# Handling Timing Constraints Violations in Soft Real-Time Applications as Exceptions

Tommaso Cucinotta and Dario Faggioli

*Real-Time Systems Laboratory*
*Scuola Superiore Sant'Anna, Pisa (Italy)*
*Email:* {t.cucinotta, d.faggioli}@sssup.it

**Abstract**

In this paper, an exception-based programming paradigm is envisioned to deal with timing constraints violations occurring in soft real-time and multimedia applications written in the C language. In order to prove viability of the approach, a mechanism allowing to use such paradigm has been designed and implemented as an open-source library of C macros making use of the standard POSIX API (a few Linux-specific optimizations are also briefly discussed).

The proposed approach has been validated by modifying `mplayer`, one of the most widely used multimedia player for Linux, so as to use the introduced library. An extensive experimental evaluation has been made, both when running the player alone and when mixing it with a workload of other synthetic real-time applications. In the latter case, different scheduling policies have been used, including both standard priority-based ones as available on the mainline Linux, and an experimental deadline-based one available as a separate patch.

The shown results demonstrate how the exception-based paradigm is effective in improving the audio/video delay exhibited by the player achieving a superior performance and a dramatically better Quality of Experience as compared to the original heuristic frame-dropping mechanism of the player.

*Keywords:* soft real-time, multimedia, exception handling

## 1. Introduction

General Purpose Operating Systems (GOPSes) are being continuously enriched with more and more features for handier development of time-sensitive, soft real-time and multimedia applications. This would allow the development of applications with stringent timing requirements, provided the programmer were also given some mechanism for specifying these timing constraints within the application, and if necessary for dealing with their violation.

It is, in fact, becoming quite common to have, even in small embedded devices, a multiplicity of activities, running concurrently under the super visioning of an Operating System (OS). Moreover, some of the involved

tasks may fall into the category of "real-time applications", i.e., they must comply with precise timing behavior by which the output of the computation must be ready. Most of the time they are *soft* real-time applications, what distinguishes them from hard real-time ones, on a number of different points. First, the typical knowledge, by developers/designers, of the main timing parameters of the application, such as the execution time of a code segment, is somewhat limited. In fact, it is not worth to recur to precise worst-case analysis techniques, and there is a need for using commonly available hardware architectures (that are optimized for average-case performance, penalizing predictability) and compression technologies (which cause the execution times to heavily vary from job to job, depending on the actual application data). Furthermore, in order to scale down production costs, a good resources saturation level is needed. Finally, timing requirements in this context may be stringent, but they are definitely not safe-critical, therefore it may be sufficient to fulfil them with a high probability. Typical examples are multimedia players available, e.g., on modern smart-phones or set top boxes, and coexisting with wireless link management tasks or with planned recording of TV shows.

Therefore, timing constraints violations should be expected to occur at run-time, and developers must somehow cope with them. This paper presents a framework that enables the adoption of the well-known *exception-based management* programming paradigm to handle timing constraints violations in C applications, making it possible to deal with such events similarly to how exceptions are managed in languages like C++, Java or Ada. Specifically, two main forms of timing constraints can be specified: *deadline constraints*, i.e., a software component needs to complete within a certain (wall-clock) time, and *WCET constraints*, i.e., a software component needs to exhibit an execution time that is bounded.

Because of the space limitations, in this paper it is then assumed that the reader is familiar with the concept of exception, the possibility of it being raised during program execution either explicitly — i.e., throwing (with a function usually called `throw`) — or implicitly, because of some illegal operation such as non existing file, forbidden access to memory, etc. Moreover, the application programmer is generally asked to specify which are the code segments that may be affected by this phenomenon by enclosing them in a special block (e.g., `try ... `).

*Contribution of This Paper.* This paper presents and experimentally validates a mechanism that allows C programmers to take advantage of exception-based management of time constraints. To the best of the authors' knowledge, no similar mechanism has been previously presented for such programming language, with the same completeness, with no need to modify the C compiler, and only relying on standard POSIX features. A preliminary paper on the topic by the same authors has previously appeared [1], however in this work the proposed technique is validated experimentally by showing results gathered modifying a real existing multimedia application.

*Organization of the Paper.* After a brief overview of the related work in Section 2, Section 3 describes some possible utilization scenarios for the framework. Section 4 identifies the main technical requirements that need to be supported by the mechanism, and Section 5 describes the fundamentals characteristics of the library implementing such requirements. Section 6 illustrates the POSIX-based implementation realized for the Linux OS. Finally, Section 7 reports some performance evaluation measurements, highlighting the impact of the Linux kernel configuration on the mechanism precision, while Section 8 describes how the `mplayer`[1] application has been modified to utilize the framework and the experimental analysis conducted on it. Conclusions are drawn in Section 9 along with directions for future work.

## 2. Related Work

The need for having more and more predictable timing behavior of system components is well-known within the real-time community, to the point that modern general-purpose (GP) hardware architectures are deemed as inappropriate for dealing with applications with critical real-time constraints. In fact, there exist such approaches as Predictable Timed Architecture [2], a paradigm for designing hardware systems that provide a high degree of predictability of the software behavior. However, such approaches are appropriate for hard real-time applications, but cannot be applied to predictable computing in the domain of soft real-time systems running GP hardware. Yet, the concept of deadline exception has been actually inspired by the concept of deadline instruction as presented in [3].

Coming to software approaches relying on the services of the Operating System (OS) and standard libraries, the POSIX.1b standard [4] exhibits a set of real-time extensions that suffice to the enforcement of real-time constraints, as well as to the development of software components exhibiting a predictable timing behavior. However, working directly with these very basic building blocks is definitely non-trivial. The code for handling timing constraints violations, as well as other types of error conditions, needs to be intermixed with regular application code, making the development and maintenance of the code overly complex. As it will be more clear later, the proposed framework improves usability of these building blocks, by enabling the adoption of an exception-based management of these conditions.

Such an approach is not new, in fact it is used in other higher-level programming languages, such as Java, with the Real-Time Specification for Java (RTSJ) [5] extensions. These, beyond overcoming the traditional issue of the unpredictable interferences of the Garbage Collector with normal application code, also include a set of constructs and specialized exceptions in order to deal with timing constraints specification, enforcement and violation. Also, the Ada 2005 language [6] has a mechanism that is very similar to the one presented in this paper, namely the Asynchronous Transfer of Control (ATC), that allows for raising an exception in case of an absolute or relative deadline miss, and/or of a task WCET violation, that cause a jump to a recovery

[1] More information is available at: `http://www.mplayerhq.hu/`.

code segment. However, the focus of this paper is on the C language, probably still the most widely used language for embedded applications with high performance and scarce resource availability constraints. By making such a mechanism easily and safely available in C, the work presented in this paper contributes in enriching the language with an essential feature useful for the development of real-time systems. On a related note, Winroth [7] provided a set of macros for the C language supporting exception handling, but it did not address timing constraints violations and asynchronous transfer of control.

Focusing on the C language, the RTC approach proposed by Lee et al. [8] is very similar to the one that is introduced in this paper. They theorized and implemented a set of extensions to the C language allowing one to express typical real-time concurrency constraints at the language level, and deal with the possible run-time violations of them, and treat these events as exceptions. However, while RTC introduces new syntactic constructs into the C language, requiring a non-standard compiler, this paper presents a solution based on a set of well-designed macros that are C compliant and may be portable across a wide range of Operating Systems. Furthermore, RTC explicitly forbids nesting of timing constraints, while the approach presented in this paper does not suffer of such a limitation, allowing for an arbitrary number of nesting levels (up to a configurable maximum).

Finally, the concept of *time-scope* introduced in [9] is also similar to the "try_within" code block that is presented in this paper. However, that work is merely theoretic and language-independent, and it does not present any concrete implementation of the mechanism.

## 3. Possible Utilization Scenarios

In order to derive the requirements for the mechanism presented in this work, two typical use cases have been considered, as illustrated in the next section: (i) a component based video player, and (ii) an embedded control scenario making use of an "anytime algorithm".

### *3.1. Video Player*

Consider a video player, designed as a single thread of execution activated periodically or sporadically. A possible behavior for the *Video Decoder* component of such application is outlined in the UML Activity Diagram of Fig. 1.

