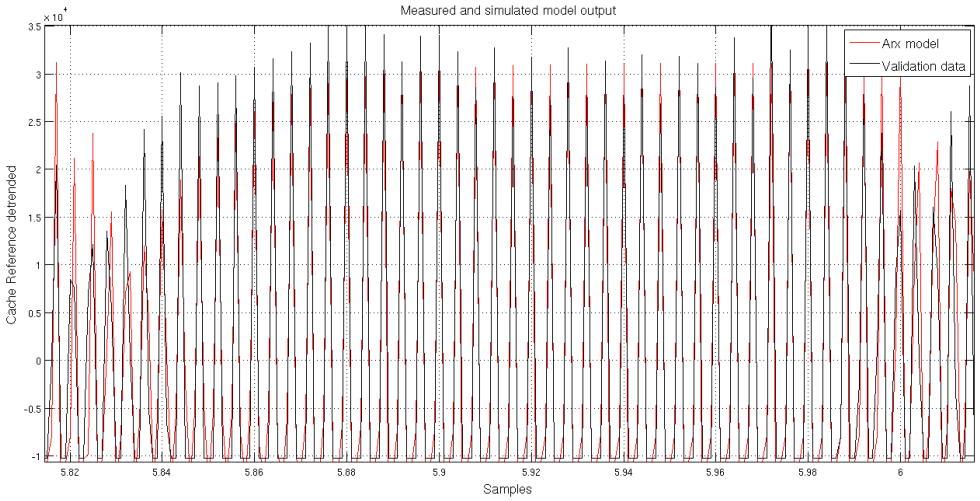

Figure 4.23.: Arx model time fitting detail

$$B(q) = 9974 + 230.6q^{-1} - 587.1q^{-2} + 2374q^{-3}$$

It was estimated using Matlab identification toolbox (arx algorithm).

4.2.2. Cache references identification for batch CPU bound tasks

In this section, a model identification for cache references signal, related to CPU batch tasks is performed. Before to proceed, a little remark is needed: Even if particular batch memory bound tasks have been implemented (in C and Python) and profiled, cache subsystem identification (cache references and cache miss signals) has been performed only on CPU bound tasks. This because Python language has a quite complex memory subsystem management, so, a CPU bound task is enough to see how a batch task stresses both CPU and memory subsystem. Moreover, we have a quite direct comparison on how two similar applications, implemented with different languages, stress the same resources (in profiling section 3.7.2, it's remarked that Python tasks and C tasks are characterized by very different cache miss and cache references signals).

CPU bound task implemented in c

To identify the model laying under cache references signal of a CPU bound batch task implemented in C, time series approach has been used; original signal, sampled at 1000 Hz, is too noisy and identifying a model over that data set results very difficult. So, to overcome this problem, original signal has been 'resampled', cumulating samples a group of ten. Original data set is divided in groups of ten samples and data of the group are summed, creating one sample that represents cumulated value of the group of ten samples. New signal results sampled at 100 Hz (so its Nyquist frequency is 50 Hz). This approach is useful to spot slow dynamics, eventually hidden by high frequency noise. On the other hand, the model identified has to be exploited to realize a multi core load manager, and load manager migration frequency should be, at least, ten times lower on core scheduling frequency. A reasonable value for on core scheduling frequency could be 1000 Hz, so, a suitable migration frequency could be 100 Hz and data sampled at 100 Hz are still a suitable data set.

The identification approach is the classical one adopted for time series and represented by block schema in figure 4.3.

Figure 4.24 shows the estimated input noise power spectrum frequency fitting.

Cache references signal has still a quite noisy behavior, but, time fitting, reported in figure 4.25, is acceptable, because identified model is able to catch main temporal trends spotted by the validation data set.

Identification procedure has led to a third order state space model; it has the following structure:

$$x(t + Ts) = Ax(t) + Ke(t)$$
$$y(t) = Cx(t) + e(t)$$

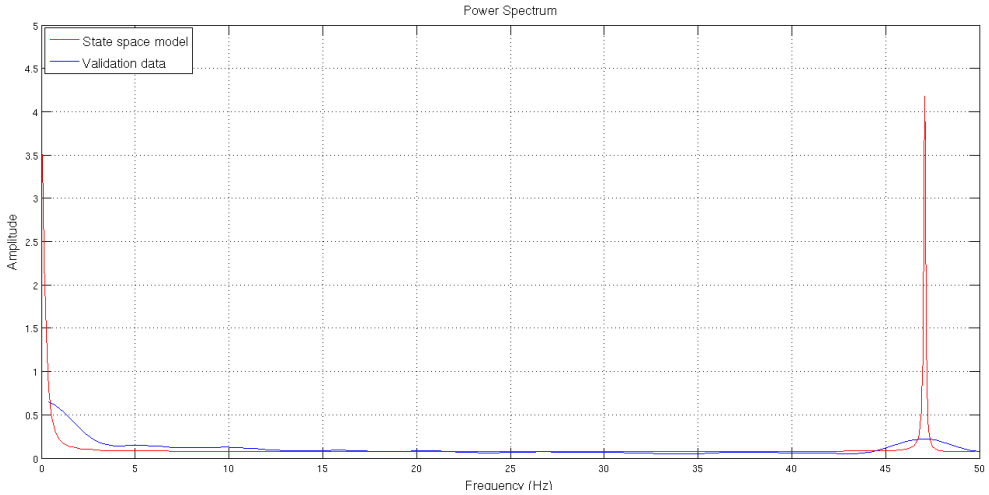

Figure 4.24.: State space model: noise power spectrum frequency fitting

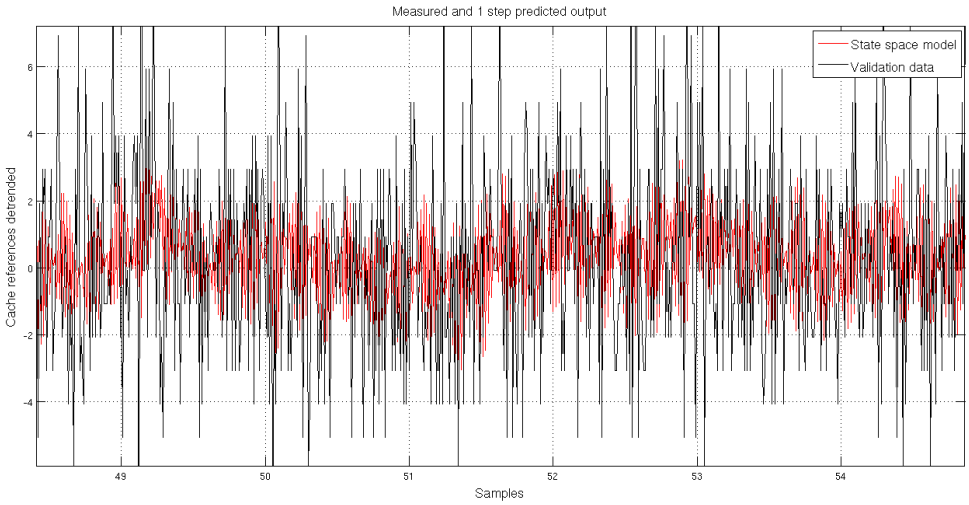

Figure 4.25.: State space model: time fitting

In our case $Ts = 0.01s$; let $x(t)$ be the state vector of the system, $e(t)$ a colored noise in input and $y(t)$ the model output. Matrices are

A =

	$x1$	$x2$	$x3$
$x1$	0.96105	−0.23924	−0.00703
$x2$	−0.21629	−0.95672	0.19741
$x3$	−0.02533	−0.16660	−0.97555

C =

	$x1$	$x2$	$x3$
$y1$	245.11	20.538	175.81

K =

	$y1$
$x1$	0.00027493
$x2$	$-2.6445e-05$
$x3$	−0.00024509

x(0) =

$x1$	0.016702
$x2$	−0.002312
$x3$	−0.001361

This model was estimated using Matlab identification toolbox (n4sid algorithm).

CPU bound tasks implemented in python

Unfortunately, cache references signal of python CPU bound task has a profile that can hardly be identified with linear methods: in figure 4.26 is reported global cache references signal plot.

How can be guessed, (and better shown in detail of figure 4.27) cache references signal is not stationary, so it has not been possible to obtain good identification results over this data set; an useful future work could be the project of a different identification approach to obtain a suitable model for this signal.

4.2.3. Cache references identification for 'batch' I/O bound tasks

The clarification made earlier about this class are still valid; batch concept is a little bit relaxed to consider that group of applications which behaves like ls. To identify model of cache references signal related to this task class, time series approach has given better results than approaches involving exogenous signals. Model structure is represented in the block schema in figure 4.3, where the only model input is noise.

Original cache reference signal, has a large amount of samples and a quite 'noisy' behavior; trying to identify a model from it has given poor results. To overcome this

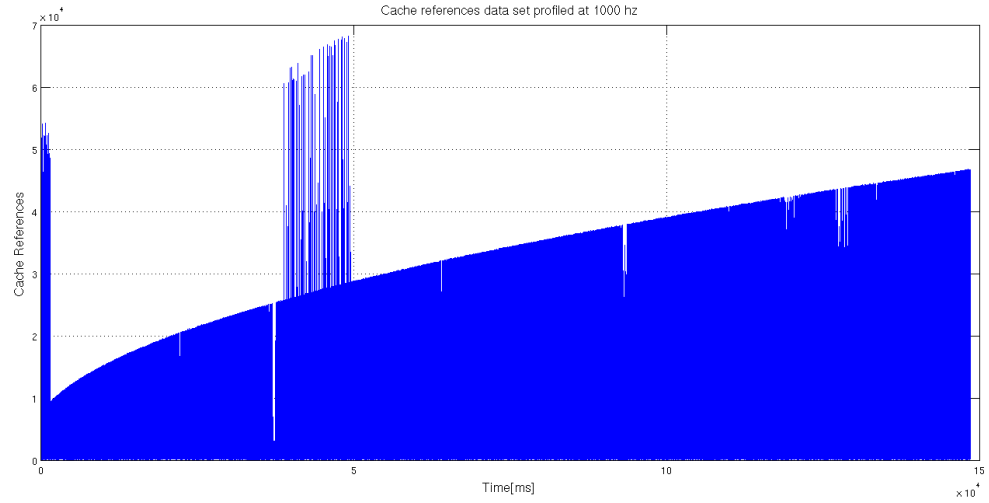

Figure 4.26.: Cache references plot of a CPU bound batch task implemented in Python

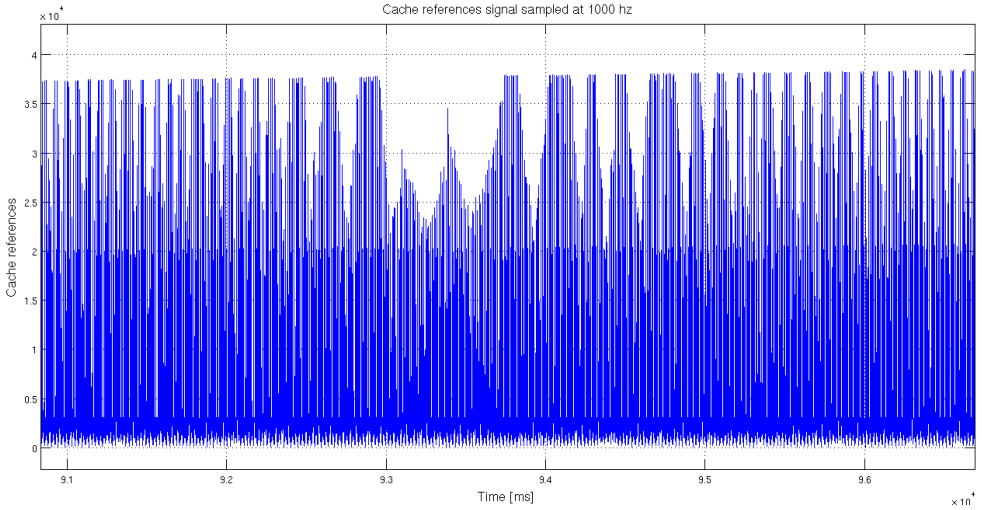

Figure 4.27.: Cache references detail plot of a CPU bound batch task implemented in Python

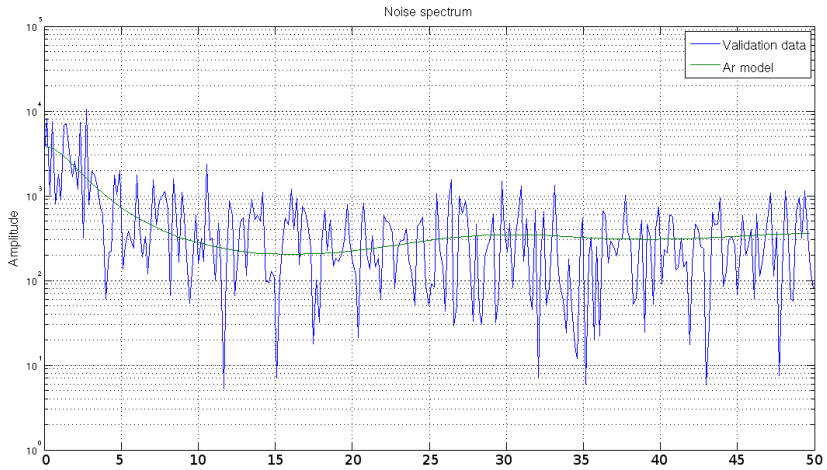

Figure 4.28.: Ar model noise power spectrum frequency fitting

problem, sample was aggregated (summed) a group of then; in this way, from 10 original samples, only one sample (which is the cumulative value) remains. This is licit, and already discussed in section 4.2.2.

Figure 4.28 shows the validation data set's power spectrum and the model's noise power spectrum comparison. Since cumulative data realized correspond to a 100 Hz sampling, new Nyquist frequency is 50 Hz. Ar model, even if it is a simple model, is able to interpolate the data frequency content in an acceptable way.

To provide a further test in support of the goodness of this modeling approach, figure 4.29 reports time fitting of identified ar model over the validation data set extracted from ls cache references profile. Figure 4.30 shows time fitting of the same ar model over an other (written ad-hoc) application's cache references data set.

The model is able to catch main signal trend in both cases, even if the two time series are quite different. This little example should give an idea about which is the purpose of the models built in this work: Our intent is not to to model every single kind of implementable application, it would be too difficult and, probably, useless; we would like to build models which can catch task 'high level' behavior, in other words, these models should grasp the structural properties of a certain class of task, in such a way that a lot of different tasks, belonging to the same class, can be represented by the same model, trying to limit model complexity.

To come back to the model, it has the classic ar structure:

$$A(q)y(t) = e(t)$$

$$A(q) = 1 - 0.1379q^{-}1 - 0.2215q^{-}2 - 0.2089q^{-}3 - 0.1253q^{-}4$$

It was estimated using Matlab identification toolbox (arx algorithm).

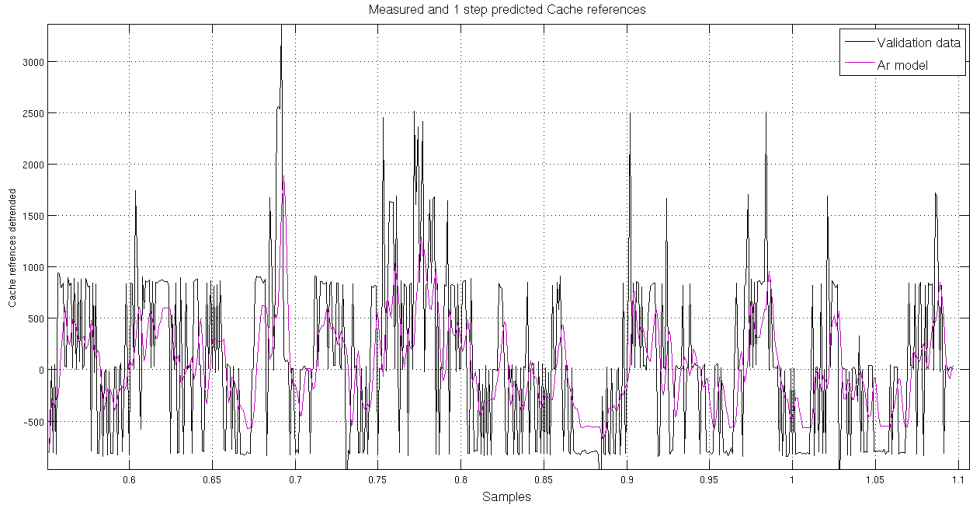

Figure 4.29.: Ar model time fitting over ls data set

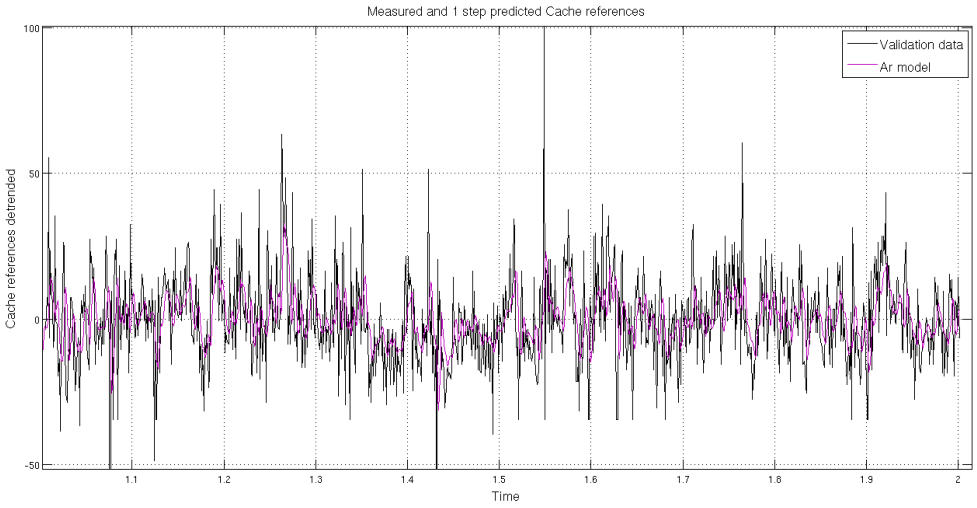

Figure 4.30.: Ar model time fitting over an other data set

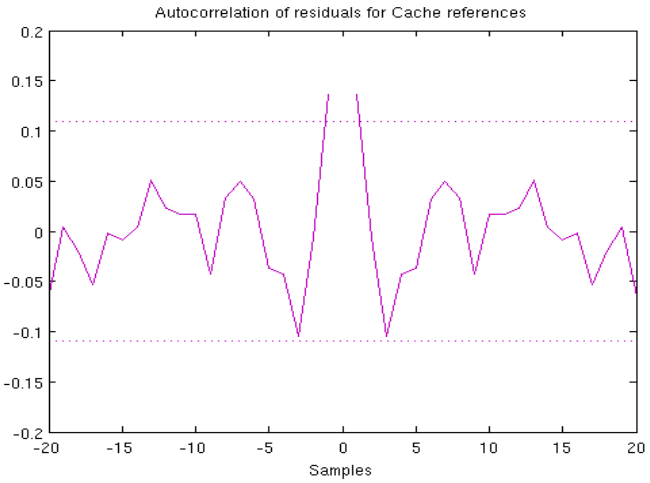

Figure 4.31.: Cache references residuals autocorrelation, 99% confidence interval

To conclude this section, figure 4.31 shows output residuals autocorrelation, with the usual 99% confidence interval highlighted.