From a run-time perspective, both audio samples and video frames must be decoded and played within precise timing, depending on the type of the media and on the format of the stream. It is well-known that, if data are not ready on time, it may be better to abort the operation on the current frame and start working on the next frame, since the user perception would not benefit from the reproduction of late samples[2]. Similar considerations can be made for the frame post-processing part, which might be skipped

[2]Whether or not it is better to abort the decoding as well, or to just skip the visualization of the late frames, is highly dependant on both the frame and the encoding algorithm.

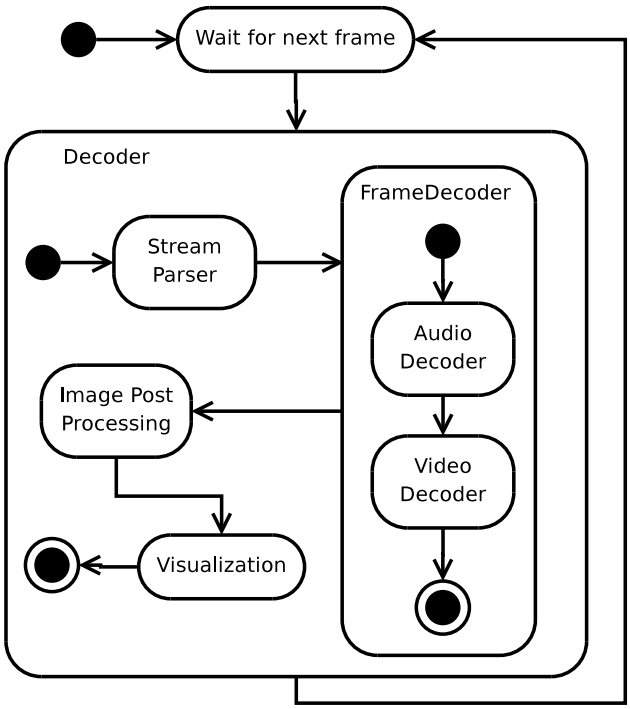



Figure 1: UML Activity diagram for the example video decoder thread.

if the decoder is lacking time. Therefore, the player described by Fig. 1 may be implemented as shown in the following code excerpt:

```
typedef ... enc_frame_t;        // An encoded/compressed frame
typedef ... dec_frame_t;        // A decoded frame
time_t dl;                      // Relative deadline for frame decoding

dec_frame_t FrameDecoder(enc_frame_t *f, *d) {
  AudioDecoder(f, d);
  VideoDecoder(f, d);
}

void Decoder(stream_t *s) {
  enc_frame_t *f;        // Encoded frame to be decoded
  dec_frame_t *d, d_old; // Decoded and previously decoded frame
  dec_frame_t *v;        // Frame to be displayed
  while (1) {
    f = StreamParser(s);
    dec_frame_t d = NewFrame();
    try(dl) {
      /* Aborted if still inside at time T */
      d = FrameDecoder(f);
```

```
      ImagePostProcessing(d);
      d_old = d;
      v = d_old;
    } catch (Exception)  {
      /* If aborted, re-use last decoded frame */
      v = d_old;
    }
    Visualization(v);
    WaitForNextFrame(s);
  }
}
```

On the other hand, from a designer perspective, it would be highly desirable to characterize each component with a WCET (or with some other appropriate statistic of execution time distribution). Also, it might be desirable that *Video Decoder* actually respects such WCET, even in cases of overload — e.g., when a frame is particularly difficult to decode. Due to the in-place timing requirements, it would be useful to characterize the *Frame Decoder* invocations that happen inside the *Video Decoder* with the WCET to be expected at run-time as well, since the sum of such value, plus the WCETs of the *Stream Parser*, *Filtering* and *Visualization* components, turns out to be the WCET of the *Video Decoder* itself. Moreover, video decoding architectures are highly modular, and make heavy use of third-party video and audio decoding plug-ins, e.g., depending on the stream format. Thus, in order to allow for an appropriate use of *Frame Decoder* within real-time applications, it would be highly desirable for libraries developers to have some mechanism for specifying a WCET estimation such that either: (1) the decoding operation terminates within the WCET limit, or (2) it is aborted.

### *3.2. Anytime Algorithms*

For what concerns anytime algorithms, they have been theorized in real-time systems from long time, for enhancing flexibility [10, 11]. Thus, whenever it is possible, the computation done at each activation is split in a mandatory part, that needs to be completed, and one or more optional parts, that may be executed if there is enough time. They are also utilized in embedded control, as in Quagli et al. [12], where such paradigm is applied for controlling the stability of an helicopter simulator.

If an accurate enough estimation of the duration of the mandatory and all the optional computation phases is available, and if a call `rmng_computation_time()`, capable of reading the time left for the current instance, is provided, then an anytime algorithm may be coded just checking, before entering each optional computation, if there will be enough time to complete it. However, if the optional parts exhibit fluctuations in their actual execution time, relying on a conservative estimate for `Di` may result in dropping them more often than strictly required. Thus, an alternative solution is to always attempt to execute the entire

computation, having the optional parts asynchronously interrupted by an exception if they are lasting longer than allowed, as it is shown below:

```
int D1, D2; /* Computation time of optional parts */
while (1) {
  StartComputation();
  res = MandatoryComputation();
  try(D1 + D2) { /* Aborted if execution exceeds */
    res = OptionalComputation1(res);
    res = OptionalComputation2(res);
  } catch {
    EndComputation(res);
  }
  WaitForNextPeriod();
}
```

Notice that in both cases the overall result comes from the "merge" of the intermediate results of the various computation phases (this is why the result of the $i$-th phase is passed as argument to the $(i+1)$-th one), and actually utilized only at the time of the `end_computation()` call. In fact, should any optional part be aborted by an asynchronous exception, the result of the computation should either completely include the last optional computation results or completely ignore them.

### *3.3. Some Shortcomings*

One of the main issue that comes to mind while envisioning such approach is that, in some cases, it may not be possible to asynchronously jump from an arbitrary position in the application code, to the exception handling logic. This implies the application should be designed so as to tolerate this kind of operation abortion, avoid possible memory leaks, and to properly cleanup any resources that might be associated with the aborting code segment.

Generally speaking, there may always be special code segments the asynchronous interruption of which should be avoided. For this reason, it is useful to have a mechanism to temporarily stop the notification of an exception and force the jump to the recovery logic for a group of statements. Obviously, the notification should reach the application anyway, and remain pending till the end of the "protected" code section. This way, the proposed approach can be used for detecting violation of a timing constraint even in these cases, and, moreover, the recovery logic can, in such cases, rely on the application data to be consistent, since the computation was not interrupted asynchronously. Obviously, should the application need to compensate for the accumulated delay, then it would be desirable to have a means provided by the framework that allows to retrieve how much such delay is.

## 4. Requirements Definition

From the above considerations, the following set of high-level requirements may be identified for the mechanism envisioned in this paper.

- Mandatory requirements:

  **Requirement 1.** *it should be possible to associate a deadline constraint to a code segment, either specifying relative or absolute time values;*

  **Requirement 2.** *it should be possible to associate a WCET constraint to a code segment;*

  **Requirement 3.** *when a timing constraint is violated, it should be possible to activate appropriate recovery logic that allows for a gracefully abort of the monitored code segment; also, it should be possible for the recovery code to either be associated to a generic timing constraint violation, or more specifically to a particular type of violation (deadline or WCET);*

  **Requirement 4.** *it should be possible to use the mechanism at the same time in multiple applications, as well as in multiple threads of the same application;*

  **Requirement 5.** *nesting of timing constraints should be allowed, at least up to a certain (configurable) nesting level;*

  **Requirement 6.** *if there are two (or more) nested timing constraints, a violation should be propagated in such a way that it is caught by the recovery logic associated with the code segment that caused it to occur;*

  **Requirement 7.** *it should be possible to cancel a timing constraint violation enforcement if the program flow runs out of the boundary of the associated code segment, e.g., when it ends normally or when another kind of exception requests abort of the code segment;*

  **Requirement 8.** *the latency between the occurrence of the timing constraint violation and the activation of the application recovery code should be known to the designer/developer, and it should be possibly negligible with respect to the task execution time;*

  **Requirement 9.** *the mechanism should allow the programmer to specify "protected" sections of a code segment that will never be interrupted by a timing constraint violation notification. Thus, the execution of recovery code for exceptions occurring while inside such a protected section would be delayed to the termination of the section.*

- Optional requirements:

  **Requirement 10.** *the mechanism could provide a method for monitoring the time remaining before the specified constraint violation occurs;*

  **Requirement 11.** *the mechanism could gather and report data about the latency between the timing constraint violation occurrence and the actual activation of the recovery logic;*

  **Requirement 12.** *the mechanism could provide a special operation mode gathering benchmarking data of the code segments, instead of enforcing their timing-constraints. This mode could be enabled at compile time, and used for tuning the actual parameters used as timing constraints for the various code segments;*

  **Requirement 13.** *the mechanism could be portable to as many Operating Systems as possible.*

## 5. Proposed approach

Here a mechanism complying with the requirements identified in Sec. 4 is presented, with a focus on the programming paradigm and syntax.

### *5.1. Exceptions for the C language*

The first step is to have the possibility of dealing with exceptions in a C program. This has been made possible through a generic framework distributed as a part of the open-source project Open Macro Library (OML) [3]. Providing details about both OML and OML support for exceptions is impossible here for space reasons; it is enough to say that the implementation is based on the standard `setjmp()/longjmp()` system calls, that it supports hierarchical arrangement of exceptions, that the user can define new exceptions with typical super-type/subtype relationships between them and that it is both process and thread safe. Some more details are also available in [13].

### *5.2. Timing Constraints Based Exceptions*

OML timing constraints related exceptions can be specified and handled by means of the following constructs (the `oml_` prefix is omitted here to improve readability):

- `try_within_abs` (`try_within_rel`): starts a `try` block with an absolute (relative) deadline constraint;
- `try_within_wcet`: starts `try` block with a maximum allowed execution time;

[3] More information at: `http://oml.sourceforge.net`

- `try_within_disable` and `try_within_enable`: suspend and re-enable, respectively, the notification of a timing exception. Notifications that reach the application after a `disable` are not lost, rather they are deferred until the next `enable`;
- `ex_timing_constraint_violation`: is the basic type for timing constraint violation exceptions; catching this will actually intercept any kind of timing constraint violation, without distinguishing between them;
- `ex_deadline_violation`: is what occurs when a `try_within_rel` (or `try_within_abs`) segment does not terminate within the specified time;
- `ex_wcet_violation`: is what occurs when a `try_within_wcet` segment executes for more than the specified time.

Below it is shown, again, how the decoder imagined in Fig. 1 can be implemented, this time using OML exceptions support. It is supposed that an estimation of the decoding time is known to be $12ms$, and that the presentation time ($pts$) of the next frame extracted from the stream can be used as the deadline for the decoding and the visualization of such frame.

```
typedef ... enc_frame_t;        // An encoded/compressed frame
typedef ... dec_frame_t;        // A decoded frame
time_t dl;                      // Relative deadline for frame decoding

#include <oml_exceptions.h>

// If cannot meet the 12ms WCET, then raise exception
dec_frame_t * FrameDecoder(enc_frame_t f) {
  dec_frame_t *d = NewFrame();
  try_within_wcet(12000) {
    AudioDecoder(f, d);
    VideoDecoder(f, d);
  }
  handle
    when (ex_wcet_violation e) {
      ReleaseFrame(d);
      raise(e);
    }
  end;

  return d;
}

void Decoder(stream_t *s) {
```

```
  enc_frame_t *f;        // Encoded frame to be decoded
  dec_frame_t *d, d_old; // Decoded and previously decoded frame
  dec_frame_t *v;        // Frame to be displayed

  while(1) {
    f = StreamParser(s);
    d = NewFrame();
    try_within_rel(f->next_frame_pts) {
      d = FrameDecoder(f);
      ImagePostProcessing(d);
      v = d;
    }
    handle
      when (ex_deadline_violation) {
        v = d_old;
      }
    end;
    Visualization(d);
    ReleaseFrame(d);
    d_old = d;
    WaitForNextFrame(s);
  }
}
```

As a final remark, the example code below shows a typical usage of the disable/enable mechanism to protect code segments that must not be interrupted asynchronously. Both the memory allocation for the new object and its construction (which in turn may involve further allocation of memory segments, and/or other OS resources) are made atomic with respect to deadline exceptions. Also, destruction of the object occurs in the `finally` statement, what ensures it always happens, even if an exception is raised within the application body, which is not caught by the `oml_when` clause immediately following.

```
...
struct my_object *p_obj = NULL;
try_within_abs(next_dl) {
  /* safely interruptible computations */
  ...
  try_within_disable();
  p_obj = malloc(sizeof(struct my_object));
  if (p_obj == NULL)
    throw(ENoMemoryException);
  my_object_init(p_obj); /* Constructor */
  try_within_enable();
  /* safely interruptible computations */
```

```
...
} finally {
  /* Free allocated resources */
  if (p_obj != NULL) {
    my_object_cleanup(p_obj); / * Destructor */
    free(p_obj);
    p_obj = NULL;
  }
}
handle
  when (ex_deadline_violation) {
      /* Recovery logic */
  }
end;
```

OML Exceptions complies with all of the requirements introduced in Sec. 4, with the few notes outlined in the following sections.

## 6. Implementation

This section provides an overview of how the proposed mechanism has been implemented, always bearing the outlined requirements in mind.

### *6.1. Time-Scoped Segment Implementation*

OML Exceptions has been realized by means of the POSIX `sigsetjmp()` and `siglongjmp()`[4] functions. The former saves the execution context such that the latter is able to restore it, and continue program execution from that point.

For the `try_within_abs` and `try_within_rel` constructs, the time reference is the POSIX `CLOCK_MONOTONIC` clock. For the `try_within_wcet` macro, the time reference is the POSIX `CLOCK_THREAD_CPUTIME_ID` clock. In fact, while `CLOCK_MONOTONIC` provides an absolute time reference, useful for deadline constraints, `CPUTIME_ID`s clocks measure the actual execution time of a specific thread, which is exactly what is needed for WCET constraints. Events are posted using interval timers (POSIX `itimer`).

Notification of asynchronous constraint violations is done by delivering to the faulting thread a real-time signal i.e., a signal that is queued and guaranteed not to be lost. The OML Exceptions signal handler performs a `siglongjmp` to the appropriate context, jumping to the `handle...handle_end` block for the check of the exception type. This implementation is portable to any Operating System providing support for POSIX real-time extensions.

[4] `http://www.opengroup.org/onlinepubs/007908799/xsh/siglongjmp.html`

### *6.2. Deadline and WCET Signal Handling*

Every time a constraint is violated, the signal has to be sent to the *correct* thread (Requirement 4). However, signal delivery to a specific thread is not covered by POSIX, according to which, signals can only be directed to entire processes. What the standard suggests is to have a special thread sensible to the signal(s), with all the others ignoring it (them), and to use it to perform the intra-process part of the notification. However, such an approach would imply that every time a timing constraint is violated, the CPU incurs additional context switches, not to mention the additional overheads of managing (creating and destroying) the "signal router" thread.

On the other hand, Linux supports delivery of signals to specific threads thanks to an extension of the POSIX semantics built into the kernel. Therefore, on Linux platforms, a much more efficient implementation is possible by using this feature. A POSIX compliant version of the library is available as well, and it is possible to choose which one to use at library compile-time.

### *6.3. Non-Interruptible Code Sections*

The two macros, `oml_within_disable` and `oml_within_enable` make it possible to fulfil Requirement 9 about atomic code segments. They simply disable and enable (respectively) delivery of the time constraint violation signals. If a signal occurs in the middle of such a protected code region, then it is enqueued by the OS, and notified immediately at the end of the section.

### *6.4. Gathering Timing Information*

Optional Requirements 10 and 11 are fulfilled by the availability of the `try_within_rmng_time` and `try_within_expr_delay` macro. They are implemented by querying the timer that is being utilized for the enforcing of the `try_within` block for the amount of time remaining or passed to/from the expiration instant, respectively.