4.2.4. Cache references identification for priority tasks

In this section, the identification procedure is carried out over nano execution trace profile. nano has been chosen as representative of priority based tasks, because it is a text editor simple but functional and, working on terminal, it is easy to profile it completely excluding graphic server (X or similaria) interference. Task models are made to implement a load balancing controller in a multicore control-based scheduler and to tune a suitable migration policy. Load balancer module should migrate tasks between different cores to guarantee an homogeneous (homogeneous in term of CPU needing, cache utilization and responsiveness needed) workload for each core, in other words, mix of periodic, batch, priority and event based tasks present on core should be homogeneous from performances point of view. To ensure that, load balancer needs to know how much and how quickly each task needs for CPU and cache, to decide what to migrate and where. This control loop should work slower than the control loop that, on every independent core, sets CPU bursts for tasks present, so, if calculus of task bursts on an independent core works, hypothetically, at 1000 Hz (realistic), load balancing control can work at 100 Hz: load balancing control should see single cores always a regime from CPU quantum assignation point of view, to avoid dangerous interference that could have bad impacts on overall system performances. For this reason, and to obtained models suitable for a wider range of real applications, original data set has been reorganized, grouping samples by ten end creating cumulated samples: each ten samples are summed forming one sample which is cumulated value of the group of ten.

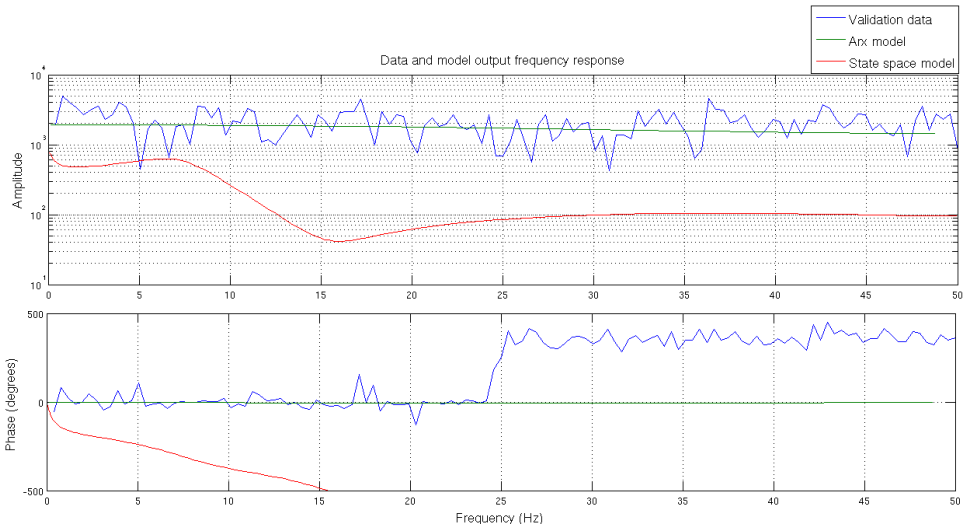

Figure 4.32.: Arx model frequency response compared to validation data one

In this way, a data set which was originally sampled at 1000 Hz, after data reorganization, results sampled at 100 Hz (and corresponding Nyquist frequency is 50 Hz). Moreover, cache references signal, often has a 'noisy' behavior; reducing sampling frequency helps pointing out slow dynamics of the signal (useful from the perspective of a load balancing model-driven), which could be hidden by fast 'noisy' dynamics.

Priority based tasks are interactive, so it makes sense trying to identify a cache references model building enable signal. This signal, as explained before, is one when CPU cycles count signal is higher than a certain threshold, zero otherwise.

The identification approach has followed the schema reported in figure 4.21 with two input signals (a colored noise and enable signal) and one output (simulated cache references signal).

The model resulting from identification procedure, is an arx model, which has proven to have good interpolation properties, both in time and frequency domains.

Figure 4.32 reports the model frequency response, compared to the data one. Arx model has a frequency response quite similar to the one of the validation data set (at least in a certain frequency range), an acceptable fitting in time domain is expected to and, in fact, time fitting of figure 4.33 is good.

This model is an arx(4,3); its structure is:

$$A(q)y(t) = B(q)u(t) + e(t)$$

where $u(t)$ and $e(t)$ are input signals ($u(t)$ is the enable signal and $e(t)$ is the input colored noise) and $y(t)$ is model output signal.

$$A(q) = 1 - 0.1605q^{-1} + 0.01819q^{-2} - 0.001504q^{-3} + 0.0003832q^{-4}$$

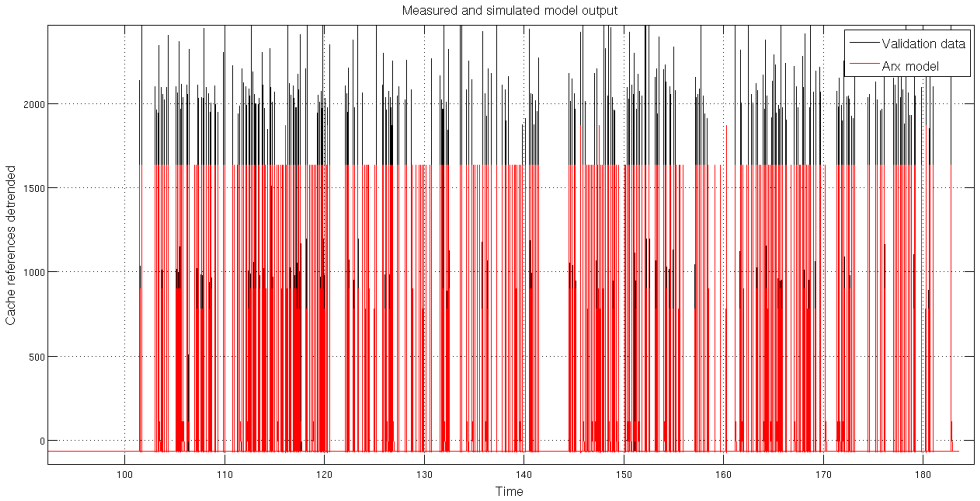

Figure 4.33.: Cache references arx model time fitting

$$B(q) = 1695 - 36.21q^{-}1 - 15.72q^{-2} + 2.98q^{-3}$$

The model was estimated using Matlab identification toolbox (arx algorithm).

4.2.5. Cache references identification for event based tasks

To represent event based class, which is mainly characterized by light computational load, high user interactivity and responsiveness needing, an interactive task has been used: details are shown in section 3.7.4. It waits on a user terminal for a number n which has to be inserted by user. After that a number has been inserted, it spawns a new thread which calculates the $n - th$ prime number by a successive divisions algorithm. The modeling approach follows the schema of figure 4.21, exploiting an enable signal in input. Enable signal construction has been already discussed in previous sections and is no longer reported to improve readability.

Figure 4.34 shows frequency response of the two identified models (a state space model and an arx model). The original data set has been sampled at 1000 Hz, so frequency axis goes from 0 to 500 Hz (Nyquist frequency).

Figure 4.35 shows arx model time fit.

Since both models have a frequency response quite different from the one of the data set, it was chosen the arx model because it is simpler and it has a better time fit.

The model identified is an arx(4,3), and it has the structure:

$$A(q)y(t) = B(q)u(t) + e(t)$$

$$A(q) = 1 - 0.006074q^{-1} - 0.06423q^{-2} - 0.03039q^{-3} + 0.004727q^{-4}$$

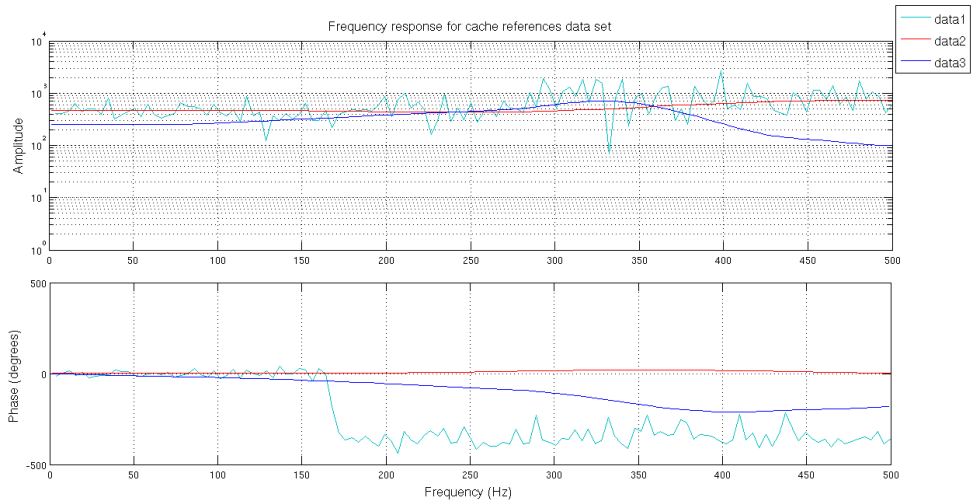

Figure 4.34.: Arx and state space models frequency response compared to the validation data set one.

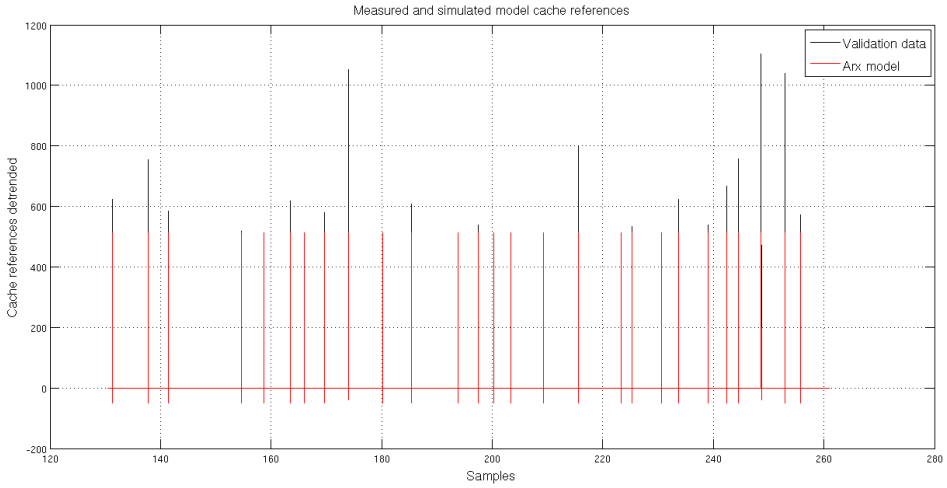

Figure 4.35.: Arx model time fitting over the validation data set

$$B(q) = 512.7 - 104q^{-1} + 56.93q^{-2} - 48.13q^{-3}$$

It was identified using Matlab identification toolbox (arx algorithm).

4.3. Cache miss identification

In this section is discussed a cache miss model identification related to the classes of tasks already exposed. The identification approaches followed are mainly the two already spotted:

1. Approach exploiting an enable signal, resulting in a state space model or in an arx model, both having enable signal and a colored noise as input signals.
2. Approach reorganizing profile data set (creating cumulative samples). Reorganized data set can be treated as a time series (in the case that enable signal is not useful) or as an input/output model if enable signal is useful, and then proceeding to identify a model and a noise spectrum.

4.3.1. Cache miss identification for periodic tasks

Now cache miss signal related to Mplayer data set is analyzed. Identification approach exploiting enable signal, as represented in figure 4.21 has given good results, both in frequency and time domains; arx model has proven to have good fitting properties together with a simple structure. The identified model is an arx(4,3) process.

Figure 4.36 shows frequency response of the identified model, compared to the validation dataset one;

Estimated noise power spectrum has an acceptable interpolation of the data set power spectrum, reported in figure 4.37.

In time domain, the identified model has a good fitting, catching main trend of cache miss validation data set; figure 4.38 shows time domain plot.

After that model properties have been described both in time and frequency domains, here it is reported model structure and coefficients values: it has a classical arx structure:

$$A(q)y(t) = B(q)u(t) + e(t)$$

where $y(t)$ is output signal, $u(t)$ is input signal (in our case enable signal) and $e(t)$ is a colored noise.

Polynomials:

$$A(q) = 1 + 0.04013q^{-1} + 0.05161q^{-2} + 0.09379q^{-3} - 0.5797q^{-4}$$

$$B(q) = 5003 - 913.3q^{-1} - 517q^{-2} + 340.2q^{-3}$$

The model, as usual, was estimated using Matlab identification toolbox (arx algorithm).

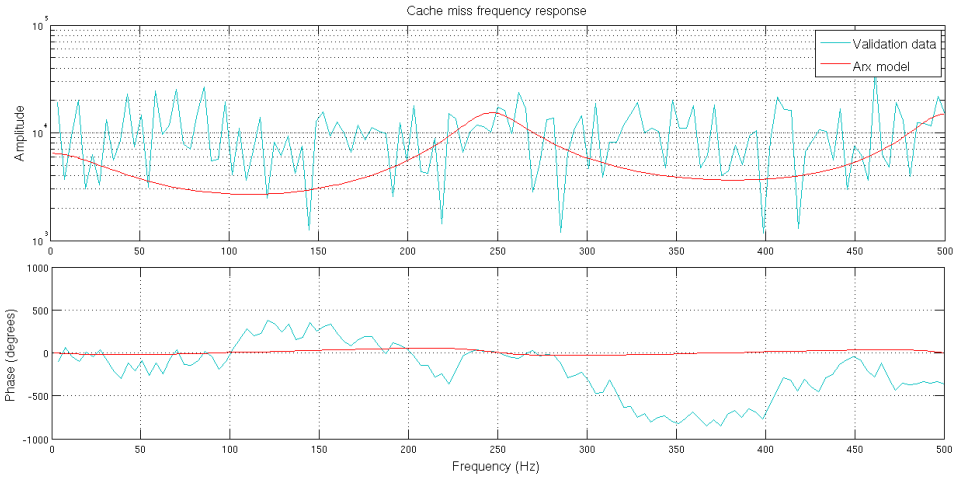

Figure 4.36.: Frequency response of Arx model

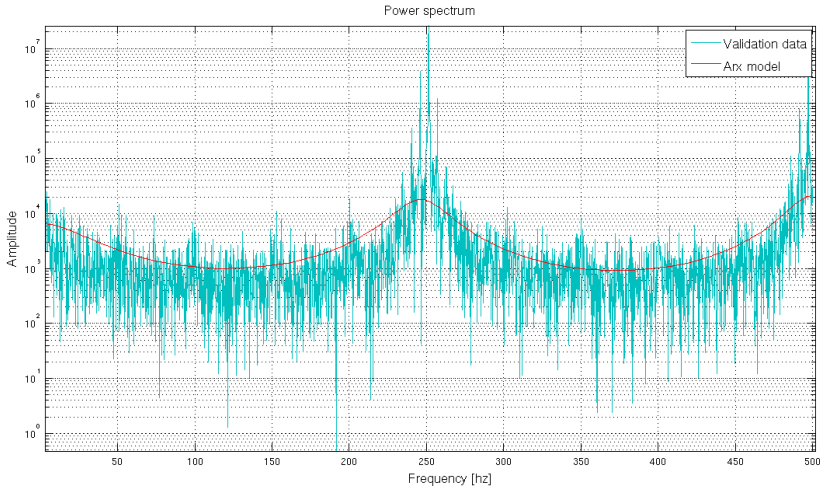

Figure 4.37.: Arx model noise power spectrum and data set power spectrum

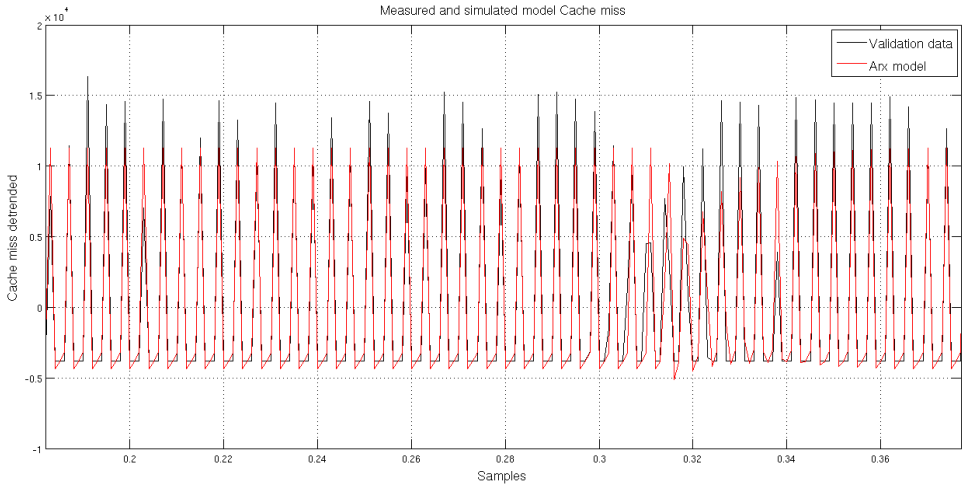

Figure 4.38.: Arx model data set time fitting

4.3.2. Cache miss identification for batch tasks

Program written in c to simulate a CPU bound batch task stresses a lot CPU subsystem, but, how it's reported in figure 4.39, it is affected by a very few cache miss (we have about 350.000 samples and less than 10 cache miss). This because computation is performed on a very small set of data, which probably can be totally stored in local core cache (probably cache miss recorded are due to other system's tasks interference). This kind of execution trace, so, is not significant to build a model. Even using enable signal does not help, because it represents execution times established by scheduler, but when task is in execution, it rarely is affected by cache miss, so frequency contribution of enable signal is not useful for model identification purpose. To build a cache miss model, related to batch tasks, Python tasks can provide more significant results.

Cache miss identification for Python tasks

To identify a cache miss model related to this task class, building enable signal is not helpful. Enable signal is one when task is in execution, so time intervals in which enable is high correspond to task execution times assigned by scheduler, but they do not have correlation with the possibility that a cache miss happens or not (the only correlation is given by the fact that cache miss are always zero when task is not executed, and that's not enough). So, to have at least one hope to identify the model laying under data, cache miss signal has been reorganized, building cumulated samples. Original signal was sampled at 1000 Hz, so, new cumulated samples signal result sampled at 100 Hz (its Nyquist frequency is 50 Hz).