### *6.5. Benchmarking Operational Mode*

Coping with Requirement 12 happens by means of a compile-time switch that, when enabled, gathers information on the duration of all the `try...handle` code segments. This allows developers to easily obtain statistics about execution times of the time-scoped sections.

### *6.6. Precision Limitations and Latency Issues*

With respect to the maximum precision with which timing constraints are checked and enforced, this is limited by the time-keeping precision of the underlying OS.

For example, on Linux, from version 2.6.21, the kernel has been enriched with high resolution timers. Thanks to them, timers are no longer coupled with the periodic system tick, and thus they can achieve as

high resolution as permitted by the hardware platform. Nowadays, large number of microprocessors, either designed for general purpose or embedded systems, are provided with precise timer hardware that the OS can exploit, e.g., the TSC cycle counter register of the CPU[5] or the Intel HPET [14]. Therefore, since this is how timers based on `CLOCK_MONOTONIC` are implemented, a Linux task requesting a deadline exception to occur at a certain instant, could expect to be notified about such event quite close to that point in time.

On the other hand, per-thread CPU-time clocks are still based on the standard process accounting, which basically means their resolution depends on the OS periodic tick frequency, which typically is 100, 250 or 1000 Hz. Thus, the notification latency of a WCET violation will be dependant on how the kernel has been configured, with a 1000 Hz tick frequency being the best choice.

## 7. Performance Evaluation

The performance evaluation for the framework is done as prescribed by Requirement 8. Experiments have been run to measure the *notification latency*, i.e., the time interval between the actual constraint violation and the instant the proper thread in the application is notified about it. In fact — even if it is not strictly necessary to achieve an exact worst case case upper bound (the framework is mainly aimed at soft real-time systems) — the developer must know that the latency is small enough, if compared to the execution times and deadlines of the involved code segments, or the whole mechanism would become useless. Figure 2 shows the main components that may contribute to the violation notification latency for both the POSIX compliant and the Linux specific implementations. It is also of paramount importance to investigate what happens to the notification latency under diverse system load conditions, when the thread hosting the `try...handle` segment (the *constrained thread*) is scheduled by means of the various available policies and with `try...handle` segments of different durations. In fact, if there are any dependencies between these factors and the performance/accuracy of the framework, the programmer must be aware of them.

All the measurements have been done using a standard distribution of the Debian GNU/Linux OS (kernel version 2.6.37) running on a commonly available desktop PC, with 2.40 GHz Intel CPU and 4 GB of RAM. The Linux kernel configuration was hand-tailored so to ensure it included high-resolution timers and the support for high precision hardware timing sources. The simple test application used for latency evaluation was composed by just one thread, while the system was subject to varying load conditions. The scheduling policy used for the constrained thread was varied among `other`, `fifo` and `cbs` (detailed in the next subsection), while the background load was always run at `other`.

[5] `http://www.intel.com/Assets/PDF/manual/253668.pdf`

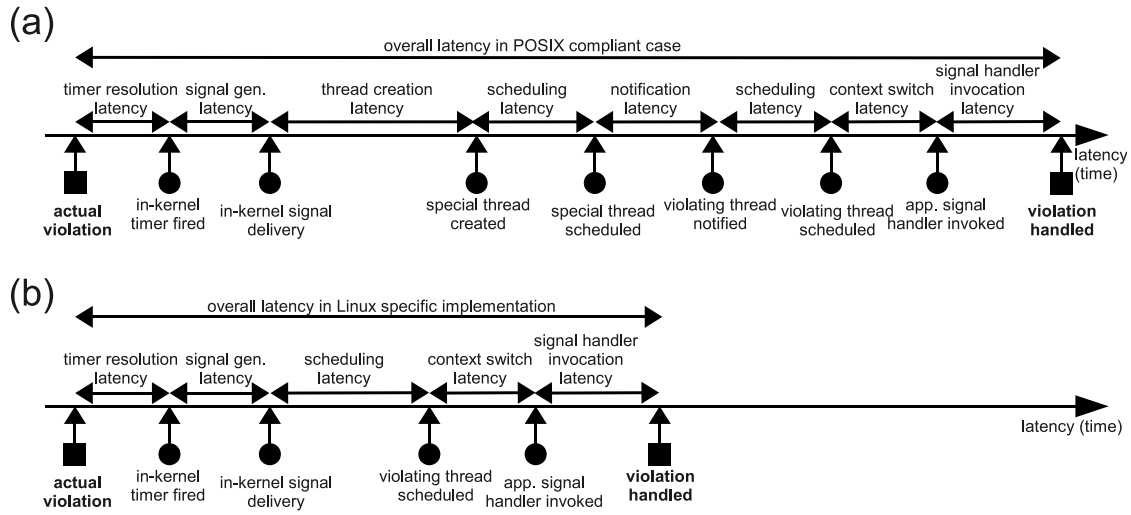



Figure 2: Main contribution to the notification latency of a timing constraint in the fully POSIX compliant case, inset 2(a), and in the Linux specific case, inset 2(b).

### *7.1. Scheduling policies in Linux*

The Linux kernel currently allows for applying to user threads up to two scheduling classes, **fair** and **real-time**, resulting in 3 possible scheduling policies, `SCHED_NORMAL`, `SCHED_FIFO` and `SCHED_RR`[6]. The fair scheduler aims at guaranteeing smooth progress to all the activities and yet retaining good interactivity and decent overall throughput, striving to fit the needs of an heterogeneous set of applications. This scheduling policy is known as `SCHED_NORMAL` (also called `SCHED_OTHER`) and it is referred to as `other` in the remainder of the paper. It achieves its goals by prioritizing the various threads according to their execution time and wakeup patterns. This is quite effective in providing fairness, but it is not a viable solution for enforcing precise timing guarantees for any of the applications using this policy.

Real-time scheduling may be realised by means of static priorities (assigned to a thread once for all at its start) or using directly the thread's deadline (renewed at each activation to the activation time plus the relative deadline or period) handled by the Constant Bandwidth Server (CBS) algorithm [15] (or a variation of that). Linux supports fixed priority real-time scheduling conforming to the POSIX standard [4] through the `SCHED_FIFO` and `SCHED_RR` policies (`fifo` and `rr` here). Threads using these policies have a priority assigned to them and are executed in strict priority ordering. The ones with the same priority run with the same order in which they arrive (FIFO) and, in case of `SCHED_RR`, they have a fixed timeslice of $100msec$.

On the other hand, if deadlines are to be used (for instance, using EDF), resource reservation can be enforced efficiently. The basic idea is that each task is reserved an amount $Q$ of CPU time (*budget*) every $T$

[6] `SCHED_IDLE` and `SCHED_BATCH` are also possible but out of the scope of this paper.

time units (*period*), e.g., by means of the CBS, which supports both periodic and aperiodic and dynamically changing tasks. Note that this is important to this paper, since the two `rt-app`s are typical real-time periodic tasks, while `mplayer`(`-dlex`) does not directly follow this model of behaviour. It then enables *temporal isolation*, meaning that the temporal behavior of an application is not affected by the behavior of the other applications in the system, since a thread which tries to consume more than the budget of its CBS is slowed down, and this will not hurt the time guarantees of the other CBSs. In Linux, such behavior is implemented by a new scheduling policy called `SCHED_DEADLINE` [7] (`cbs` in the remainder of this paper) that is being proposed for mainline integration [16].