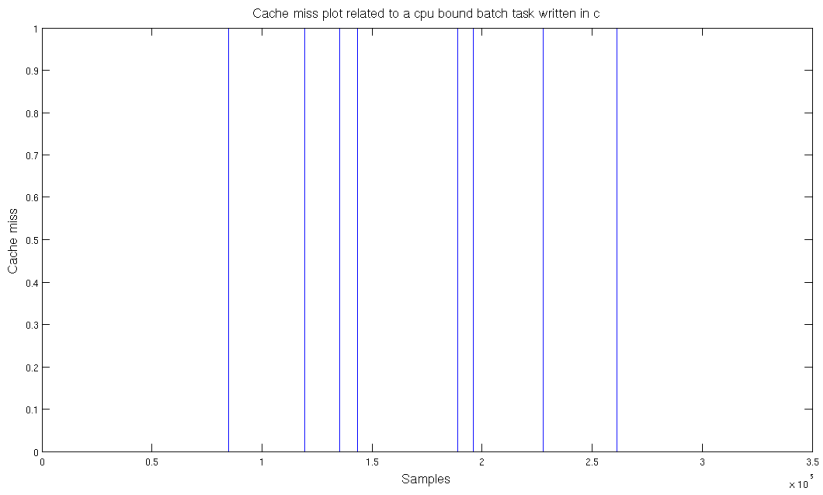

Figure 4.39.: Cache miss data set related to CPU bound task written in C

Python language manages objects through dictionaries, so it has a quite complex memory system. This is reflected by spectrum of cache miss signal, reported in figure 4.40.

Data spectrum is almost flat from 0 to Nyquist frequency; it is quite similar to a white noise. This dramatically limits our expectations on time domain fitting; in fact, the identified ar model, is able to catch mean value of the data validation set, but the remaining dynamics are modeled by the noise; figure 4.41 shows the model time fitting.

The identified model is simple: it has a classic ar(4) structure:

$$A(q)y(t) = e(t)$$

where $e(t)$ is input colored noise and $y(t)$ is the only model output; $A(q)$ is the polynomial

$$A(q) = 1 - 0.06061q^{-1} - 0.03198q^{-2} - 0.02875q^{-3} - 0.08048q^{-4}$$

This model was estimated using Matlab identification toolbox (arx algorithm). Figure 4.42 shows model output residuals autocorrelation, with 99% confidence interval level highlighted;

Residuals can be considered white.

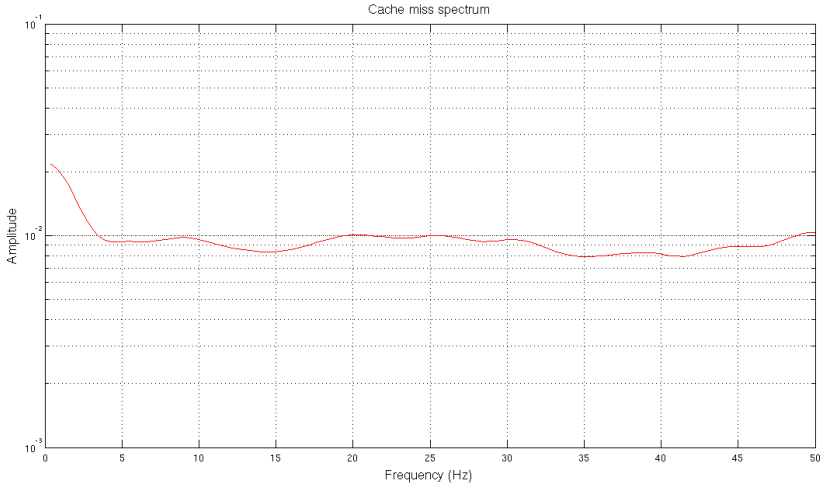

Figure 4.40.: Cache miss data spectrum of a CPU bound task written in Python

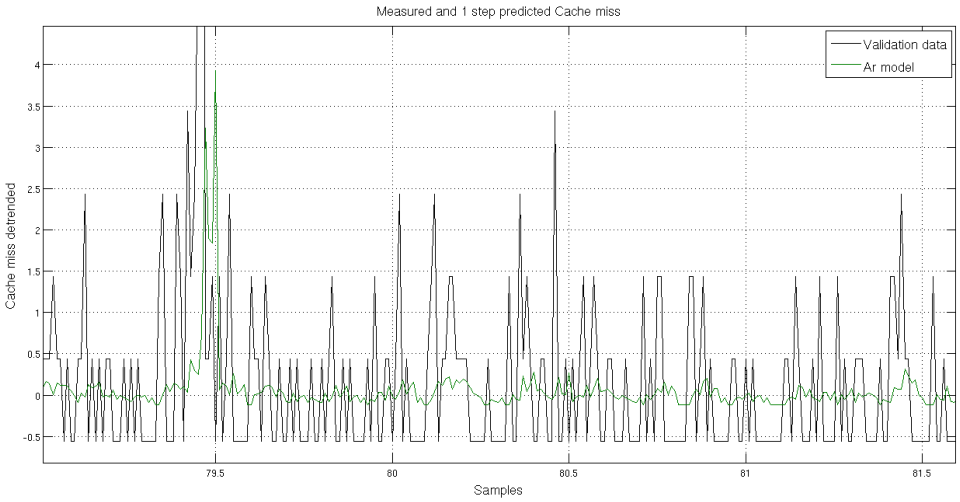

Figure 4.41.: Cache miss model time fit of a CPU bound task written in Python

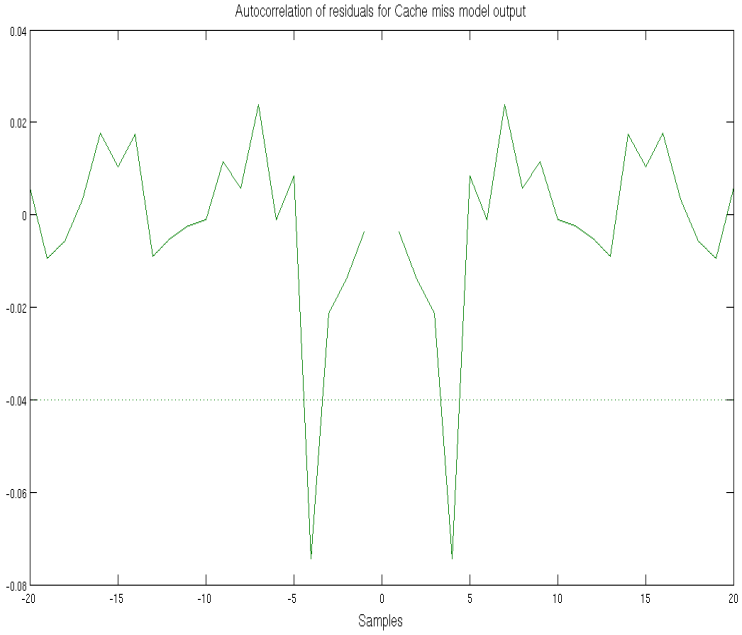

Figure 4.42.: Cache miss ar model output residuals autocorrelation plotted with 99% confidence level interval

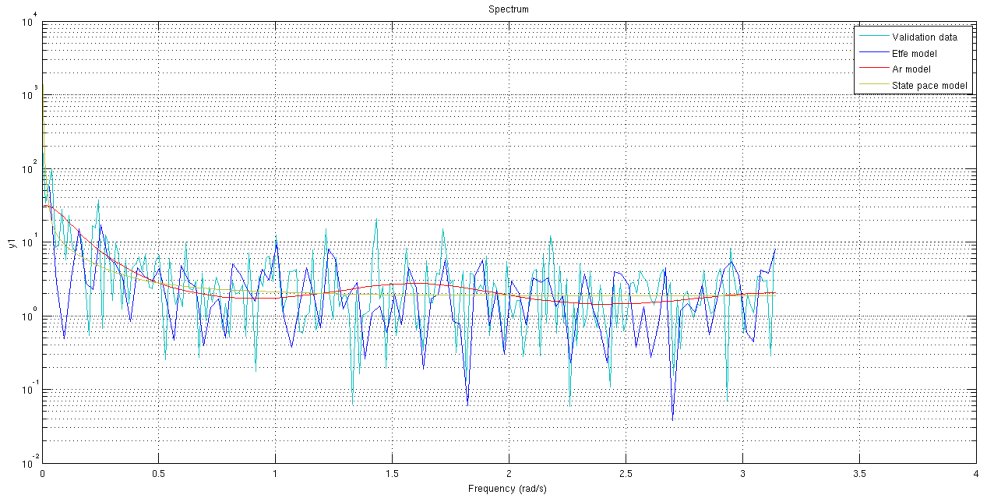

Figure 4.43.: Arx model estimated noise power spectrum compared to the validation data power spectrum

4.3.3. Cache miss identification for 'batch' I/O bound tasks

In this section, identification procedure is carried out to obtain a model for cache miss signal related to ls execution trace. As happened for cache references analysis, the identification process did not give good results on cache miss data set 'as it is' so, data have been reorganized, creating cumulative samples grouped by ten. This operation is equivalent on reducing of an order of magnitude original sampling rate (which was 1000 Hz), so, resulting sample rate is 100 Hz and Nyquist frequency is 50 Hz. The identification procedure on this data set has given acceptable results using the time series approach and considering noise as the only input signal of the model.

Figure 4.43 shows estimated noise power spectrum compared to the data set power spectrum.

Ar model is simple and gives anyway a good interpolation.

The model structure is the one of classical ar processes (represented by block schema in figure 4.3) let $e(t)$ be the colored noise in input and $y(t)$ the model output; the model structure is

$$A(q)y(t) = e(t)$$

$$A(q) = 1 - 0.2724q^{-1} - 0.1252q^{-2} - 0.1358q^{-3} - 0.1902q^{-4}$$

This model is ar(4), (quite simple) and it was estimated using Matlab toolbox (arx algorithm).

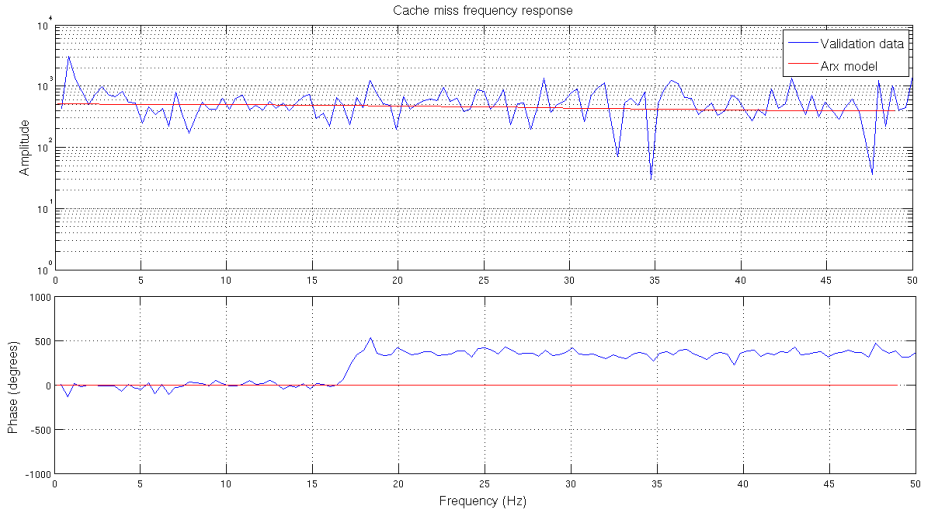

Figure 4.44.: Arx model frequency response compared to validation data one

4.3.4. Cache miss identification for priority tasks

Now, cache miss identification procedure related to nano data set is discussed. nano is an interactive application, so, the approach used to identify cache miss model, exploits the enable signal, built as usual, making a threshold analysis over CPU clock cycles count signal: if clock count ¿= threshold, then enable = 1, else, enable = 0.

Since data profiled at 1000 Hz presented an excessive noisy behavior, identification has been performed on 'resampled' data, summing cache miss values by group of ten samples. Cumulative cache miss signal is so sampled at 100 Hz.

The model structure is the one presented in figure 4.21, in this particular case, data sets are all sampled at 100 Hz. The model frequency response, reported in figure 4.44 and the model time fitting, (a detail is shown in figure 4.45), are both acceptable, confirming the validity of the approach.

This model is an arx(4,3), identified by Matlab identification toolbox (arx algorithm) and it has the classic arx structure:

$$A(q)y(t) = B(q)u(t) + e(t)$$

let $e(t)$ be the colored input noise and $u(t)$ the exogenous enable signal, $y(t)$ is the model output.

$$A(q) = 1 - 0.1043q^{-1} + 0.009027q^{-2} - 0.0006997q^{-3} + 0.0001723q^{-4}$$

$$B(q) = 450.7 + 11.2q^{-1} - 3.513q^{-2} + 0.3579q^{-3}$$

The model residuals autocorrelation is reported in figure 4.46, where interval re-

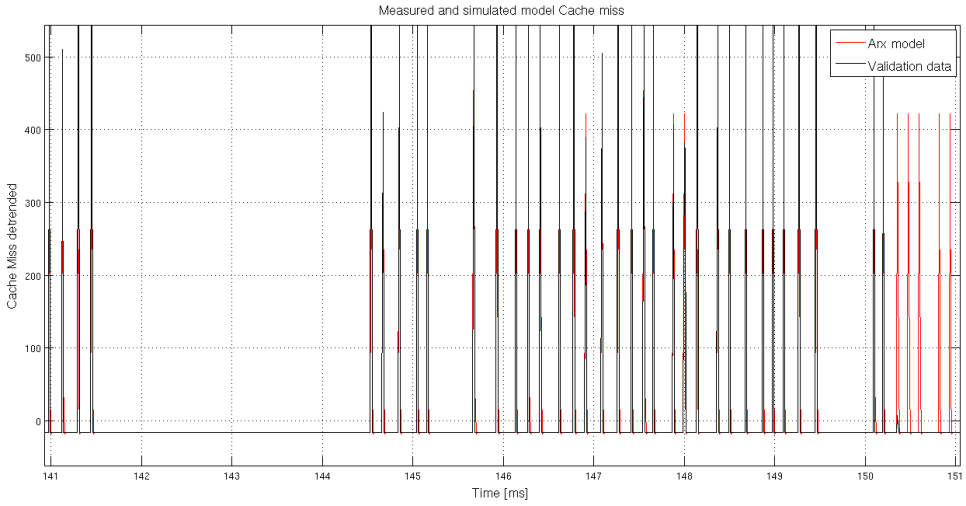

Figure 4.45.: Cache miss arx model time fitting

marked is 99% confidence level interval. Residuals can be considered uncorrelated.

4.3.5. Cache miss identification for event based tasks

In this section is identified a cache miss model for event based tasks class. Event based tasks, usually, have a quite high degree of 'interactiveness' (quotes because with interactiveness is intended not only interactivity with user, but also with system or even with other systems). Since identification results on the original data set were not good enough, the data set has been reorganized creating cumulated samples. Then enable signal has been calculated and identification has been performed following the schema of figure 4.21, with all signal sampled at 100 Hz, so, in spectrum and frequency response plots, values go from 0 to 50 Hz (Nyquist frequency).

The identified model is an arx(4,3).

Figure 4.47 reports the model frequency response compared to the data set one;

The model frequency response is quite similar to the validation data set one (at least in a certain range of frequencies); also estimated noise spectrum (reported in figure 4.48) is similar to the one related to the validation data set.

This model has the classic arx structure:

$$A(q)y(t) = B(q)u(t) + e(t)$$

where $u(t)$ is the enable signal, $e(t)$ is a colored noise and $y(t)$ is the model output. Polynomials $A(q)$ and $B(q)$ are:

$$A(q) = 1 - 0.3013q^{-1} + 0.02627q^{-2} - 0.06741q^{-3} + 0.04804q^{-4}$$

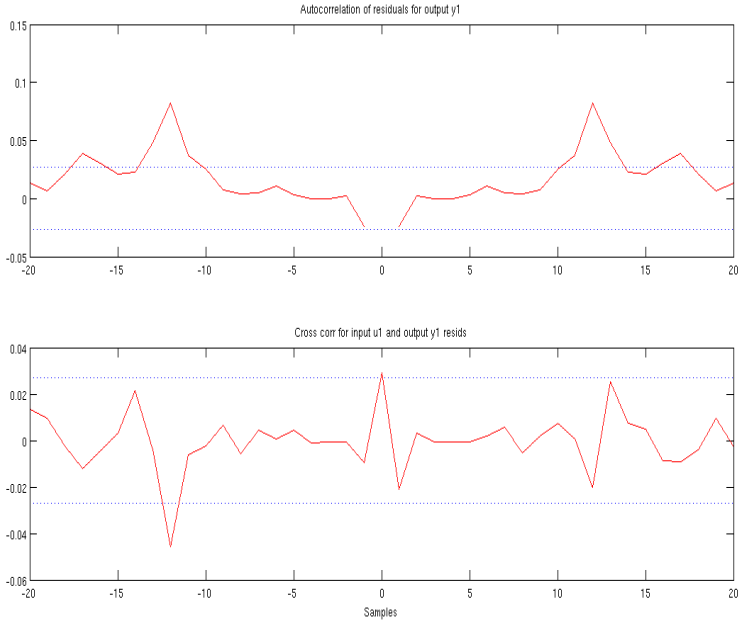

Figure 4.46.: Cache miss arx model residuals plotted with 99% confidence level interval

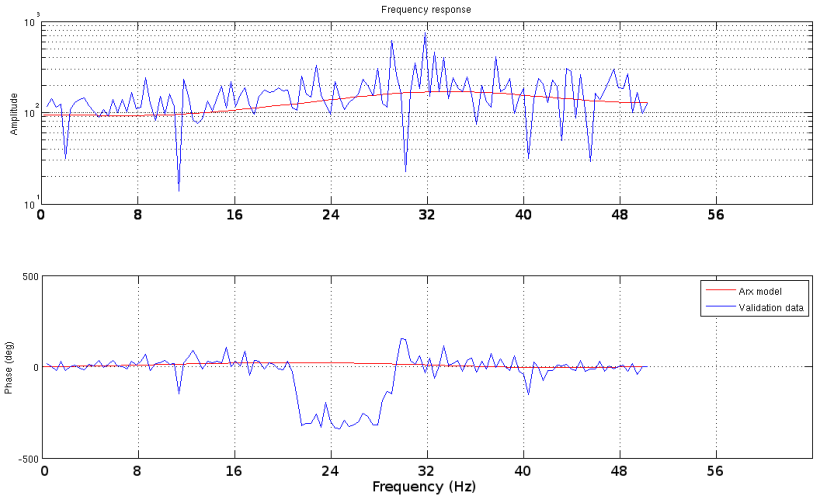

Figure 4.47.: Arx model frequency response compared to the validation data set one

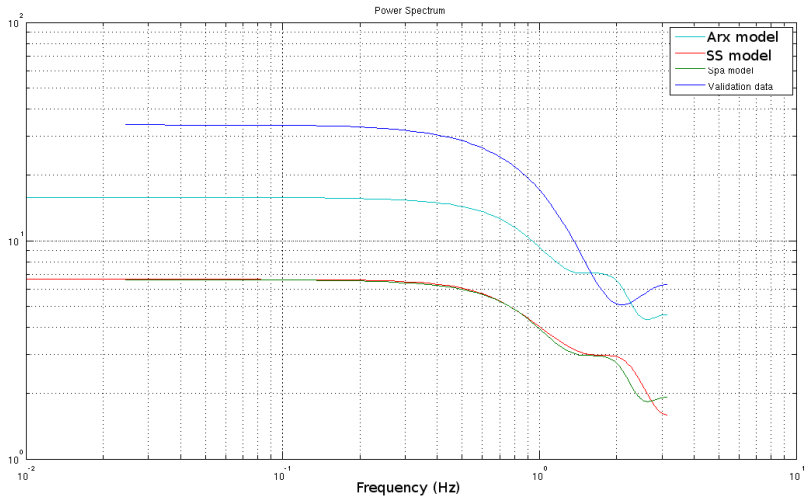

Figure 4.48.: Arx model noise spectrum compared to the validation data set one

$$B(q) = 126.3 - 70.88q^{-1} + 0.1422q^{-2} + 10.83q^{-3}$$

The model was identified using Matlab identification toolbox (arx algorithm).

Part III.

Simulator

5. Development of a Multicore Scheduler Simulator

5.1. Interconnection between task profiling and scheduler simulator

In a multicore context, as already noticed, some aspects of the system extend beyond the model used for the single core simulator, exposed in section 2.1. In the new context, in fact, tasks can not be modeled as a simple delay anymore, Thus, a new and more complicated model is needed.

To start specifying the new model requirements, a first fact to notice is the necessity to catch task behavior aspects that specifically impact a multicore system, like the parallelism degree, the induced CPU stress level, and the memory utilisation (accounting for inter-core cache effects). Apparently, given the need for fast simulation that is ubiquitous whenever the model has to be used for control synthesis and verification, the route to follow cannot be that of an instruction set simulator (hereinafter iss for short). Incidentally, the choice just expressed is backed up by the further need of parametrizing the model easily, and in such a way to represent not a specific application or task, but some class of task behavior that is relevant from the control standpoint at the system level. From both viewpoints the iss approach is too heavy, and works at too low a level [13, 22]. It has to be made clear that this is in no sense a criticism to the iss framework, rather just noticing that it is not the preferable one for this particular work: iss tools in fact give very detailed and precious information about code execution, which the models dealt with in this thesis cannot provide, but it does not yield a system point of view simulation in a reasonable time.

As such, the first point to discuss is how to devise an alternative to iss for the addressed problems. Given the system-theoretic approach followed throughout, a quite natural way to go is that of resorting to the so called "filter approach". In a nutshell, and neglecting a number of mathematical details to quickly introduce the matter, the idea is that whenever a signal of interest, in the form of a time series, comes from a stochastic process, it is possible to replicate said signal by filtering a white noise through a dynamic block, the parameters of which come to describe the behavior of the system that produces the signal. If that system traverses some different operating modes, the hope is that a few parameters of the mentioned block, suitably changed over time according to those models, permit a compact representation of the observed behavior.

In 4 the matter is further discussed, showing that the filter approach actually proves suitable, providing parametric models that by suitably changing parameters values,

can represent very different simulation *scenarii*. For example, once a general model is identified for a "periodic" task, this can be used, with just different parameter sets, to model a video player, an audio player, and virtually any application that processes data and delivers output in lots at a specified rate. A relevant fact to mention is that an adoption of the filtering approach like that just mentioned, is made possible by the choice – that characterizes this research right from the beginning, see [14] – to model any unforeseen action on the controlled variables as disturbances, in the system-theoretical sense of the term.

Coming to how models are parametrized, and in more than one sense also structured, the most natural way to go is to start from profiling data, and adopt (at least initially) a black box approach. This brings the typical difficulties of input/output identification, with the additional characteristic that in general the inputs of the model are here meant to model some qualification of the behavior of the modeled task at a given time instant (think for example of a time-varying parallelism degree), while the measured and treated data refer to a far shorter time scale. This makes the identification phase potentially critical, as models should not be overparametrized, so as to avoid overfitting on some single task execution trace, and to preserve evidence of the effects that a given parameter (and its variation) produces on the output of the filter. Additionally, as tasks do not possess a precise physical identity, contrary to the objects encountered when identification and filtering are used for example in process control, the filter structure is definitely not known *a priori*.

In any case, once a task modeling framework related to the multicore context is set up and the corresponding task models are ready, it is still necessary to develop a control structure suitable to manage inter-core load balancing, and then, to tune the load balancing policy. To do this, however, a controlled simulation environment is further necessary, compatible with any kind of filter structure, structured to simply allow load balancing and single core scheduling implementation, and capable of providing simulation results in a reasonable short time.

Develop a simulator is also useful from an algorithmic point of view; once the scheduling/load balancing algorithm is completed and placed into the simulator, and once the contained control policy is tuned, said algorithm is ready to be deployed, with (ideally) a few adjustments but in any case with no *conceptual* differences, in an operating system. Finally, in a simulator like that envisaged, there is a plain controlled environment, where scheduling and migration policies can be tested without direct influence of other tasks and the operating system.

After motivating the choice to develop an *ad hoc* simulator, the next sections deal with its simulator development, going from the language choice to the architecture definition, implementation, and testing.

5.2. Developing language: Python

The multicore scheduler simulator (mss for short) treated in this work, has been developed in Python [10]. This choice has different motivations. First, Python is a particularly powerful dynamic programming language, used in many types of applica-

tions. Some of its key features are

- a clear and readable syntax;
- a strong capacity for introspection [1];
- an intuitive management of object-oriented programming;
- a natural ability to express procedural code;
- a completely modular character, with full support for hierarchical structures in modules;
- error handling based on exceptions;
- dynamic high level data types;
- standard libraries, very complete and extendible;
- ease in realizing graphical representations of the data manipulated (i.e. plots, histograms, etc..);
- capacity to extend and integrate modules written with the most popular programming languages as Java, .NET (Java for Jython, or .NET languages for IronPython);
- integrability as a scripting interface for applications written in popular programming languages.

The rapidity of development, combined with the peculiarities of the Python language, are ideal characteristics for the type of simulator to implement. Moreover, the choice of this language allows to easily reuse the code in other projects. Last but not least, relevant motivations for the choice are the open source nature of Python project, and the capacity to immediately execute code on any operating system.

The decision to implement the project with **version 2.7** of Python was driven by the greater diffusion of this version with respect to newer releases [2]. This allows to have a larger amount of support material to develop the project.

5.3. Multicore Scheduler Simulator Architecture

At present, there is no definite agreement on an appropriate control structure to manage load balancing, that incidentally extends far beyond the scope of this thesis. As such, the simulator does not include an implementation of the final components related to load balancing, but it has a complete architecture to easily implement task models and control structures to realize (and tune) multicore scheduling.

[1] Type checking is completely dynamic.
[2] Python versions since 3.x are not compatible with 2.x versions

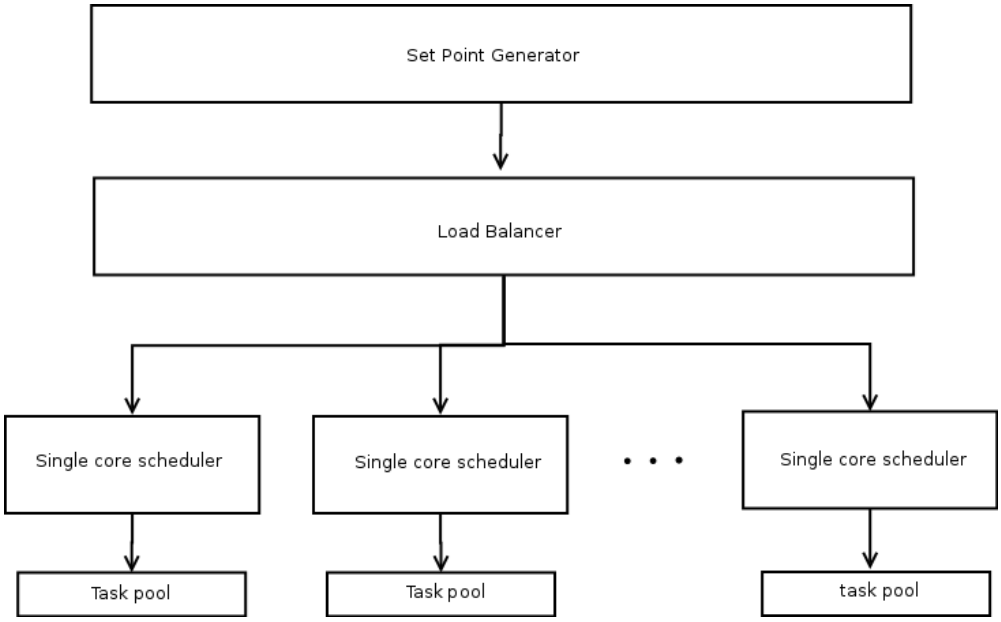

Figure 5.1.: General scheme of the multicore scheduler simulator

In fact, mss has a layered structure, in order to apply a logic of encapsulation between the various levels. The basic idea is that each layer manages data within its competence and pass it to the upper level, conforming to that level's expectations. If an error occurs, an exception is thrown and caught by the proper level. In this way, each level has to manage only its own data and problems, and communicates with other layers in a kind of black box strategy. To realize this, the overall scheduling problem has been decomposed into the following subproblems:

- Set point signals generation,
- Inter core load balancing,
- On core (or core-level) scheduling,
- Task and task pool modeling,

This choice made it easier to exploit work already done about single core scheduling. The set point signals generator is the highest level module; it encapsulates the load balancer, the load balancer encapsulates the schedulers (one for each core), and each scheduler encapsulates a task pool. Figure 5.1 gives a very simple and high-level overview about how the simulator is layered; each layer will be better explained later.
In figure 5.3, the four layers introduced before are recognized.

- Set point signals generation is composed by the SpGen class, which implements the SetPointGenerator interface. Each core implements a control structure similar to the one in figure 5.2. In this way, for a n-core architecture, we have, for every core, the same kind of control structure. The set point generator has to compute vector α for every core (scheduler). This is done in according to the task pool needs, modeled with signals created by multicore context task models (which will be implemented in simulator).

- Inter core load balancing is composed by the MultiCoreLoadBalancer class, which implements the LoadBalancer interface. This module should implement a load balancing policy; at present, the control structure and its implementation are not ready yet, given the effort devoted to identification and modeling, but the placeholder for the mentioned element already exists, since a suitable way to implement load balancing is apparently through a cascade control structure, which migrates tasks to make load homogeneous between different cores.

- On core scheduling; this module reflects quite precisely control structure presented in [14]; figure 5.2 shows the corresponding block diagram. Block $R_t(z)$ is devoted to computing the task bursts b in such a way that each task's CPU time usage τ_t follows the corresponding set point τ_t^o. This set point is obtained by partitioning the measured round duration τ_r according to the (possibly time-varying) vector $\alpha(t)$, the elements of which sum to one. An idle task can be introduced to manage the case in which the total CPU share request is less than one, but this is totally straightforward and has no relevance here. The bursts are additively corrected by signal b_c output by block $R_r(z)$, so that τ_r follow its set point τ_r^o. This is important because if, for example, some task gets blocked, $R_t(z)$ can still keep the CPU time usage for the others, but the round duration set point is lost.

- Task Pool: this, in a real system, is not a proper control layer; tasks are, in some way, the process who should be controlled. In the simulator, however, tasks models are something different and quite independent from the single core scheduler, load balancer and set point generation. Task models derived from profiling and subsequent identification works, are not implemented yet, but simulator has task and task pool interfaces and related single core context implementations, suitable to verify the quality of single core scheduling and easy to extend to multicore context.

The main advantage of this modular approach is that, modules of each levels can be changed, expanded or reduced, without that other levels need to know details; is sufficient that new modules implement same interfaces of the old ones. In example, to implement a single core simulator,a scheduler can be initialized in place of a load balancer and set point generator is able to directly deal with it: LoadBalancer interface and Scheduler interface are made to be both compatible with SetPointGenerator interface, underlying modules (task pool etc) do not need modifications. Moreover,load balancing policy can be changed, for example, taking in account thermal issues and

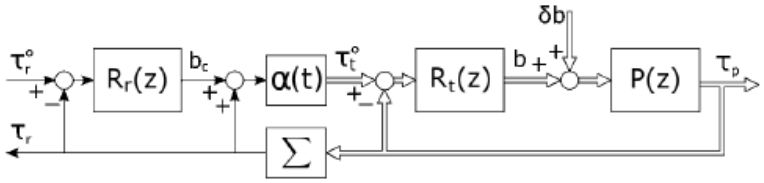

Figure 5.2.: Single core scheduler control structure

this can be done changing only details of load balancer layer (changing some methods implementations), but keeping the whole module compatible with other layers interfaces. In this way, every layer knows details exposed from layers under him and ignores details coming from layers over him.

In example, details about task pools are known at load balancer level, because single core schedulers expose their own task pool, to let inter-core load balance, while load balancer ignores how set point signals are generated and task models structure, it just uses signals generated by set point generator to perform inter core load balancing. With this kind of structure, complexity is divided through different layers, and every layer can be extended in a modular way, without that other layers need modifications. Below, in figure 5.3 general UML class diagram of the simulator is reported, to better clarify simulator structure and hierarchy.

In current simulator version, no load balance policy is implemented, because realization of a controller to perform load balancing is not part of this work, but simulator is ready to be integrated with a load balance controller, all methods necessary to move tasks from a core to an other are already implemented and tested. To test load balancer, tasks have been explicitly moved from a core to an other during simulation, anyway, simulator testing is explained in details in section 5.5.

Main class has the role to initialize all data structures in the correct order, starts simulation and then plots simulation data. Actually, task models obtained by identification phase, described in chapter 4 are not implemented yet, so task models used are the one related to single core context. They are characterized by some nominal parameters:

Periodic tasks:

- Period
- Workload: it is cumulated CPU time that task needs to complete its work over a period
- Beta: beta is a coefficient which represents a real time constrain; task has to reach its workload completion in t(1-beta) since task's first scheduling over a period. Beta has values in the interval $[0, 1]$ and represents the angular coefficient of the line representing cumulated CPU time of the task; higher is beta value, higher is

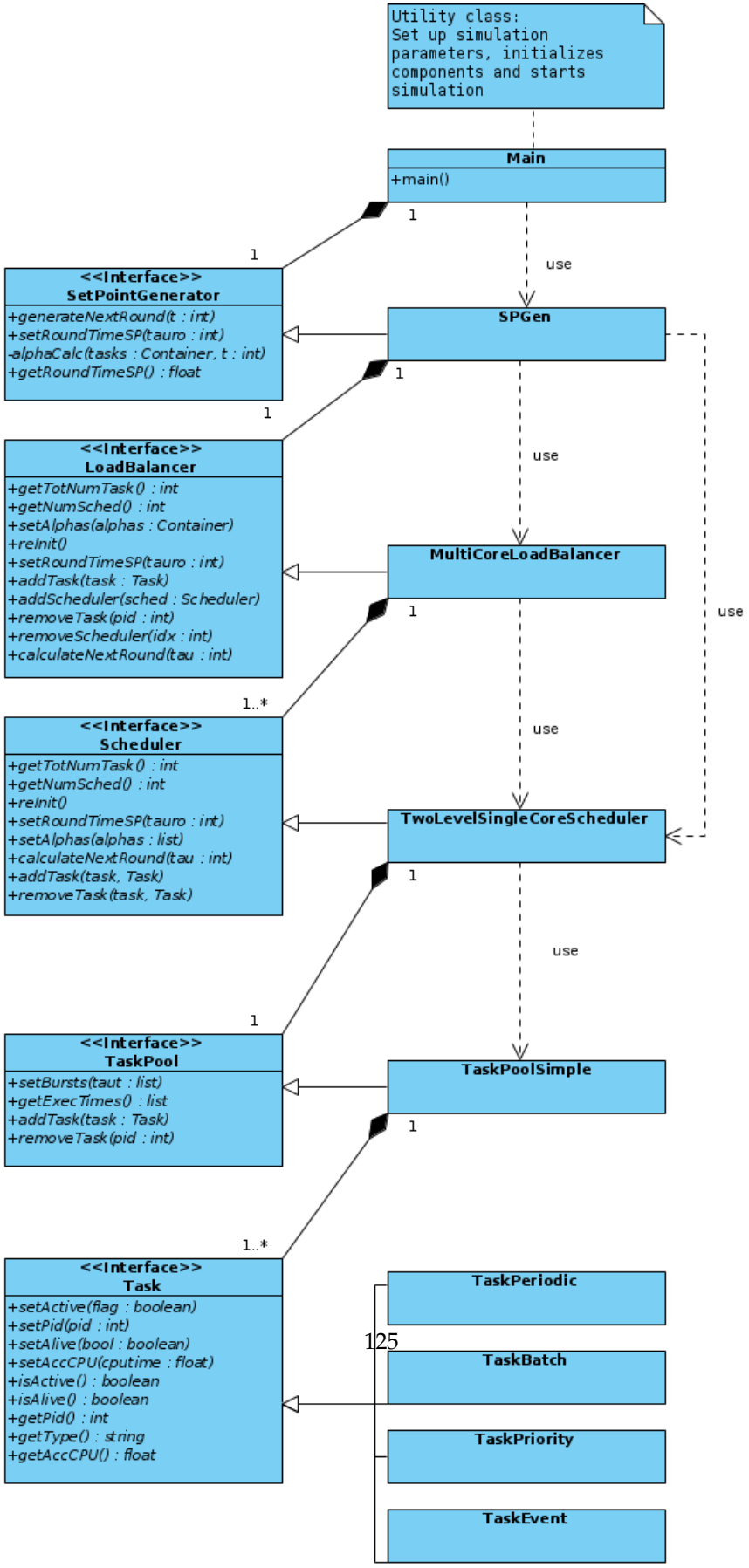

Figure 5.3.: General class diagram of multi core scheduler simulator

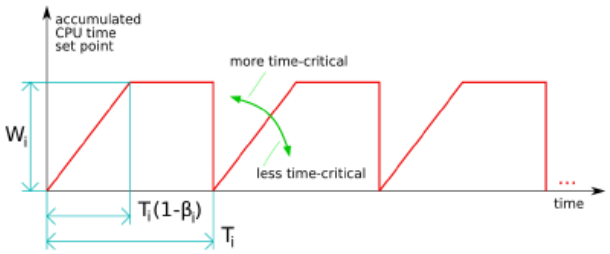

Figure 5.4.: workload and beta in task execution

the slope of the line. in figure 5.4 there's a graphical representation of workload and beta concepts.