### *7.2. Deadline Violation Latency*

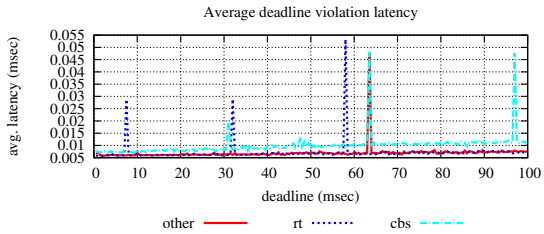
For benchmarking deadline violation latencies a simple single-threaded application is used. It executes a `oml_try_within_rel(deadline){}` block until `deadline` passes and the exception triggers. The value of `deadline` has been varied from $500\mu sec$ to $100msec$, in steps of $500\mu sec$ and, for each of these values, 10 samples of deadline violation have been collected. For measuring such latencies, the `oml_try_within_expr_latency()` construct, directly provided by the framework, was the preferred choice, in order to gather the same data a developer will be able to get while using the library. As it can be imagined, this facility reports how much later than the ideal notification instant — which in this case would be exactly `deadline` — the notification is being handled by the deadline miss signal handler inside OML. The testing program is distributed along with OML, and can be found in its source form inside the framework package. When running as `other`, the thread's priority (i.e., UNIX nice level) has not been modified from the Linux default; while `rt`, the POSIX real-time priority was set to 50 and while `cbs` it run in a reservation of $(budget, period) = (25\mu sec, 50\mu sec)$. For each of these scheduling policies, the program has been run both alone and with 4 different kind of background load, always running as `other`. More specifically, experiments with 0, 2, 5 and 10 load tasks were conducted, by means of the commonly available `stress` program. Disturbance was selected to affect not only CPU but also memory, striving to provide a more realistic benchmarking scenario, as detailed in Tab. 1.

| **load tasks** | **CPU intensive tasks** | **Memory intensive tasks** |
|---|---|---|
| 0 | - | - |
| 2 | 1 | 1 |
| 5 | 4 | 1 |
| 10 | 8 | 2 |

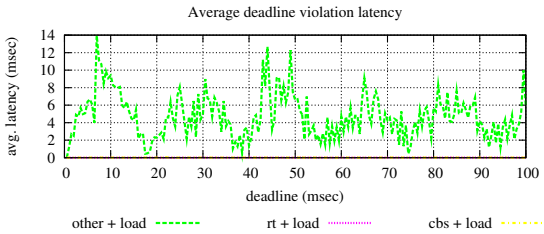
Table 1: Load conditions to which the various experiments were subject to.

Results are shown in the following Figures: 3, 4 and 5.

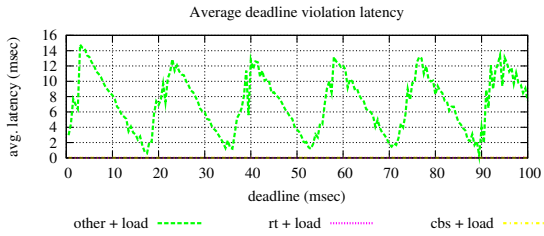
[7] More info at: `http://gitorious.org/sched_deadline`



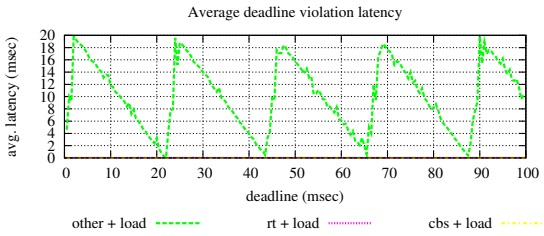

(a) 1 constrained task, 0 load tasks



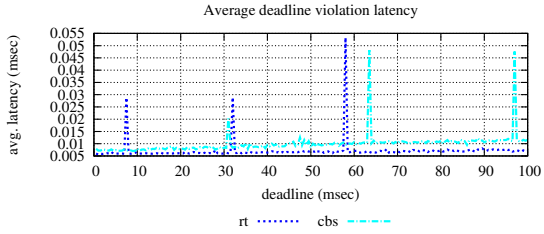

(b) 1 constrained task, 2 load tasks



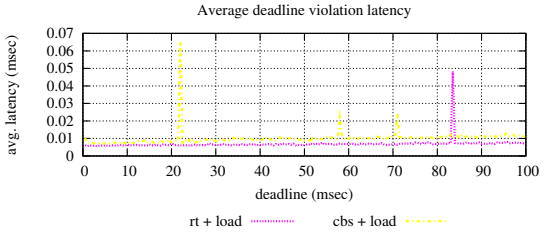

(c) 1 constrained task, 5 load tasks



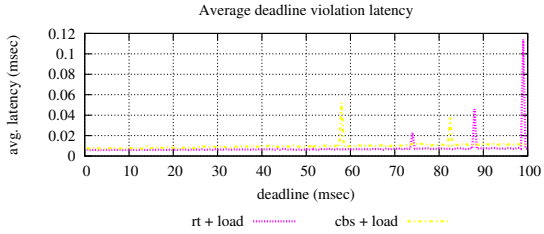

(d) 1 constrained task, 10 load tasks



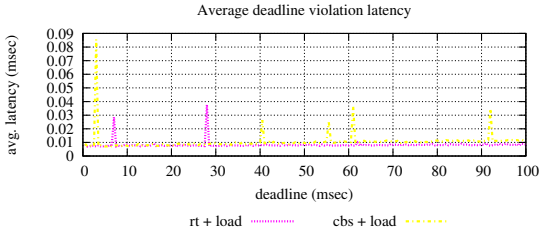

(e) 1 constrained task, 0 load tasks




(f) 1 constrained task, 2 load tasks




(g) 1 constrained task, 5 load tasks




(h) 1 constrained task, 10 load tasks

Figure 3: 3(a) — 3(d): Measured latency (on the y-axes) of the notification of a deadline constraint violation with various deadlines (on x-axes), with increasing (from (a) to (d)) background load run with the `other` policy.

Figures 3 and 4 show the average and the standard deviation of the deadline violation latency in the 10 instances of each experiment, at the different deadline constrained `try...handle` durations, for all the scheduling policies and with and without load. It appears quite clear how such latency depends more on the system load than on the duration of the constrained segment itself, provided the load and the constrained thread have the same scheduling policy. 3(e) — 3(h): as above, but without the curve about the constrained

thread and the load both at `other` policy, to highlight the behavior of the other cases.

All the graphs are reported twice, with (in 3(a) — 3(d) and 4(a) — 4(d)) and without (in 3(e) — 3(h) and 4(e) — 4(h)) the curve representative of the case when the constrained thread is run at `other` policy and the disturbing load is on. This is because such case consistently resulted in considerably different behavior, with respect to all the others, to the point that if `other+load` is shown, the curves for the remaining cases are barely visible.

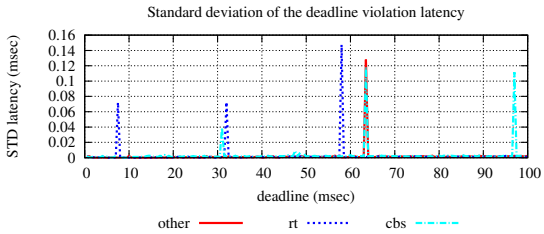
Looking at all these graphs it emerges quite clearly that the effectiveness of the framework in notifying a thread about a deadline violation highly depends on the system load and on which scheduling policy is being used. In fact, in case the constrained thread is not "protected" from external disturbances of other applications, performance easily drops down to unacceptable level. Notice that the curves for `other` in Figs. 3(b), 3(c) and 3(d) show some sort of a periodic behavior, with the "period of repetition" increasing with the number of disturbing load threads. This is a direct effect of the scheduler, since trying to be fair in distributing the processor time with an increasing number of intensive threads results in each of them being able to use less CPU. Thus, the constrained thread has to wait more before being scheduled again and noticing that a deadline miss notification is pending. It is impressive to see how the notification for a `try...handle` segment with a deadline of less than $5msec$ may reach the thread with a delay of $\sim 15$, $\sim 20$ or $\sim 32$ msec, making the mechanism fairly useless in these conditions. The highly fluctuating curves of the standard deviations (4(b), 4(c), 4(d)) and the long tails of the cumulative distribution functions (5(b), 5(c), 5(d)) of the same cases just reinforce this argument.

On the other hand, whether there is no significant load in the system or the constrained thread is given priority on such load (by means of any scheduling policy that achieves so), results in a reasonably low (Fig. 3) and stable (Fig. 4, 5) notification latency, even in cases of `try...handle` blocks no longer than a few milliseconds.