Batch tasks:

- Arrival time (absolute time, with respect to simulation beginning instant)
- Duration: it is a relative time; it represents task deadline starting from task arrival time.
- Workload: it is cumulated CPU time that task needs to complete its work.
- Beta: it has the same function of beta explained for periodic tasks.

Priority tasks:

- Priority: it represents task need of responsiveness; it varies on interval $[0, 1]$ it is not related with priority concept present in Linux scheduler; priority is a *real value* variable on [0,1] interval which represents responsiveness degree requested by task.

Event based tasks:

- Triggering times: it is an absolute time (with respect to simulation beginning instant); it represents triggering task instants
- Alpha decay base: event based tasks get CPU time according to a decreasing exponential function of the type $b^{-(t-\tau)}$ b is the alpha decay base, τ is the last triggering instant.
- Alpha0: it represents initial tentative value to calculate execution burst. It has function of initializing value, it is overwritten by scheduler at every simulation step.

5.4. Multicore Scheduler Simulator Implementation

In this section details related to Multicore Simulator implementation will be explained; in particular, discussion will be carried out layer by layer, explaining modules details and their relations with other modules. To support discussion, some UML schemes (in particular class diagrams) are reported; notation is conform to UML2, but to remark some particular python constructs implementations, some special notation are used.

Before starting exposition, a further clarification is needed: according to UML2 notation, abstract methods should be reported with italic text. Often, due to image size and layout constraints, italic text is not easily distinguishable from normal text, so here it is remarked that every interface class is composed **only** by abstract methods. This remark is very important, because the presence of non abstract methods inside an interface declaration, violates the definition of interface. Scheduler implementation is presented with a bottom-up approach, so that is easier to understand modules layer division and aggregation.

5.4.1. Task and TaskPool implementation

For simulation purposes, different task categories share some data structures, useful to manage basic task execution simulation aspects, as cumulated execution time, active and alive flag and so on. These data structures comes directly from Task interface definition and will be discussed before going into the details of the different tasks categories. They are present in every Task implementation presented in this work.

Common attributes:

- `pid:int` it is process identifier; it represents uniquely task inside system; even though the methods to manage pid, for extensibility reasons, are public, It should be managed only by task pool, or directly set by the constructor.
- `alive:boolean` it is a boolean which indicate whether task is currently waiting in a scheduler queue to be executed. This field is general, every task has this flag, because, for implementation reasons related to simulation data plotting (and exporting) explained later, every time a task is migrated, a deep copy of him is made and put into an other scheduler queue; the old copy is set not alive. This solution is not the best from memory efficiency point of view, but it allows to avoid reordering every auxiliary scheduler queue every time a task is migrated.
- `active:boolean` it is a boolean field, it tells if the task is active and waiting to be executed.
- `accCPU:float` it is a float which represents how much time task has already spent in execution; it stands for cumulated CPU time. In real systems, time flows forward naturally, but for simulation purposes, time passing must be managed explicitly.

- `type:string` represents task category: it can assume only values periodic, batch, priority, event.

Common methods:

- `hash():string` overrides standard hash() method implemented in python; python uses dictionary based data structures, overriding hash methods is a good practice to have control of how language associate dictionary keys to data structure instances.
- `eq(other:Task):boolean` overrides standard comparison operator ==. It defines when two Task instances are equal. Each Task subclass has a specific implementation of this method.
- `neq(other:Task):boolean` it is the dual of eq() method.
- `str():string` it gives a string representation of the Task object; each Task subclass has a specific implementation of this method.
- `setAccCPU(cputime:float)` set cumulated execution time collected by task
- `getAccCPU():float` returns cumulated execution time collected by task.
- `getPid():int` returns process identifier
- `setPid(pid:int)` set process identifier. This method is public and common to all Task implementations, to let Task object be compatible with different TaskPool implementations, but pid should be managed only by TaskPool or other higher level modules (Scheduler or LoadBalancer), in order to avoid errors on pid management.
- `isAlive():boolean` returns a copy of the field `alive` for read purposes.
- `isActive():boolen` returns a copy of the field `active` for read purposes.
- `setActive(flag:boolean)` setter method: it turns a task active (and ready to be executed) or inactive.
- `setAlive(bool:boolean)` setter method: it turns a task alive or dead on a given task pool queue.
- `getType():string` returns task's category string representation.

A little remark on difference between `alive` field and `active` field: `active` flag represents a condition quite similar to the ready state on a real OS. A non active task has its related $\alpha = 0$ and will not be taken in consideration for next round's bursts calculus, but is present in task pool and at every simulation round is taken in consideration to decide if it will become active or not at the beginning of the next round. `alive` flag represents whether task is dead or alive. a dead task's reference

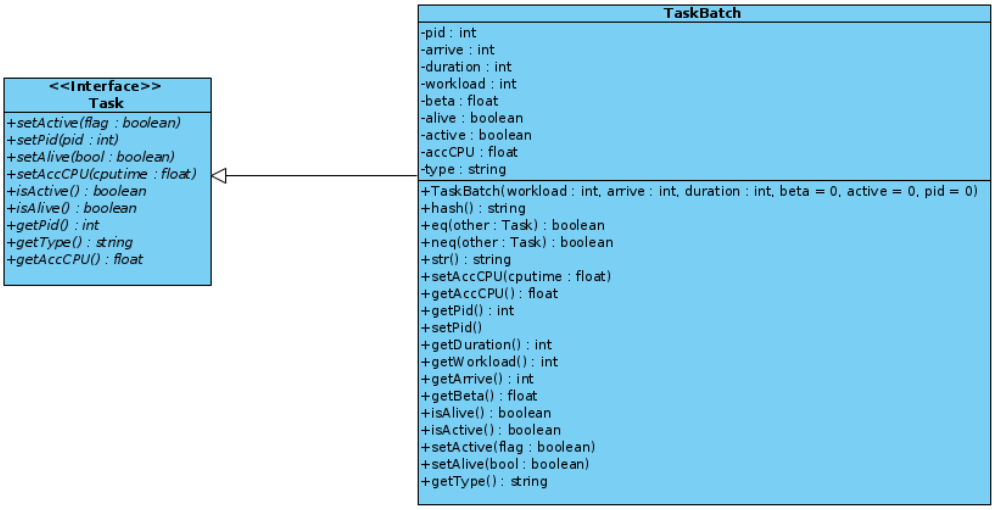

Figure 5.5.: UML class diagram of TaskBatch class

remains in the task pool to address problems related to simulation data storing and plot. Python automatically reorders a list when an element is removed, but this is not a problem from, strictly speaking, scheduling point of view, but it is a problem when simulation data have to be plotted (because not all data list have the same length etc). So, when a task is removed, it is not physically removed from the task list, but its `alive` flag is set to zero. A task having alive flag set to zero, is totally ignored during scheduling. From a practical point of view, a not alive task has only a placeholder function, preserving at zero all its simulation data.

5.4.2. Task Batch

in figure 5.5 is reported class diagram related to batch tasks representation. TaskBatch class represents a batch task; in this work, a batch task is intended as a task who is spawn at a certain time and that principally needs CPU time, memory and data (coming from hard disks or other devices) for execution, with a few (or none) user interaction. Attributes and methods come directly from this type of characterization. Here are described only peculiar attributes and methods that distinguish batch tasks from other tasks.

Attributes, they are all private fields, to access them, getter and setter methods are provided.

- `arrive:int` it is task's spawning instant, when it first came into system for execution.
- `duration:int` it is task duration; it represents a temporal deadline; task has

to complete its workload within duration interval time. with respect to figure 5.4 T_i represents duration.

- `workload:int` represents cumulated execution time necessary to complete task execution. In figure 5.4, W_i is task workload.
- `beta:float` beta function is explained in previous section: anyway beta is a coefficient which represents a real time constrain; task has to reach its workload setpoint in $T(1-\beta)$ so, it represents rapidity needed by task to complete its workload. In figure 5.4 a graphical explanation is given.

Methods

- `TaskBatch(workload, arrive, duration, beta=0, active=0, pid=0);` this is the constructor, some fields, if not specified, have default value.
- `getDuration():int` getter method: it returns task's cumulated execution time; it returns an integer for simplicity, because in simulation only task with an integer duration have been used. anyway python has dynamic typing and binding, so type indication is purely indicative; a float duration can be used and it will be correctly managed. In fact simulation time is continuous.
- `getWorkload():int` getter that returns task's workload. It returns an integer for simplicity, but consideration made for `getDuration()` method are still valid.
- `getArrive():int` getter method that returns task's arrival time. As before, int return type is purely indicative; it has been used to spot a precise simulation step, i.e. if a certain task has arrive = 3, it means that task has been spawn during the third simulation step. it is important to remark that burst granularity is not integer, in fact time flows with continuous granularity, at each simulation step, an execution round is complete, and it has not, in general, an integer duration. Indeed, control works over discrete granularity (scheduler works by discrete steps). This remark is important to do not confuse time flow with control steps. Stated that simulation time flows with continuous granularity (and so task CPU burst have a non integer duration, from time point of view) is the reason why integer durations do not reduce work generality.
- `getBeta():float` returns a copy of task's beta.

5.4.3. Task Periodic

In figure 5.6 is reported class diagram related to periodic tasks representation. In this work, a periodic task is intended as a task which needs a certain amount of CPU time periodically. two simple examples of periodic tasks that we intended to represent are audio players and video players, but we can imagine other examples, as periodic

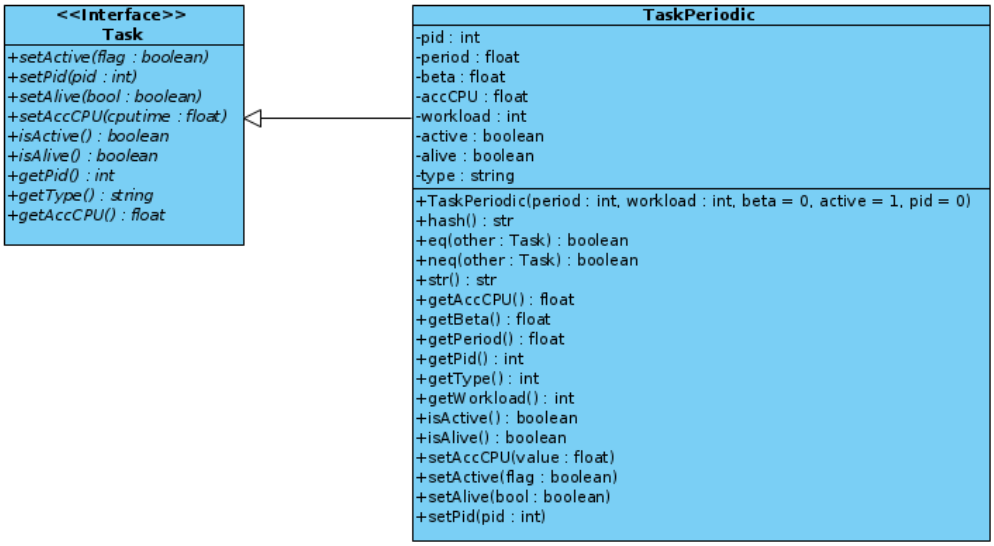

Figure 5.6.: UML class diagram of TaskPeriodic class

data transmission in wireless sensor networks and so on. If we compare a single period execution of a periodic task with a batch task, we found similar characteristics: both have a setpoint workload to reach within a temporal deadline, both can have constraints related to the rapidity needed to reach workload setpoint. The main differences are that normally a batch task executes once, while a periodic task executes periodically. These differences are reflected by TaskPeriodic class implementation:

below are described only attributes and methods that distinguish periodic tasks from other ones.

Attributes:

- `period:int` it is period which characterizes task periodic execution.

Methods:

- `TaskPeriodic(period:float, workload:int, beta = 0, active = 1, pid = 0)` it is the constructor. task period is defined by constructor and it can't be further modified.
- `getPeriod():float` getter that returns a copy of task period.

5.4.4. Task Priority

In figure 5.7 is reported class diagram related to priority tasks representation. In this work, a priority task is intended as a task which is highly I/O bound, like text

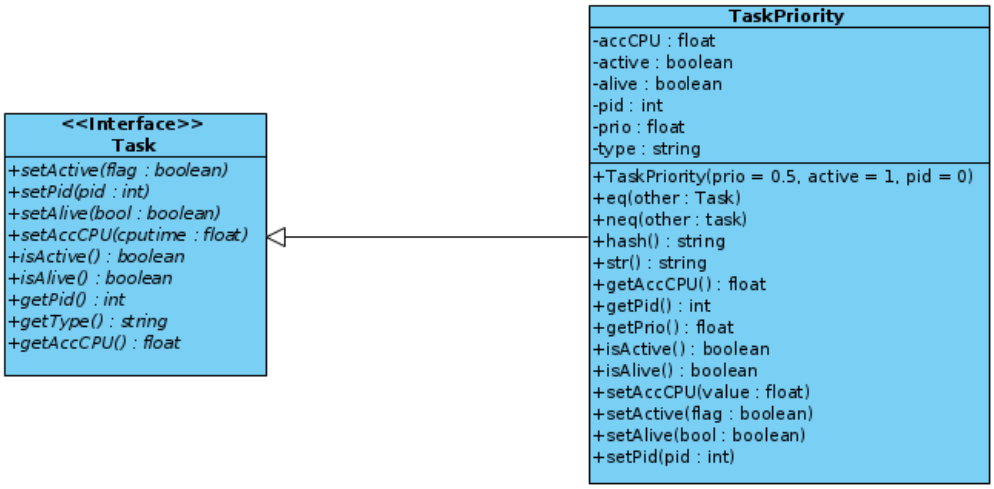

Figure 5.7.: UML class diagram of TaskPriority class

editors and file managers. Typically they are not particularly heavy, once they have been spawn and are ready to receive user input, but they usually require a good system responsiveness to ensure good user experience. This is the reason why in this representation, TaskPriority has not a duration field; it is considered always active, waiting for user input. Initial set up phase, where task is launched and loads data and user interface, can be modeled as a batch task. [3]

Priority task characterizing attributes are:

- `prio:float` it represents task priority; it varies on $[0, 1]$interval, where 0 is the minimum priority, and 1 is the maximum one (real time priority) `prio` represents responsiveness degree needed by task.

Methods:

- `TaskPrio(prio=0.5, active=1, pid = 0)` it is the constructor; `prio` field is initialized by constructor and it can't be further modified. if it is not explicitly defined, `prio` is set to 0.5.
- `getPrio()` getter method: it returns a copy of `prio` field.

5.4.5. Task Event Based

In figure 5.8 is reported class diagram related to event based tasks representation. In this work, an event based task is intended as a task that needs to be executed only

[3] to make model identification, initial set up phase has been discarded.

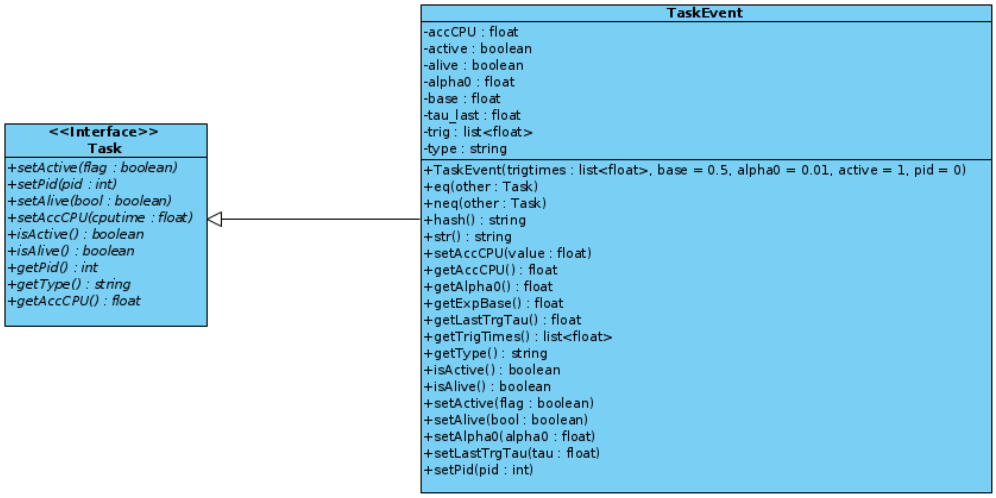

Figure 5.8.: UML class diagram of TaskEvent class

after that a certain event (or group of events) occurs. it is highly I-O bound (for example, mouse interrupt handler is a light, I/O bound task), but it could be also CPU bound and memory bound, just think about the requests made to a web server. Models for CPU and memory bound event based task are discussed in chapter 4, in the simulator light I/O bound event based task are modeled [4]. Light I/O bound event based tasks, are intrinsically active (like priority tasks) and normally need a good system responsiveness (just think about mouse interrupt handler), but they usually do not need a big amount of CPU time. To model this kind of behavior, when triggered, they receive CPU time according to a decreasing exponential function, of the type $b^{-(t-\tau)}$. TaskEvent class implementation derives directly from this representation.

Attributes:

- `alpha0:float` it represents initial α value used to calculate initial burst (by default is a quite low value). This value is overwritten by scheduler at every simulation step.
- `base:float` it is the base used to calculate CPU time allocation by the formula $b^{-(t-\tau)}$ where b is `base` and t is time; τ is the last triggering instant (last time that triggering event has occurred).
- `tau_last` it is the last time instant where a triggering event has occurred; in the formula reported above is τ.
- `trig` is a list of float; it contains all time instants in which a triggering event occurs.

[4] CPU/memory bound tasks can be treated as batch tasks, with variable interarrival times

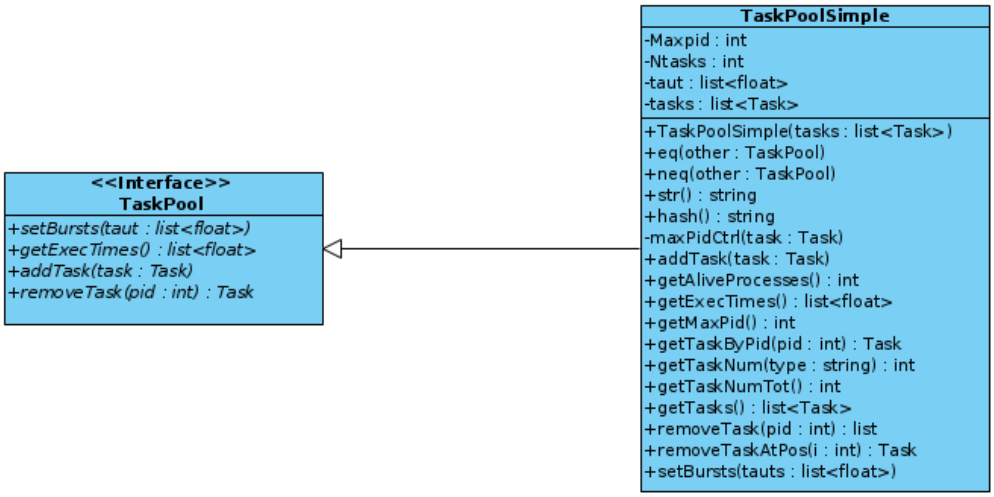

Figure 5.9.: UML class diagram of TaskPoolSimple class

Methods:

- `TaskEvent(trigtimes, base=0.5 , alpha0=0.01, active=1, pid=0)` constructor method: some parameters are set by default (but they can be overwritten); trigtimes is the list of triggering instants and should be not empty. This list is set by constructor and it is not editable.
- `getAlpha0():float` it returns a copy of the `alpha0` field.
- `getExpBase():float`it returns a copy of the `base` field.
- `getLastTrgTau():float` getter method: it returns a copy of the `tau_last` field
- `getTrigTimes():list<float>` getter method: it returns a deep copy of `trig` list.
- `setAlpha0(alpha0:float)` setter method: it overwrites `alpha0` field.
- `setLastTrgTau(tau:float)` setter method: it overwrites `tau_last` field.

5.4.6. Task Pool

Figure 5.9 shows Task Pool interface and the implementation used by simulator. Task pool is an abstraction used to model a set of Task, in general, belonging to different categories (batch, priority etc). It presents data structures and operations necessary to manage an ordered set of Task objects.

Attributes:

- `Maxpid:int` an integer variable used to keep trace of the maximum process identifier contained in the task pool
- `Ntasks:int` an integer variable used to keep trace of the number of tasks present in task pool.
- `taut:list<float>` it is the list of time slices given at each task; this list is recalculated at every simulation step by Scheduler module.
- `tasks:list <Task>` it is task pool's task list. each list element has to implement Task interface.

Methods:

- `TaskPoolSimple(tasks:list<Task>)` it's the constructor: it initializes a new TaskPoolSimple object. `tasks` has to be a non empty list of Task objects.
- `eq(other:TaskPoolSimple)` overrides == operator. Two TaskPoolSimple objects are equal only if they have the same number of tasks and every task in `tasks` list is equal to the corresponding one in `other.tasks` list.
- `neq(other:TaskPoolSimple)` overrides ! = operator. Is the dual of the `eq(other:TaskPoolSimple)` method; two TaskPoolSimple objects are different if they do not have the same number of tasks in respective task list or if, at least one task is different from the corresponding one in `other.tasks` list.
- `str():string` returns string representation of the task pool: it is a list of string, every string is the result of `str():string` method invoked on corresponding Task element.
- `hash():string` overriding of standard `hash()` method.
- `maxPidCtrl()` it's a private support method to keep up to date `Maxpid` field after task insertion/removal.
- `addTask(task:Task)` adds a Task object to the task list.
- `getAliveProcesses():int` returns number of alive processes present in the task pool.
- `getExecTimes():list<float>` returns a deep copy of `taut` data structure. it is useful to calculate burst correction at Scheduler level.
- `getMaxPid():int` getter methods: it returns a copy of `Maxpid` field; it's useful to manage pid assignment at LoadBalancer level.
- `getTaskByPid(pid:int)` returns a deep copy of the task identified by pid passed as parameter; if task is not in task pool, a ValueError exception is raised.

- `getTaskNum(type:string):int` returns number of tasks of a given type; type can assume values 'periodic', 'priority', 'batch', 'event'.
- `getTaskNumTot():int` returns total number of tasks present in task pool.
- `getTasks():list<Task>` returns the reference to task list. This method is public and exposes `tasks` field; it should be used with caution. it is implemented to let Scheduler and LoadBalancer access directly to the task pool.
- `removeTask(pid:int):list` Removes task with a certain process identifier. For implementation reasons, in particular to store, for every scheduler, lists of simulation data having all the same length (at the end of simulation), avoiding troubles when simulation data are plotted, tasks are not physically removed from task list, because, after removing, task list would be automatically resorted to maintain no empty elements, reducing its length. So, this method doesn't physically remove tasks: it sets alive flag into the task (`alive=0`). This flag tells whether task is alive or not in the task pool and maintain list order and length, avoiding plotting troubles. In real scheduler, task can be safely removed from task list, because bursts vector and alpha vector are completely recalculated at every scheduling round. If task is found, method returns a list of 2 elements: result[0] is position index of removed task; result[1] is a deep copy of removed task. If task identified by pid is not found, a ValueError exception is raised.
- `removeTaskAtPos(i:int)` removes task at position `i` in task list. Task is not physically removed, just `alive` flag is set to zero, as it is remarked with respect to `removeTask(pid)` method. If index `i` is not correct, a TypeError or ValueError exception is raised.
- `setBursts(tauts:list<float>)` set current simulation step tau vector calculated by Scheduler and refreshes accumulated CPU time related to each task of the task pool. To simulate real system execution, bursts are affected by a Gaussian noise, to simulate the fact that sometimes tasks remain in execution for a time a little bit longer (or shorter) respect to CPU quantum assigned by scheduler (reasons are different: critical section to be completed, interrupts events etc).

5.4.7. Single core scheduler implementation

Single core scheduler layer is one of the most important of the whole project; it manages taskpool and calculates CPU burst for each task present in the pool. Simulation data are saved in suitable data structure to let plotting and exporting them on file for further manipulation. TwoLevelSingleCoreScheduler is the class which implements Scheduler interface; it contains all data structures necessary for scheduling simulation and all auxiliary data structure to let plotting and data export. For two reasons auxiliary data structure are not reported in the class diagram:

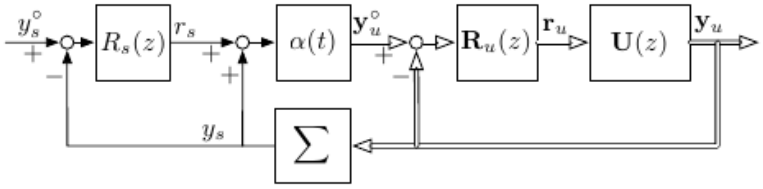

Figure 5.10.: Block diagram of the control structure in general, evidencing the controllers in the ULC and the SLC.

1. auxiliary data structure are not conceptually related to Scheduler; in this implementation, for simplicity, are managed by scheduler, but they should be managed by a stand alone auxiliary module. It could complicate a bit project architecture and reduce memory efficiency, but will result in a more clean and clear scheduler implementation.

2. readability: reporting all auxiliary methods and data structure would result in a huge and quite useless class diagram, reducing its efficacy.

TwoLevelSingleCoreScheduler class implementation makes reference to control structure shown at the beginning of the section. To better exploit two level control structure, it can be reformulated as it is shown in figure 5.10.

According to this structure, and given the necessity of minimizing the scheduling overhead, simple User Level Control (ULC) and System Level Control (SLC) controllers should be preferred. Based on previous works, there are some typical choices. The ULC controller can either be an integral or a deadbeat one, while the SLC controller can be a PI, a pure proportional, or, again, a deadbeat one. Using I/P/PI blocks results in the lowest orders, while deadbeat blocks may require more stages, but allow to shape the set point responses. For example, this can be useful if it is required that when the α component for a task is increased, the CPU allocation to it has an initial overshoot. However, since this matter was already discussed and is not central to this work, no further details are reported.

With reference to figure 5.10, Set point and alpha generation are managed by setpoint generator layer and this discussion is forwarded to section 5.4.8. Other works, as reported before, has suggested that ULC and SLC should be of the I and PI type respectively. These controllers, which have transfer function $R_s(z)$ and $R_u(z)$ respectively, have the following structure:

$$R_u(z) = diag\left(\frac{K_u}{z-1}\right)$$

$$R_s(z) = K_s * \frac{z - zero_{slc}}{z-1}$$

At each scheduler round, $R_s(z)$ block calculates burst correction r_s to minimize difference

$$y_s - y_s^o$$

y_u^o is single task CPU burst set point and it depends on α vector, r_u is single task CPU burst calculated by control and y_u is actual CPU burst really used by single task (y_u is affected by noise). y_s is round duration and it is the sum on all the tasks of the actual CPU bursts y_u y_s^o is scheduling round set point and, together with alpha vector, is set by setpoint generator.

Given the block scheme and the transfer functions reported, if we call x_{SLC} the state variable of the SLC, burst correction calculus is given by the formulas:

$$x_{SLC} = x_{SLC} + kSLC * (1 - zero_{slc}) * (y_s^o - y_s)$$
$$r_s = xSLC + kSLC * (y_s^o - y_s)$$

After that burst correction r_s and alpha vector (calculated by setpoint generator) are ready, single task burst setpoint can be calculated; let i be the index of the $i-th$ task,

$$y_u^o[i] = alpha[i] * (y_s + r_s)$$

And then, single task burst is calculated in this way:

$$r_u[i] = r_u[i] + kULC * (y_u^o[i] - y_u[i])$$

where $r_u[i]$ at the right of the equal is the previous value of single task burst calculated by controller (control structure works in feedback). Then burst are saturated:

$$r_u[i] = max(0, min(bmax, r_u[i]))$$

These formulas clarify furthermore why $R_s(z)$ is the system level controller and why $R_u(z)$ is the user (task) level controller. SLC calculates burst correction r_s trying to nullify difference between round time setpoint and round time. Whit round, it is intended the sum of all the timeslices given to the various tasks. In this way, SLC works at system level, taking care about the duration of the whole execution round. ULC takes care about single task timeslicing; once that round duration and burst correction are fixed, through alpha vector, timeslicing for each single task is made, fixing single task burst setpoint and ULC tries to nullify difference between single task actual burst and its burst setpoint.

TwoLevelSingleCoreScheduler class reported in figure 5.11 implements two level scheduler shortly described here.

Attributes:

- `id:string` is the scheduler text identifier, useful to have a text reference to various schedulers, especially when simulation data are plotted.

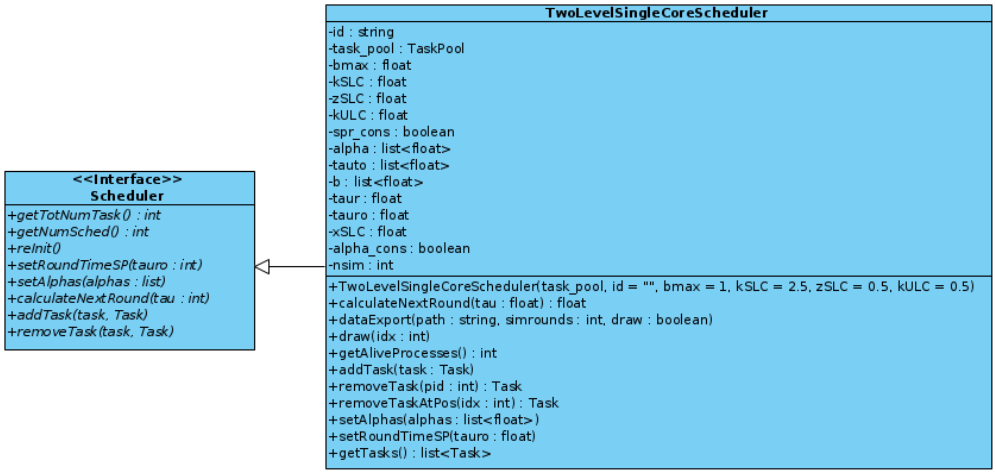

Figure 5.11.: UML class diagram of TwoLevelSingleCoreScheduler class

- `task_pool:TaskPool` is the reference to TaskPool object managed by TwoLevelSingleCoreScheduler class. it contains all the task which should be executed by a certain scheduler.
- `bmax:float` is the maximum CPU burst assignable to a certain task. it is useful to insert a saturation in next round's bursts calculus. It should avoid the phenomenon of integral charge and the occurrence of excessively unbalanced situations in the allocation of CPU time.
- `kSLC:float` is the gain of the SLC.
- `zSLC:float` is the zero of the SLC [5] it corresponds to $zero_{slc}$ parameter.
- `kULC:float` is the gain of the ULC.
- `spr_cons:boolean` is a flag used to indicate whether round time setpoint has been set or not.
- `alpha:list<float>` is alpha vector, needed to realize timeslicing between different tasks. it is calculated by SetPointGenerator module
- `tauto:list<float>` is the list of single task burst setpoints; it corresponds to y_u^o signals
- `b:list<float>` is the list of single task bursts calculated by controller; it corresponds to r_u signals.

[5] A zero is a value that given to the Laplace variable z makes numerator of a transfer function equal to zero.

- `taur:float` is actual round duration: it is a single value because is given by the sum on all tasks of their actual burst y_u where $y_u = r_u + \delta b$ where δb is an exogenous noise. It corresponds to y_s signal.
- `tauro:float` is round duration setpoint. it is calculated by SetPointGenerator module and corresponds to y_s^o signal.
- `xSLC:float` is SLC state variable. [6]
- `alpha_cons:boolean` flag used to know whether alpha vector has already been calculated or not (useful for an eventually parallel implementation).
- `nsim:int` keeps trace of current simulation step.

Methods:

- `TwoLevelSingleCoreScheduler(task_pool, id='', bmax=1, kSLC=2.5, zSLC=0.5, kULC=0.5)`
  it is constructor method; it needs a TaskPool object (already initialized). other parameters, as maximum burst allowed `bmax`, SLC gain, ULC gain and SLC zero position can be specified, otherwise default safe values are used.
- `calculateNextRound(tau:float)` calculate next round duration and next round single task's bursts according to control structure explained before. Parameter `tau` is the duration of the current round, passed by SetPointGenerator.
- `dataExport(path:string, simrounds:int, draw=True)` it is a support method to save simulation data into a textual file. `path` is the absolute path of the file, `simrounds` is the number of rounds simulated and `draw` is a flag used to tell whether to plot simulation data after data exporting or not.
- `draw(idx:int)` it is a support method, used to plot simulation data of the current Scheduler.
- `getAliveProcesses():int` returns the number of alive tasks (tasks with alive flag set to one) present in the underlying TaskPool.
- `getTasks():list<Task>` getter method: it returns scheduler's tasks list (useful to let alpha vector calculus performed by SetPointGenerator).
- `getMaxPid():int` getter method: it returns max process identifier contained by the underlaying TaskPool object.
- `addTask(task:Task)` adds object passed as parameter to the underlaying task pool; parameter has to implement Task interface.

[6] SLC has a PI structure, so it is a dynamic system.

- `removeTask(pid:int):Task` set alive flag of the Task identified by pid to zero and returns a deep copy of the task. If task is not present in the task pool, propagates ValueError exception launched by task pool.
- `removeTaskAtPos(idx:int):Task` set alive flag of the task in position `idx` in the task pool's task list to zero.
- `setAlphas(alphas:list<float>)` set alpha vector calculated by set point generator into scheduler's data structures. Alpha vector has to be not null and its length has to be equal to the number of tasks contained in underlaying task pool.
- `setRoundTimeSP(tauro:float)` setter method: it sets round time setpoint in the Scheduler's data structures.

5.4.8. LoadBalancer implementation

Load balancer module has been made to simulate management of work load on multicore architecture. The idea behind this module is that on a multicore architecture, let n be the number of independent cores, timeslicing on every core is managed by a TwoLevelSingleCoreScheduler, through the strategies described in section 5.4.7, and this control layer has to act, usually, fast (i.e every millisecond burst have to be recalculated), so controller acts in a certain frequencies interval, which represents on core control band. LoadBalancer module, instead, has to make different independent core's task pools as much homogeneous as possible. With homogeneous task pools, it is intended that each task pool has to present a mix of task belonging to the four categories stated in section 3.1.2 (periodic task, batch task, priority task and event based task) as homogeneous as possible in terms of system resources (or system responsiveness) needing. For example, if on a quad core architecture there is a certain number of tasks and four tasks are batch tasks, with heavy requests in memory allocation and CPU time, they shouldn't be scheduled all four on the same core. If this happens, LoadBalancer has to react and migrate three tasks on three different cores. Similarly, all priority tasks shouldn't be scheduled on the same core because it probably would case cache saturation and cache miss increase, so if this happens, some tasks should be migrated (probably the ones with smaller cache footprint). These simple examples, anyway, introduce the intuitive idea that each core should work on a task pool composed by a certain number of batch task, a certain number of periodic ones, priority and so on.

The exact number can't be fixed a priori, it depends on single tasks resource needs, but, in well controlled system, each core should be characterized by similar amounts of cache references, CPU utilization and cache miss rate. If it is not some tasks should be migrated. This statement further clarifies why in multicore context, task model can't be a simple time delay (as it is in single core architectures), but should be improved and refined, taking in account task resource needs and the temporal dynamic that characterizes these needs; we are not interested in cumulative data, as **how much** cache references have been allocated to let task complete execution, or

how many clock cycles have been totally spent, to make load balancing decisions, we need to know transients, **what temporal dynamic** has cache references allocation? **What temporal dynamic** has execution times (ton) and sleep/wait times (toff)? **What temporal dynamic** has cache miss counting? This is the reason why profiling real tasks and deriving models referred to their system behavior (as described in chapters 3 and 4 is useful. Once models are ready, it is possible to implement new task models (MulticoreTask), which produce realistic signals representing cache reference, ton, toff and cache miss; on the base of these signals and of overall system load, an extended SetPointGenerator (MulticoreSetPointGenerator) will generate suitable setpoints and load balancing control can be realized through a linear control structure (maybe P/PI or something similar).

This thesis work has not gone so far, MulticoreSetPointGenerator and MulticoreTask are not implemented, but simulator architecture is completely able to easily integrate these future extensions. Anyway, coming back to load balancer focus, after these brief comments, is quite evident that load balancing control should be 'slower' than single core scheduling, to let single core scheduling reach an equilibrium and then balancing system workload between different cores. If load balancing control works in the same frequencies interval of single core scheduler, we would have a system that tries to migrate a task every time that a CPU burst is recalculated. It easy to imagine that a similar policy would take to an unstable system.

This work doesn't include load balancing control structures or extended task models implementation. MultiCoreLoadBalancer module, shown in figure 5.12 implements methods to divide alphas calculated by SetPointGenerator between different Scheduler modules, to migrate tasks between different schedulers and to 'manually' add and remove tasks from system.

MultiCoreLoadBalancer attributes:

- `Nsim:int` a variable to keep trace of simulation step
- `NsysTask:int` a variable to keep trace of total number of tasks present in the system.
- `actual_sched:reference<Scheduler>` it is a reference to the current Scheduler object
- `alpha_sc:list< list<float> >` it is a list of float lists; it is internally managed by MultiCoreLoadBalancer to divide alpha vector, which is calculated by SetPointGenerator at system level, without taking in account how tasks are divided into different scheduler's task pool. every element of `alpha_sc` is alpha vector related to the corresponding scheduler. i.e. on a quad core architecture, alpha_sc[0] contains alpha vector of scheduler 0, alpha_sc[1] contains alpha vector of scheduler 1 and so on.
- `alphas:list<float>` it is the data structure used to save alpha vector passed by SetPointGenerator module.