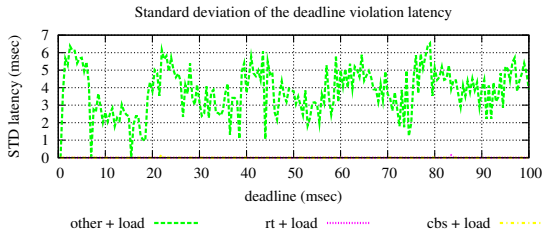
Finally, something that all the graphs agree in testifying is that there is no dependency between the latency of a deadline violation notification and the deadline itself, i.e., the duration of the deadline constrained `try...handle` segment. This basically means that trying to enforce a deadline of $1msec$ or one of $0.1sec$ would work equally effectively, i.e., it will happen with a delay of less than $0.01msec$ on average, provided the interference of external load is properly taken care of (e.g., boosting the scheduling policy). This is mainly due to the facts that the high resolution timer used by the `oml_try_within_rel()` construct is capable of being far more precise than $1msec$, which is what Requirement 8 is asking.

### *7.3. WCET Violation Latency*

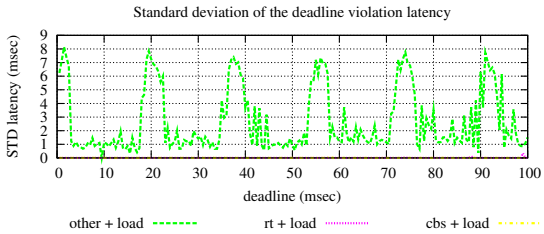
Benchmarking the latency for WCET violations happens with the very same method described above for deadline violations, i.e., using a `try_within_wcet(wcet)` {} block which always overruns, and repeating this for all the values of `wcet`, scheduling policies and load conditions already described in the previous



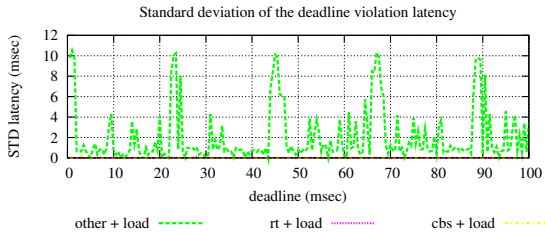

(a) 1 constrained task, 0 load tasks



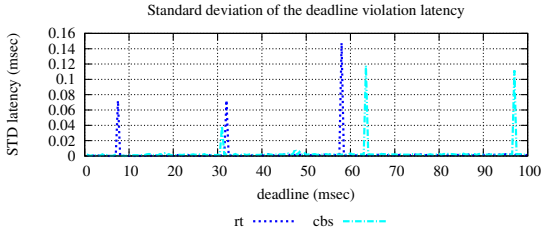

(b) 1 constrained task, 2 load tasks



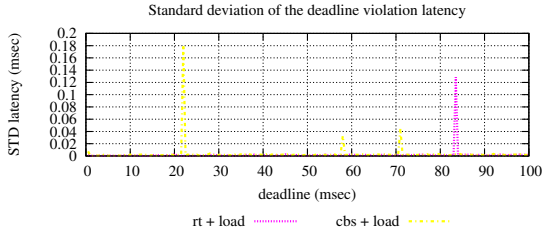

(c) 1 constrained task, 5 load tasks



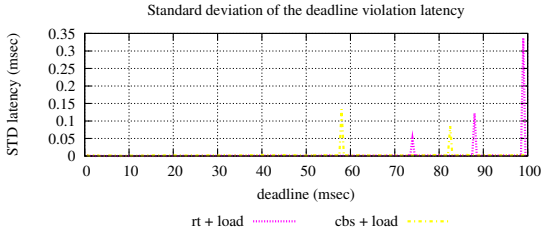

(d) 1 constrained task, 10 load tasks



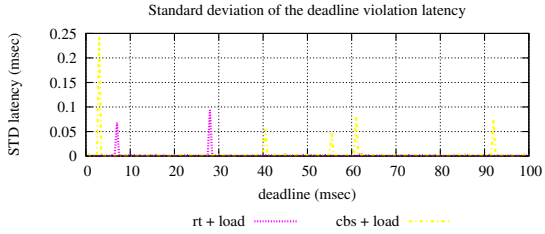

(e) 1 constrained task, 0 load tasks




(f) 1 constrained task, 2 load tasks



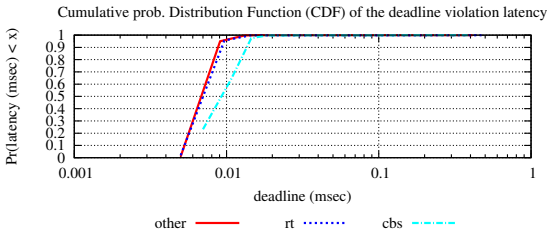

(g) 1 constrained task, 5 load tasks



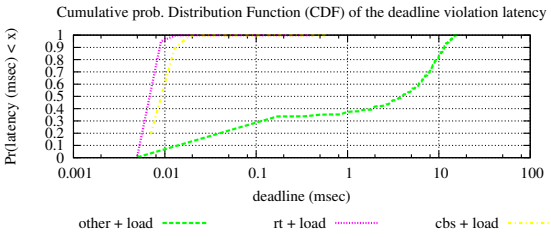

(h) 1 constrained task, 10 load tasks

Figure 4: 4(a) — 4(d): standard deviations of all the cases in Fig. 3. 4(e) — 4(h): as above, but without the curve about the constrained thread and the load both at `other` policy, to highlight the behavior of the other cases.

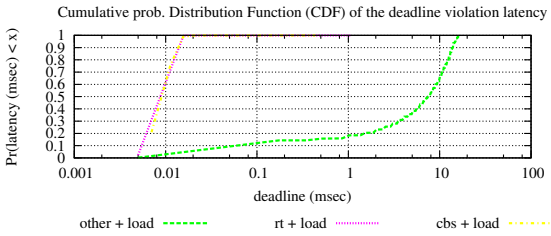
section. Note that in this case the `oml_try_within_expr_latency()` construct reports for how longer than `wcet` the thread executed before the violation handler gets to run. Moreover, the situation is quite different with respect to deadline miss notifications. In fact, deadlines are checked against a Linux high resolution timer backed by the system wide monotonically increasing `CLOCK_MONOTONIC` time base. These two factors make its resolution not only independent from all the configuration parameters of the experiment (as shown



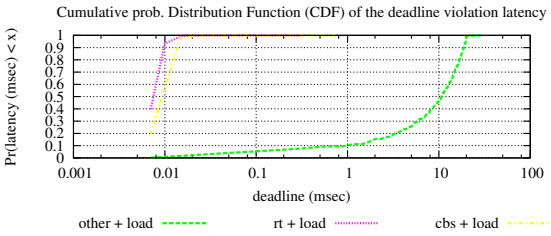

(a) 1 constrained task, 0 load tasks



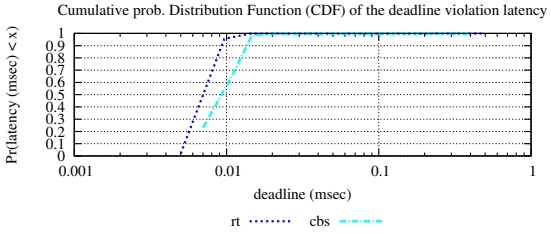

(b) 1 constrained task, 2 load tasks



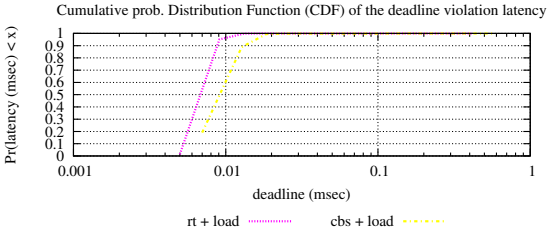

(c) 1 constrained task, 5 load tasks



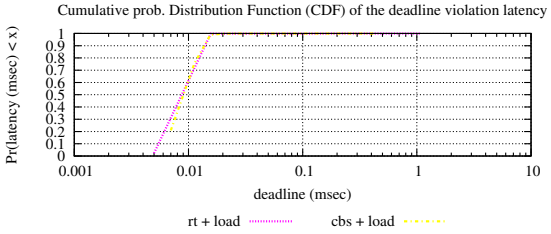

(d) 1 constrained task, 10 load tasks



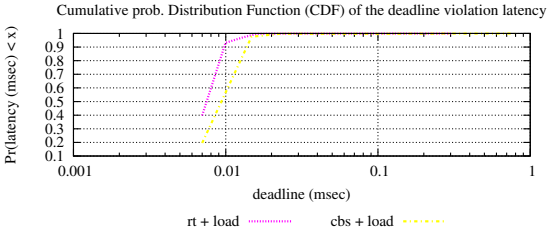

(e) 1 constrained task, 0 load tasks




(f) 1 constrained task, 2 load tasks




(g) 1 constrained task, 5 load tasks



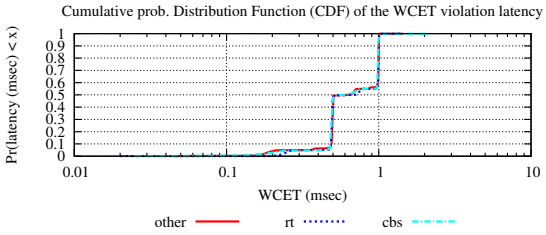

(h) 1 constrained task, 10 load tasks

Figure 5: 5(a) — 5(d): Cumulative Distribution Function (CDF) of all the notifications of a deadline violation of the all the various experiments. 5(e) — 5(h): as above, but without the curve about the constrained task and the load both at `other` policy, to highlight the behavior of the other cases.