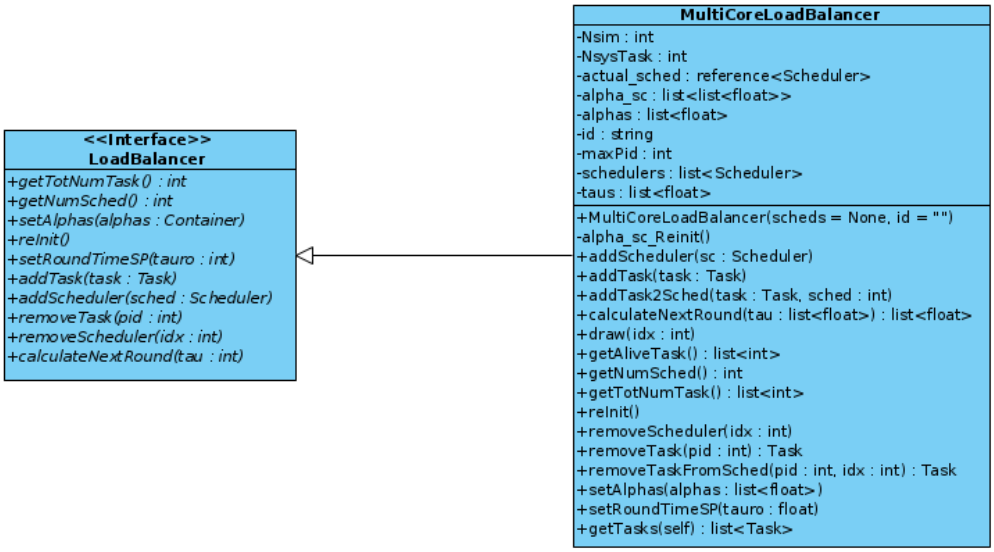

Figure 5.12.: UML class diagram of MultiCoreLoadBalancer class

- `id:string` it is the text identifier related to MultiCoreLoadBalancer object (its 'name').
- `maxPid:int` it is the maximum process identifier present in the system. useful to automatically manage pid assignation.
- `schedulers:list<Scheduler>` it is the list of Scheduler objects managed by MultiCoreLoadBalancer.
- `taus:list<float>` this list contains previous round duration of every Scheduler object. Needed to let each Scheduler calculate next round duration.

Methods exposed:

- `MultiCoreLoadBalancer(scheds = None, id='')` it's the constructor: `scheds` parameter should be a list of Scheduler objects.
- `alpha_sc_Reinit()` it's a private support method: reinit alpha_sc; each element of the list will be reinitialized to a list of zeros having the same length of the corresponding scheduler's task list.
- `addScheduler(sc:Scheduler)` adds a Scheduler object to the MultiCoreLoadBalancer instance.
- `addTask(task:Task)` adds a Task object to the Scheduler referenced by actual_sched field.

- `addTask2Sched(task:task, sched:int)` adds a task object to the scheduler at position indexed by `sched` in load balancer's scheduler list.
- `calculateNextRound(tau:list<float>):list<float>` calculates next round duration calling `calculateNextRound(tau:float)` method on every Scheduler belonging to `schedulers` list. collects new rounds duration in a new list and returns it.
- `draw(idx:int)` it's a support method to plot simulation data referred to scheduler in position `idx` in underlaying scheduler list.
- `getAliveTask():list<int>` it returns a list; each element of the list is the total number of alive tasks present in the corresponding scheduler.
- `getNumSched():int` getter: it returns number of schedulers owned by MultiCoreLoadBalancer.
- `getTasks():list<Task>` it returns overall system tasks list (useful to let alpha vector calculus performed by SetPointGenerator).
- `getTotNumTask():list<int>` it returns a list; each element of the list is the total number of tasks present in the corresponding scheduler.
- `reInit()` reinitializes internal MultiCoreLoadBalancer state.
- `removeScheduler(idx:int)` removes scheduler at position idx of current MulticoreLoadBalancer scheduler list. Then internal MultiCoreLoadBalancer state is reinitialized.
- `removeTask(pid:int):Task` removes task identified by pid passed as parameter; pid has to be a valid process identifier. If task belongs to one of the schedulers contained in load balancer, its alive flag is set to zero and a deep copy of the removed task is returned, else ValueError exception is propagated.
- `removeTaskFromSched(pid:int idx:int)` set to zero alive flag of the task identified by pid from scheduler in position idx in scheduler's list. A deep copy of the task is then returned. If task is not found, a ValueError exception is propagated.
- `setAlphas(alphas:list<float>)` set alpha vector in load balancer data structures; this alpha vector refers to all tasks present in the system and is calculated by setpoint generator. alphas is a list of floats (generally positive).
- `setRoundTimeSP(tauro:float)` method to set round time set point in MulticoreLoadBalancer data structures; parameter `tauro` is round time set point, it is calculated by setpoint generator and it has to be positive.

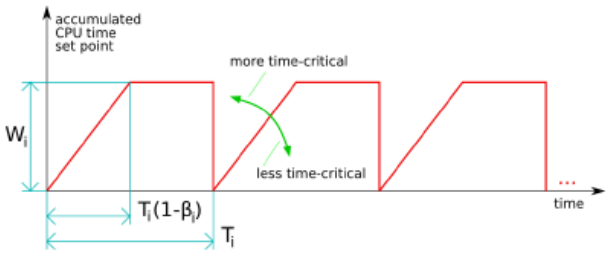

Figure 5.13.: Accumulated CPU time for periodic tasks

5.4.9. SetPointGenerator implementation

Setpoint generator is an other important module of the simulator. This module is compatible with both LoadBalancer objects and Scheduler objects (to directly realize a single core scheduler simulator). It get task list from the underlaying module and calculates alpha vector, which is essential to realize task's timeslicing at core level. From a conceptual point of view, is interesting to deepen the discussion about how alpha coefficients are calculated. They can't be separated from the belonging category of the task, in other words, alpha calculus related to a priority task is structurally different from alpha calculus related to a periodic task. Let proceed discussing it category by category:

Alpha calculus for periodic tasks

Let T_i be the period of a task, and W_i its workload in the period (i.e., every T_i time units the task must receive the CPU for W_i time units). The accumulated CPU time in the period needs thus following the trapezoidal profile of figure 5.13, the start of which is triggered by a periodic interrupt, while the end can be required by the task once its per-period workload is accomplished (all the major operating systems provide primitives for such a signaling). This means that at the beginning of each period the task will be allotted a *tentative* α component, denoted by $\hat{\alpha}_i$, given by

$$\hat{\alpha} = \frac{W_i}{(1-\beta_i)T_i\tau_r^o}, 0 \leq \beta_i < 1$$

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

hard real-time constraints, a matter however too vast to be addressed here.

Alpha calculus for batch tasks

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

Alpha calculus for priority tasks

These tasks have no deadlines: this is the typical situation for interactive tasks, such as desktop applications. Since the proposed scheduling framework uses neither deadlines nor priorities, and in some sense section 5.4.9 lead to emulate a deadline mechanism relying for its enforcement on the feedback action, the same idea can be used to emulate priorities for tasks that do not have deadlines. As such, one can define **for this kind of tasks only** a **real** – i.e. not integer nor quantized – priority range, say from zero (lowest) to one (highest), and obtain the corresponding tentative $\hat{\alpha}$ elements as:

$$\hat{\alpha_i} = \alpha_{min,i} + p_i(\alpha_{max,i} - \alpha_{min,i}), 0 \leq pi \leq 1,$$

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

Alpha calculus for event based tasks

This is the case of many services, think for example of the mouse driver. The idea is that, when awakened, the task gets a certain tentative α element, which then decays to zero at a given rate, i.e. as

$$\hat{\alpha_i}(t) = \hat{\alpha_i}(t) b_i^{-(t-t_{0,i})}$$

and is reset to the initial value ($\hat{\alpha_{i0}}$ in the following) when a new awakening event is triggered, typically via an interrupt, that simply has to reset $t_{0,i}$ to the current time index. Note that this is the only case in which $\hat{\alpha_i}$ can in general undergo variations over each round. If this is not acceptable for any reason, one could for example allot a "small" fixed CPU share to the task, and consider it blocked when not awakened. It was already shown that the proposed schedulers manage blockings correctly, so the only price to pay for such a choice is a potentially (slightly) increased overhead.

Once all the tentative components $\hat{\alpha_i}$ are available, α is simply obtained by rescaling them (uniformly, as "relative task importances" are already substantiated by the choices of the sections above about $\hat{\alpha_i}$ calculus) so that the sum be one. In the case of particularly critical periodic tasks a flag can be introduced to prevent their components from being rescaled. Configuring a system simulation consists essentially in setting

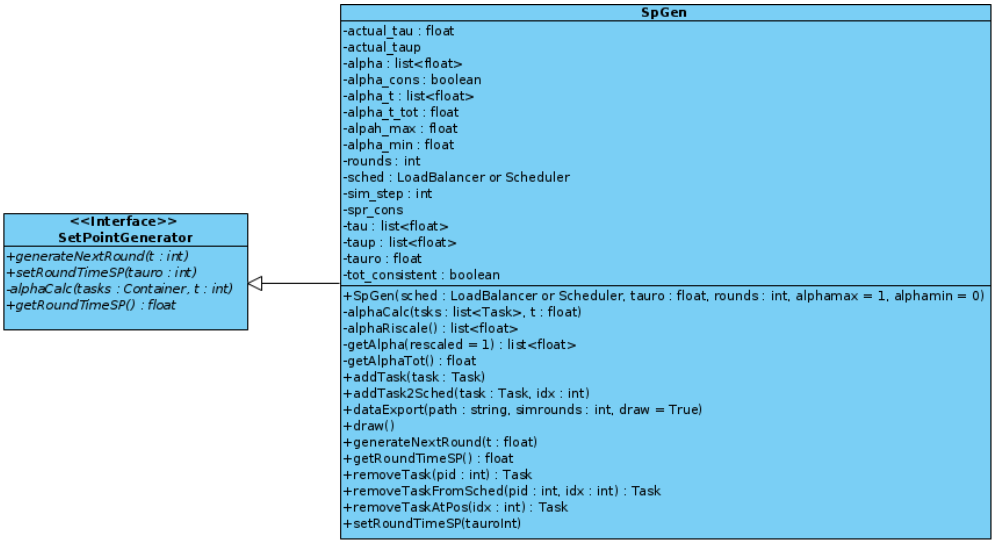

Figure 5.14.: UML class diagram of a simple implementation of a setpoint generator

1. What kind of architecture will be simulated (how many independent cores)
2. Set up initial task pools, setting limits on the admissible values for β_i, p_i and b_i , either system-wide or for some classes of tasks.
3. Set up TwoLevelSingleCoreScheduler parameterskSLC, kULC, zSLC for each single core scheduler initialized
4. Set up SetPointGenerator, deciding admissible values for parameters $\alpha_{min,i}, \alpha_{max,i}$ and how many tasks can be excluded from rescaling.
5. Decide which tasks manually migrate from a core to an other and at what point of simulation this should be done
6. Decide to plot simulation data or to export data to a file or both.

The interesting feature is that limits set on system behavior can be clearly interpreted in terms of the used control scheme.

Implementation of SpGen class, shown in figure 5.14, comes straightforward from the observations set out above.

Attributes:

- `actual_tau:float` it's an auxiliary field, to manage separately round duration of each different single core scheduler.
- `actual_taup:float` it's an auxiliary field, to manage separately previous round duration of each different single core scheduler.

- `alpha:list<float>` it's a list of rescaled alpha coefficients.
- `alpha_cons:boolean` it's a flag to state whether alpha rescaled vector has been calculated or not.
- `alpha_t:list<float>` it's a list of tentative alpha coefficients.
- `alpha_t_tot:float` it's the sum of all the elements of `alpha_t` vector; necessary to normalize it.
- `alpha_max:float` it's the maximum alpha value admissible.
- `alpha_min:float` it's the minimum alpha value admissible.
- `rounds:int` it's the number of simulation rounds to perform.
- `sched:LoadBalancer or Scheduler` it is the scheduling manager that directly deals with setpoint generator; it can be a LoadBalancer -in the case that simulation is referred to a multicore architecture- or a Scheduler, in the case that simulation is related to a single core system.
- `sim_step:int` it's a variable to keep trace of current simulation step.
- `spr_cons:boolean` it's a flag to state whether round time setpoint has been already set or not.
- `tau:list<float>` it's a list of current round durations (an element for each scheduler).
- `taup:list<float>` it's a list of previous round durations (an element for each scheduler).
- `tauro:float` it's round time set point.
- `tot_consistent:boolean` it's a flag to state whether `alpha_t_tot` field has been already calculated or not.

Methods:

- `SpGen(sched, tauro, rounds, alphamax = 1, alphamin = 0)` it's the constructor method: `sched` should be a Scheduler or LoadBalancer object reference; `tauro` is scheduling round duration setpoint `rounds` is the number of simulation rounds to perform. `alphamin` and `alphamax` are the respectively minimum and maximum value admissible for alpha coefficients. They can be set explicitly or safe default values can be used.
- `alphaCalc(taks:list<Task>,t:float)` private support method to calculate alpha coefficients. `t` is current simulation step, different from simulation time.

- `alphaRiscale()` it's a private support method to rescale alpha vector
- `getAlpha(rescaled = 1):list<float>` this method returns alpha vector; if `rescaled` is one, alpha vector is rescaled, else, it is returned without rescaling.
- `getAlphaTot():float` it's a private method: it returns the sum of the all alpha coefficients.
- `addTask(task:Task)` it adds a task to the underlaying Scheduler or LoadBalancer module.
- `addTask2Sched(task:Task, idx:int)` it adds task to the scheduler in position idx in underlaying LoadBalancer module.
- `dataExport(path:string, simrounds:int, draw = 1)` it is an auxiliary method to export simulation data to a file; `path` is the absolute path where data should be saved; `simrounds` is the number of simulation steps performed, `draw` is a flag to state whether data should be plotted or not. it is a wrapper of the methods exposed by MultiCoreLoadBalancer and TwoLevelSingleCoreScheduler.
- `draw()` it's an auxiliary method to plot simulation data. It is a wrapper of the method exposed by MultiCoreLoadBalancer and TwoLevelSingleCoreScheduler.
- `generateNextRound(t:float)` generates next simulation round; it calculates alpha vector and set it into `sched` data structure. Once alpha vector is set (rescaled or not), `calculateNextRound(tau)` method, exposed by LoadBalancer (or directly by TwoLevelSingleCoreScheduler, in case of single core system simulation) is called.
- `getRoundTimeSP():float` it is a getter method; it returns set point of round time duration.
- `removeTask(pid:int):Task` removes task from underlaying task pools (setting its alive flag to 0), searching by task pid (passed as first parameter). It returns a deep copy of removed task. If task is not found, a ValueError exception is propagated.
- `removeTaskFromSched(pid:int idx:int):Task` exposes underlaying Load Balancer interface: removes task identified by pid (setting its alive flag to 0) from the scheduler in position idx in load balancer's scheduler list. pid has to be a valid process identifier; idx has to be a valid list index; returns a deep copy of the removed task, if task is not found, a ValueError exception is propagated.
- `removeTaskAtPos(idx:int):Task` removes task at position idx (setting its alive flag to zero) in current scheduler's queue. it is compatible only with TwoLevelSingleCoreScheduler module; if module managed by SpGen object is

not a TwoLevelSingleCoreScheduler, NotImplementedError exception is raised. idx is an index, so it has to be a positive integer; it returns a deep copy of removed task.

- `setRoundTimeSP(tauro:int)` setter method: sets round time set point, so tauro has to be non negative, even if zero is a nonsense.

5.5. Testing and profiling

As described in previous sections, simulator is composed of a quite wide range of modules. The most of them needs parameters in input and returns results to the caller, so, exploiting python support modules as PyUnit tests, testing procedures have been automatized.
Explaining testing procedure in detail would be very long and not particularly interesting: Python is an interpreted language with dynamic typing and dynamic binding, so, parameters type are not controlled at compile time, and testing procedures have to take care to control parameter data type, parameter values, correct data management and data returned. Work has been quite long, and completely report it here would result in an excessive burdening of the thesis work, with little value added. For these reasons, only general guidelines about testing project will be given, explaining the concepts followed to realize a good code covering and how to eventually expand test cases in the future. In the whole implementation, parameters passing has been managed 'the pythonic way': python is able to dynamically add methods or attributes to single object instance and has a quite good and extended error management system, so, it has been exploited to manage parameter passing. When a parameter is used inside a method, it is surrounded by a try-catch block, managing only the exception necessary to keep system working in a correct state and propagating the others. [7]

This is a good way of managing parameter passing, for at least two reasons:

1. it is more efficient that using **isinstance()** function
2. Controlling runtime data types is anyway not sufficient to have absolute guarantees on the data correctness, because some instances could have methods added a run time that original class archetype did not have or some attributes type could be different from the one that we expected, so absolute guarantees about data types are almost impossible to obtain

Anyway, simulator layered structure has helped on organizing and automatizing testing procedures. All modules are equipped by test classes, having the following structure:
for each module, one ore more test modules are implemented: all modules have PyUnit test module; in figure 5.15 is reported TaskPool unit test module as example;

[7] In python community it is said that if an animal seems a duck and acts like a duck, then, probably is a duck.

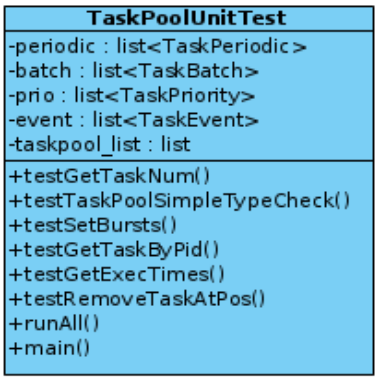

Figure 5.15.: UML class diagram of task pool unit test class

Attributes:

- `periodic:list<TaskPeriodic>` it is a list containing initialized PeriodicTask objects.
- `batch:list<TaskBatch>` it is a list containing initialized TaskBatch objects.
- `prio:list<TaskPriority>` it is a list containing initialized TaskPriority objects.
- `event:list<TaskEvent>` it is a list containing initialized TaskEvent objects.
- `taskpool_list:list` it is a list: for task pool module, it is built with this template:
  taskpool_list(idx,task_pool, Nperiodic, Nbatch, Nprio, Nev, Ntot), where idx is the list index, task_pool is a TaskPool object, Nperiodic is the number of periodic tasks contained by task_pool , Nbatch is the number of batch tasks contained by task_ and so on.

objects have parameters useful to check methods correctness; particularly useful are parameter values on the frontier of the sets of admissible values.

Methods:

- `testGetTaskNum()` checks, through assertions, every **taskpool_list** element:
  it checks that
  task_pool.getTaskNum(periodic) == Nperiodic,
  task_pool.getTaskNum(batch) == batch and so on;
  Ntot checks that task_pool.getTaskNumTot() == Ntot, testing the getTaskNumTot() method, which returns total number of tasks present in task pool.

- `testTaskPoolSimpleTypeCheck()` this method tests that constructor method **TaskPoolSimple()** works correctly; it tries to initialize different TaskPoolSimple objects with non admissible parameters values and verifies that the correct exception is thrown.

- `testSetBursts()` this method tests **setBursts(burtst:list <float >)** method. in the control schema, task's burst are affected by a white Gaussian noise. This is realized in simulator, adding a white Gaussian noise to bursts after they are set. For this reason, this method does not check that a correct list of bursts is correctly set into task pool data structure (effective burst values are not known a priori). However, it tests that parameters have the correct type and that list of bursts have the correct length and that every burst is effectively a positive number.

- `testGetTaskByPid()` this method tests that method **getTaskByPid(pid:int)** manages correctly parameter passed, throwing appropriate exceptions, in case parameters passed do not assume correct values. Testing that the returned task is effectively a deep copy of the Task object which is identified by the passed pid and that the list contains the original task with **alive** flag set to 0 is too complicated to do only through PyUnit test. To test this feature, functional test has been used, using test TaskPoolSimple objects and manually verifying that methods work correctly, inspecting TaskPoolSimple object state before an after invoking **getTaskByPid(pid:int)** method and inspecting Task object returned after method completion. This is realized in a stand alone test module with a new test class (TaskPoolTest class).

- `testGetExecTimes()` this method tests **getExecTimes()** method of TaskPoolSimple class; as for **setBursts()** method, due to the additive white Gaussian noise, it is not possible to confront single values set into data structure (set values are generally different from the ones which are returned by getter method) anyway, exploiting taskpool_list attribute, test method controls that length of execution times list is equal to Ntot field and that each element of the list is a positive number.

- `testRemoveTaskAtPos()` this method tests **removeTaskAtPos(idx:int)** method; as explained for **testGetTaskByPid()**, only correct parameter management is tested by assert methods (in particular we test that not admissible parameter values cause correct exception raising and that admissible values are accepted). Correct method behavior is tested by TaskPoolTest class, through functional testing, similar to the one described for **testGetTaskByPid()** method.

- `testRemoveTask()` this methods tests **removeTask(pid:int)** method. Strategy adopted is very similar to the one of previous test method (parameter management tested through PyUnitTest and correct method behavior controlled 'by hand' exploiting functional testing).

- `runAll()` this method launches all test methods implemented in the test class.

- `main()` main method simply calls **runAll()** method; if every test class has this structure, to test the whole simulator is sufficient to import all test classes and, for each class, to invoke its **main()** method [8]

Above example is related to TaskPoolSimple module, but the approach is general, and it has been used to test all MultioreSchedulerSimulator project. For each module, a test directory has been created, in this directory, PyUnitTest module and functional test module [9] are implemented. PyUnitTest module at least controls that constructor and all the methods exposed by tested class manages correctly parameter passed and return correct data. Test cases list allows to easily append new test cases to the one already present and to re-run all the test cases. Functional testing module controls (but not automatically, test team has to control results) that complex methods work correctly on the objects they modify. In this way is quite simple (but quite long, this is the reason why testing project is not entirely reported) to write tests that realize a good code coverage. In particular, it is easy to write new test cases to add to the one already present or re run entire tests after that one ore more methods have been modified.

5.5.1. Profiling

To understand which are the main bottle necks of simulator implementation, it has been profiled. NetBeans offers a Python profiler module and it has been exploited to carry out project profiling, finding out which are the main areas where future development works should focus on. In table above are reported the first ten heaviest (in term of total time execution) function calls:

as expected, the first is **main()** method of Main class. Simulation is carried out by this method, which is called once and then calls, for every simulation step, methods exposed by underlaying classes, so it is normal that is the function in which the most of execution time is spent. Second function is a built-in python function, used to manage lists (arrays in python are implemented as lists),since the most of simulator structures are implemented exploiting lists, this is an other expected result. Result at third, fourth and fifth position are more interesting: in fact it is pointed out that **calculateNextRound()** method of TwoLevelSingleCoreScheduler class is the heaviest, after **main()** method of Main class and that built-in function **isinstance(object,string)** is the heaviest after the one to manage lists. In fact, **isinstance()** function controls that runtime type of **object** is exactly the one identified by **string**. This function is one of the heaviest, because it has to inspect deeply object passed and all its attributes and methods. This explain also why the next most stressing method is **alphaCalc()** method;
calculateNextRound() method result heavier because TwoLevelSingleCoreScheduler manages also data structures used to plot and/or export simulation data. **isinstance()**

[8] NetBeans IDE does it automatically when 'test project' command is launched (but every test method name has to have 'test' prefix.
[9] in case that correct behavior testing it is too difficult to be realized only through assertions.

function is used anytime a module wants to be sure about runtime type of a given object, in particular, it is used a lot inside **alphaCalc()** method, to find out whether a given task is a periodic one, a batch one, a priority one etc, to calculate its alpha coefficient in the proper way. A possible solution to remove this bottle neck is to check only **type** field exposed by the task. It will increase efficiency, but it is less safe from software engineering point of view. Choice depends on how much simulation time is a critical aspect.

3618863 function calls (3592247 primitive calls) in 89.850 seconds
Ordered by: internal time, cumulative time

ncalls	tottime	percall	cumtime	percall	filename:(function)
1	75.167	75.167	76.634	76.634	{built-in method mainloop}
161010	2.022	0.000	2.022	0.000	{numpy.core.multiarray.array}
4000	0.786	0.000	2.533	0.001	TwoLevelSingleCoreScheduler:(calculateNextRound)
526081	0.563	0.000	0.563	0.000	isinstance
4000	0.503	0.000	1.177	0.000	SpGen:(__alphaCalc)
4000	0.431	0.000	0.880	0.000	TaskPoolSimple:(setBursts)
93486	0.419	0.000	1.747	0.000	fromnumeric:(size)
302/278	0.365	0.001	2.103	0.008	{built-in method call}
143928	0.357	0.000	1.906	0.000	numeric:asarray
20616	0.337	0.000	0.751	0.000	path:__init__

Table 5.1.: Table summary of profiling results, ordered by cumulative time

6. Conclusions and future developments

Overall, we can state that the results of the work described in this thesis are interesting and encouraging. The classification of tasks into classes according to their different behavior with respect to the system has allowed for the identification of high-level models that can simulate the behavior of multiple tasks of different types, sharing however some characteristic features (such as how they use the memory or interact with the user). Moreover, the devised models are linear, have a simple structure, and are characterized by a reduced number of parameters (ten in the case of the most complicated ones).

The presented simulator works satisfactorily, as it proved for example able to simulate a quad core architecture with more than thirty tasks in about ten seconds. Its architecture allows to easily implement new modules (most important, those that will implement tasks models for multicore environment) and to test different control policies, either by modifying the parameters of the control structures already implemented, or by implementing new structures: a module that takes into account thermal issues, a function that manages automatically task migration, and so forth.

For completeness, we have to mention that a module is still missing, that automatically handles the migration of tasks at high level, but the functions to handle the migration at low (core) level are already available, tested, and working: in particular, to summarize some relevant obtained outcomes in the context of the concluding remarks of this section, figure 6.1 reports simulation result related to core 4, in particular, detail of a batch task migration at simulation round 200.

Figure 6.2 shows what happens on core 3 which receives task migrated from core 4.A small remark on accumulated time of periodic task with period 73: this task has a quite high value of beta (0.7) and a short period, so, when new period starts, its accumulated CPU time is reset, but task is immediately rescheduled, refreshing accumulated CPU time (this is the reason why its accumulated CPU time does not reach zero value) and at the next step workload set point is reached.

In figure 6.3 is shown that scheduler manages in a good way also periodic tasks which are not harmonic. Moreover, around step 400 a new period task is received from an other core and properly scheduled.

6.1. Future work

This work was intended to set the foundations and the starting point for the application of a control-centric attitude to simplify the development of new solutions for the

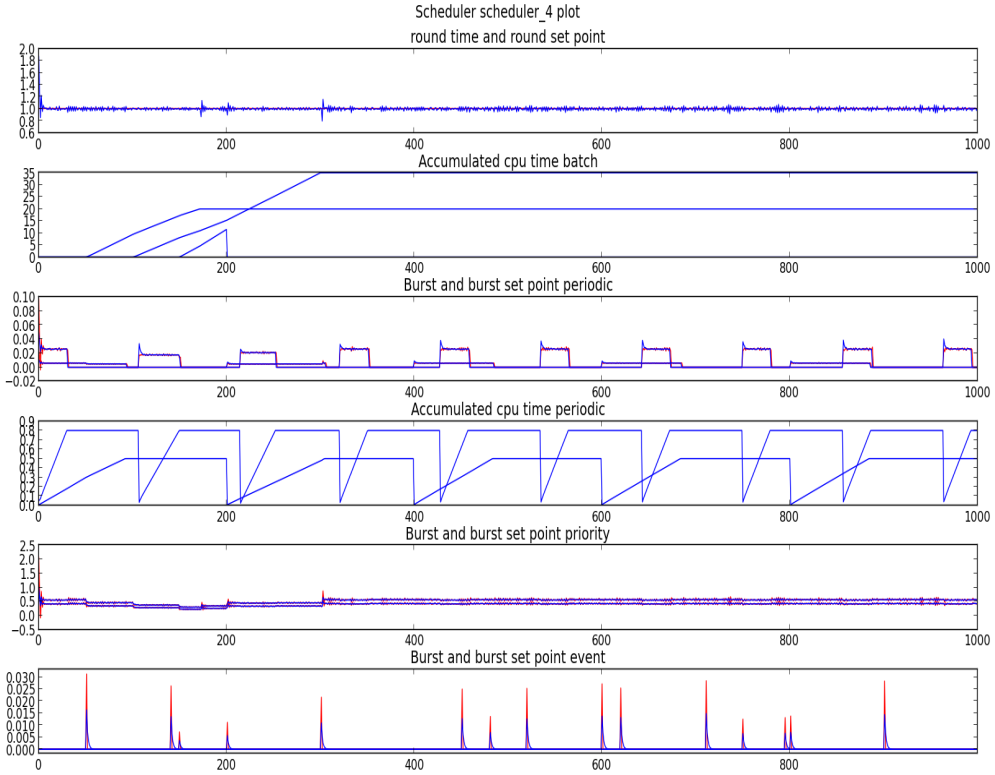

Figure 6.1.: Quad core scheduler simulation: in plots 1,3,5,6 the red trace is the task time slice set point, the blue one is the actual task burst; in the other plots, the blue trace is the accumulated CPU time. At time 200 a batch task is migrated, the detail is shown in second plot.

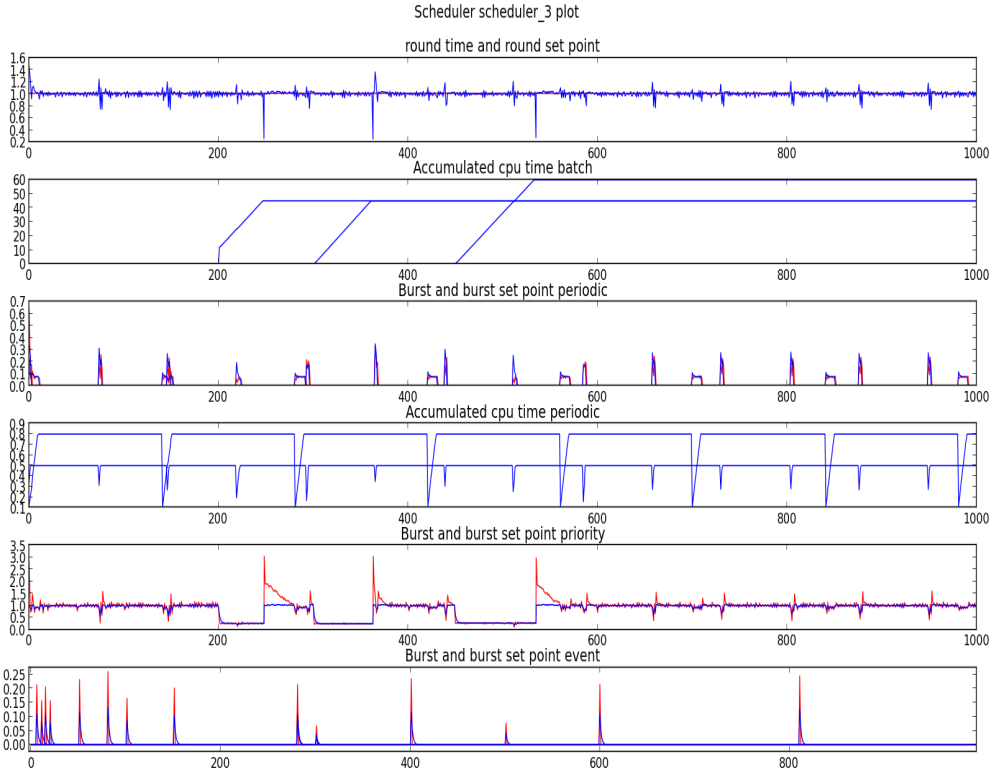

Figure 6.2.: Quad core scheduler simulation: in plots 1,3,5,6 the red trace is the task time slice set point, the blue one is the actual task burst; in the other plots, the blue trace is accumulated CPU time. At time 200 a batch task is received, the detail is shown in second plot.

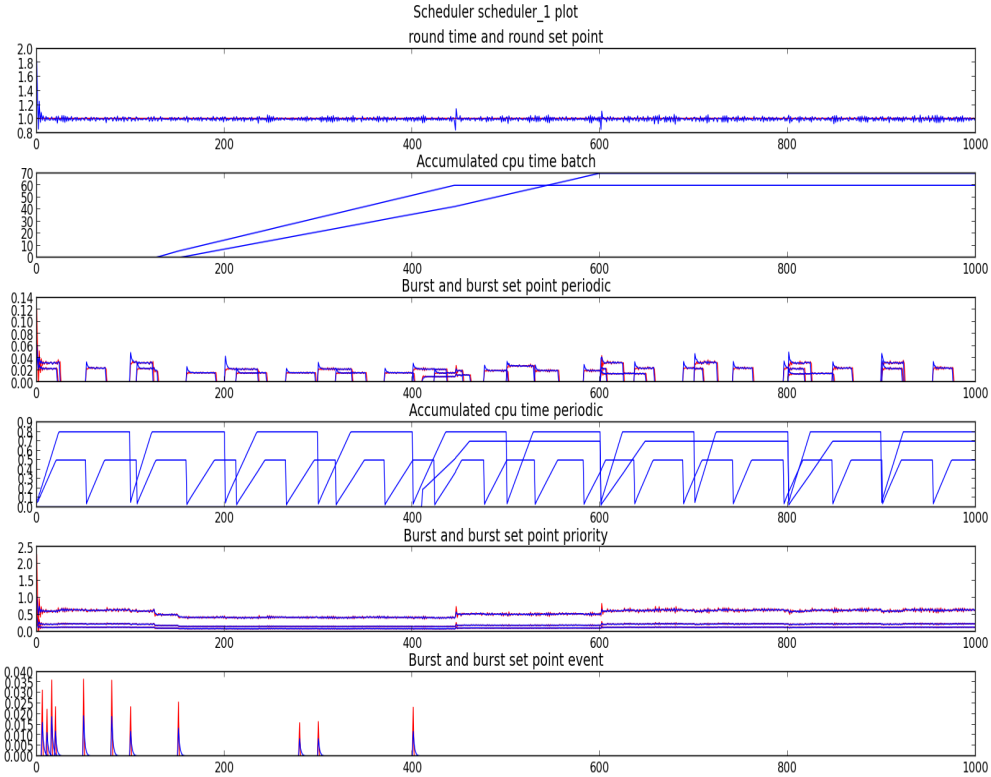

Figure 6.3.: Quad core scheduler simulation: in plots 1,3,5,6 the red trace is the task time slice set point, the blue one is the actual task burst; in the other plots, the blue trace is accumulated CPU time. Around time 400 a periodic task is received, the detail is shown in second plot.

project, deployment, tuning and testing of a multicore Scheduler, designed according to the principles of the systems and control theory: some interesting and desirable extensions could be, for example, the following ones.

- Develop a different approach to estimate models that have not been possible to identify with linear methods
- Project a control structure to manage load balancing exploiting task models for multi core context;
- Analyze closed loop system considering load balancing loop and single core scheduling loop; is it stable? Exists an optimal control policy?
- Implement multi core task models and load balancing control structure in simulator;
- Tune up a good load balancing policy using simulator;
- Include a thermal module to take into account thermal issues in task migration policy

Also, from the future work suggestions just sketched, it can be guessed that the simulator proposed is a flexible and easy to use framework to project and implement control systems suitable for resolving problems similar, although not perfectly identical to CPU time allocation. Probably, another interesting work could be the extension of simulator to manage problems which involves managing system resources (like for example memory), or network management.

Bibliography

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

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

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

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

[5] Ken W. Batcher and Robert A. Walker. Dynamic round-robin task scheduling to reduce cache misses for embedded systems. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 260–263, New York, NY, USA, 2008. ACM.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Jan. 2011.

[7] S. Bittanti and G. Picci. *Identification, adaptation, learning: the science of learning models from data*, volume 153. Springer Verlag, 1996.

[8] Sergio Bittanti. *Model Identification and Adaptive Systems*. Pitagora Editrice, Bologna,, 2004.

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

[10] Python community. Python official site, 2000. [Online].

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

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

[13] R. Leupers, J. Elste, and B. Landwehr. Generation of interpretive and compiled instruction set simulators. In *Design Automation Conference, 1999. Proceedings of the ASP-DAC '99. Asia and South Pacific*, pages 339–342 vol.1, 1999.

[14] A. Leva, M. Maggio, A.V. Papadopoulos, and F. Terraneo. *Control-based Operating System Design*. IET, London, UK, 2013.

[15] A. Leva, M. Maggio, and F. Terraneo. Performance analysis of operating systems schedulers realised as discrete-time controllers. In *IEEE International Conference on Control Applications (CCA)*, pages 745–750, 2012.

[16] Lennart Ljung. *System Identification: Theory for the User*. Prentice Hall, 2 edition, Jan. 1999.

[17] M. Maggio, F. Terraneo, A.V. Papadopoulos, and A. Leva. A PI-based control structure as an operating system scheduler. In *Proceedings IFAC Conference on Advances in PID Control*. IFAC, Mar. 2012.

[18] P. Van Overschee and B. DeMoor. *Subspace Identification of Linear Systems: Theory, Implementation, Applications*. Springer Verlag, 1996.

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

[20] Steven Rostedt, 2010. [trace-cmd man page].

[21] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. Online cache modeling for commodity multicore processors. In *PACT*, 2010.

[22] Jianwen Zhu and D.D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 298–302, 1999.