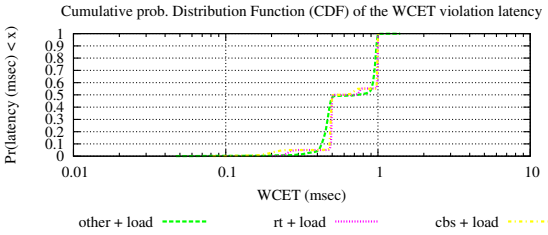
by the results in the previous section), but also from the periodic OS tick, since the chosen timer go off right at the instant it is programmed. On the other hand, `CLOCK_THREAD_CPUTIME_ID` time bases, used for this particular `try...handle` block, are updated on the OS tick boundary, which in this case has been configured to happen every $\texttt{HZ} = 1msec$. For this reason, plotting the average notification latency would result in the

very same curves, all superimposing onto the others, for all the cases of 0, 2, 5 and 10 disturbance load threads, and for all the various scheduling policies. Since this makes the plots impossible to understand, they are not showed here. This happens because each WCET constrained thread is subject, whether or not the system is under any kind of load, to a notification latency for such constrained segment of, on average, $\frac{HZ}{2}$.

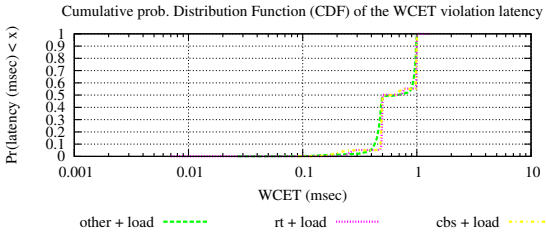
Therefore, focus on Fig. 6, depicting the CDF of the notification latency in the usual cases. All the curves in all the graphs exhibit pretty much the same behavior, highlighting that, as expected, a WCET violation happens almost all the times with a latency which is multiple of $\frac{\texttt{HZ}}{2}$, i.e., at $0.5msec$ and $1msec$ in this case. Therefore, in case of WCET violations, a tick aligned execution time constrained segment is most probably going to be notified with as small as possible latency, while the others may have to wait some time. More precisely, whatever the length of the segment, a latency not greater than `HZ` should be expected. Figures 6(b), 6(c) and 6(d) confirm this, showing that even in case some external load exists against which the constrained thread is not protected, this can make the curve less steep, but it does not affect its own inherent shape.



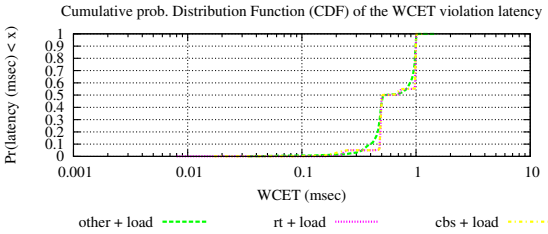

(a) 1 constrained task, 0 load tasks




(b) 1 constrained task, 2 load tasks




(c) 1 constrained task, 5 load tasks




(d) 1 constrained task, 10 load tasks

Figure 6: Cumulative Distribution Function (CDF) of all the notifications of a WCET violation of the all the various experiments.

Note that, there has also been no need of showing multiple copies of the graphs, since `other` and `other+load` behaves similarly to the other cases, and thus the curves are easily visible all together.

## 8. A Case Study: The `mplayer`

MPlayer is a free and open-source video streaming software and the starting of its development dates back to 2000. Born to run only on Linux it is now a cross-platform application, capable of running also on almost all other Unix-like systems, on Microsoft Windows, on Mac OS and on many other (minor) operating systems.

In order to provide the user with the best possible playback experience, `mplayer` utilizes the Audio/Video delay (*A/V delay*) as the prominent metric of its own performance. This means that, if the time passing between the playback of an audio sample and the showing on the screen of the frame associated with it starts growing too large, the program tries to do something in order to avoid loosing the synchronization among audio and video, possibly before this start being noticed by the user. The action that is undertaken in such cases is to *drop* one or more video frames. Dropping a frame means the decoder is asked to avoid processing it, unless it is a key or a reference frame (missing reference frame would cause serious artifacts in the video playback and this can not be tolerated). For the sake of completeness, note that a frame that has been decoded but is not visualized, still for timing reasons, it is said to be dropped as well. For example, when dealing with an MPEG stream, all the frames can be dropped — in the sense that they may not be visualized on the screen — but skipping the decode is only allowed for B type frames[8].

The `mplayer` modified to use OML Exceptions, referred to as `mplayer-dlex` in the remainder of the paper, can be downloaded from `http://gitorious.org/mplayer-dlexceptions`, either in the form of full source code or as a set of patches.

### *8.1. Frame Dropping in* `mplayer`

In its original configuration `mplayer` uses, in order of deciding if a frame should be dropped, the following information:

- an estimation of the current A/V delay;
- the number of frames that are being dropped in the current dropping burst (i.e., how many frames have been dropped continuously, one right after the other; it is reset to zero as soon as one frame is not dropped);
- the timestamp of the next frame, which depends on the stream format.

These are put together into an heuristic. The objectives are:

- keeping the A/V delay under control, ideally below $100ms$;

[8] More information at: `http://www.mpeg.org/MPEG/video/`.

- dropping as few frames as possible, with special attention to avoiding bursts of dropped frames.

This approach has repeatedly proven to be effective, during the many years of development of the `mplayer`, and leads to a remarkable performance. However, at least two main drawbacks can be recognized:

1. it is not immediate to understand how the heuristic works, even looking carefully at the code and trying to track from where each term comes from;
2. the drop/no-drop decision is taken *in advance*, i.e., without being sure whether or not the actual decoding of that frame will negatively affect the A/V delay.

There therefore might be situations in which frames that it would have been possible to decode in time are dropped or, on the other hand, in which the heuristic decides to decode a frame that pushes the A/V delay outside of the desired window because it reveals to be unexpectedly complex.

### *8.2. Frame Dropping in* `mplayer-dlex`

Both audio and video data in a typical multimedia stream have some temporal information attached to them. For example each frame $i$ has its own *presentation timestamp* ($v\_pts_i$). The most natural setting for a possible video decoding deadline seems to be the time by which a frame must be displayed on the screen, which for a $25fps$ video happens every $40msec$. However, since some maintenance operations have to be performed per each frame as well, a safer value for such a deadline has been determined to be $37msec$. Such a value has proven — during preliminary experiments — to be high enough for guaranteeing drop free playback in "normal" (e.g., `other` scheduling policy on an unloaded system) conditions, and it also leaves the player some time for preparing the decode and playback of the next frame, even if some load is added. Moreover, since `mplayer` is particularly interested in controlling the A/V delay of the playback, the deadline should reflect this as well. However, bending too much toward an A/V delay based heuristic would defeat the main purpose of this paper, which is to show how the proposed framework allows for explicitly dealing with the timing constraints of time-sensitive applications (which in this case are well stated and quite clear), in favor of rule of thumbs or trial and error approaches. Therefore, the deadline of a frame decoding operation is determined by the following equation, for the $i-th$ frame:

$$\texttt{deadline}_\texttt{i} = 0.037 - \frac{(\texttt{audio\_time} - \texttt{v\_pts}_\texttt{i})}{100} sec$$

This means some extra decoding time is available in case audio is currently backward with respect to video. On the other hand, if video is lagging, the more the delay, the faster the decoding is asked to be. In fact, following the discussion above, the influence of the actual A/V delay the playback is experiencing is being considered in the formula, but only with the 1% of its "weight".

The exceptions mechanism described in the previous sections has been leveraged to modify the A/V control loop inside `mplayer-dlex` as follows. First, the frame decoding function has been enclosed into a

`try_within_rel(deadline){}` block. Second, the decoder itself has been configured so as to never drop a frame. In fact, frame dropping is automatically enforced by the deadline exception as soon as the imposed deadline is reached. However, it must also be guaranteed that, even if the deadline is being violated, a reference frame (i.e., a non B frame in MPEG and in H.264 too) is always decoded. This is possible in the OML exceptions framework thanks to the atomic code sections. In fact, it is sufficient to disable (`oml_try_within_disable()`) the notification of timing constraints violations until the decoding library retrieves enough information to decide which type of frame it is manipulating. At this point, if the frame is droppable, the notification mechanism is re-enabled (`oml_try_within_enable()`). Thanks to the use of POSIX real-time signals, this also guarantees that, if the deadline expired during the non-interruptible period, the violation is immediately notified after it finishes.

*8.3. Experimental Evaluation*

An experimental evaluation of the performance of `mplayer` and `mplayer-dlex` has been done by playing 100 times an 812 frames video (33*sec* duration) together with two (synthetic) time sensitive applications, under different scheduling policies. This resulted in more than 7 hours of playback, the most notable and representative results of which are summarized in this section. The platform utilized is the same already described in Sec. 7, and the results refers to the playback of the "Big Buck Bunny" movie trailer[9] in H.264 format at 1920x1080 resolution.

The main focus of this evaluation is to highlight the behavior of `mplayer-dlex`, with respect to original `mplayer`, while the playback is run concurrently with other real-time applications, using different scheduling policies, as already described in Sec. 7. In order of achieving this, two periodic tasks simulating the behavior of a typical real-time application have been added to the system. The program implementing them is called `rt-app`[10]. Both these threads have been configured to execute for a fixed amount of time, $C_1$ and $C_2$, and it is required for each instance to finish before a deadline, which was equal to the period of the threads themselves, $P_1$ and $P_2$. This means, if a particular instance $i$ of the first `rt-app` starts at time $r_1^i$, executes for $C_1$ and finishes at $f_1^i$, then $f_1^i \leq d_1^i = r_1^i + P_1$ must hold, in order for `rt-app-1` to meet its timing constraint. Numbers for many performance metrics have been gathered during the evaluation, this section reports the ones that are considered to be the most useful for showing the behavior of all the tested scenarios. For `mplayer` and `mplayer-dlex` the A/V delay and the number of dropped frames are considered the most representative indicators of a smooth playback of the video. Obviously, the A/V delay is expected to be both as low (ideally zero) and as stable (ideally zero variance) as possible, and the number of dropped frames should be small (ideally zero) as well. Figures for average and standard deviation of the A/V delay, number of dropped frames, and a combination of these two are reported below and discussed in this section. For the

[9] http://www.bigbuckbunny.org

[10] More info at: `https://launchpad.net/rt-app`

synthetic real-time tasks — the `rt-apps` — the amount of time left from the finishing time to the deadline (slack) or the time by which the deadline is missed (tardiness) is considered, as it is given by $sl_1^i = d_1^i - f_1^i$. In the ideal situation, both applications should be able to meet all their deadlines in each of their instances, which would mean $sl_{1,2}^i \geq 0, \forall i$.

*8.3.1. Experimental Setup*

Two sets of experiments have been executed, both with two instances of `rt-app` running with the following parameters: `rt-app-1` $= (C_1 = 3msec, P_1 = 8msec)$, `rt-app-2` $= (C_2 = 20msec, P_2 = 200msec)$. These values have been chosen since we wanted a scenario with sufficiently heterogeneous application periods. This happened concurrently with an instance of `mplayer`, in the first set of experiments, and of `mplayer-dlex`, in the second. As per the scheduling policies, the same algorithm has been used for `mplayer` (or `mplayer-dlex`) and the two `rt-apps`, at any time, as detailed in Tab. 2. Note that the reservation guaranteed to `mplayer(-dlex)` has $40msec$ as its period, reflecting the periodicity of the $25fps$ video playback, and an amount of budget that makes it possible for the player to use 37.5% of the CPU in every interval of such length. Note also that the reservations assigned to the two `rt-apps` have slightly more budget than their computation times. This should guarantee complete isolation among the three tasks and, since $\frac{3.15}{8} + \frac{21}{200} + \frac{15}{40} = .87375 \leq 1$, it is expected that `rt-app-1` and `rt-app-2` will not experience any deadline miss. For each line of Tab. 2, 100 tests with the described configuration have been performed.

| **experiment** | `mplayer(-dlex)` | `rt-app-1` | `rt-app-2` |
|---|---|---|---|
| **other** | `SCHED_NORMAL` | `SCHED_NORMAL` | `SCHED_NORMAL` |
| **fifo** | `SCHED_FIFO:5` | `SCHED_FIFO:5` | `SCHED_FIFO:5` |
| **rr** | `SCHED_RR:5` | `SCHED_RR:5` | `SCHED_RR:5` |
| **cbs** | `SCHED_DEADLINE:`$(15, 40)$ | `SCHED_DEADLINE:`$(3.15, 8)$ | `SCHED_DEADLINE:`$(21, 200)$ |

Table 2: Scheduling policies and parameters used for the various experiments. for `SCHED_FIFO` and `SCHED_RR` 5 is the POSIX real-time priority utilized, while for `SCHED_DEADLINE` $(budget, period)$ are the budget and the period (respectively) of the reservation, expressed in $msec$.

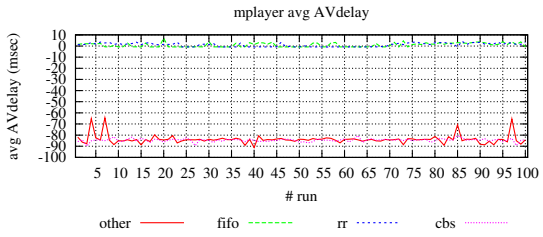
It would be possible, in the `fifo` and `rr` cases, to use different priorities among `mplayer(-dlex)` and the `rt-apps`, and maybe to chose two different values for the priority of two `rt-apps` as well. However, this would either mean that any of the above applications is more important than the others, which is not true for this study, or that priorities are being assigned according to the timing characteristics of the applications, e.g., trying to enforce Rate Monotonic ordering. The latter is certainly feasible in such a simple example with just 3 applications, however it was not done as it generally requires to know in advance the number and periodicity of all the time-sensitive applications that may be present in the system, something hardly realistic in an open environment.

*Results for* `mplayer`. Figure 7 shows the performance of `mplayer` while running concurrently with the two `rt-app`s at the various scheduling policies.

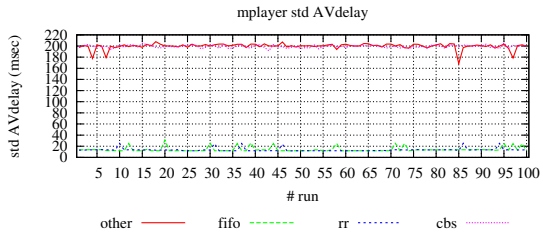
It depicts that, using `other` and `cbs` policies for `mplayer` results in considerable amount of A/V desynchronization ($\sim 85msec$ on average, 7(a)) and, even worse, that this A/V delay varies a lot among the various frames during the playback (7(b)). Moreover, this happens even if $\sim 55$ frames are dropped during each of the 100 experiments, i.e., $\sim 7.3\%$ of the total frame count (7(c)). This appears even more evident in Fig. 7(d), where the total number of dropped frames in each experiment is plotted against the average A/V delay. In fact, all the samples for runs at `other` and `cbs` are far from the origin, which clearly means bad performance and jerky playback experience. This is happening while running at `other` mainly because of the interference on `mplayer` of the two instances of `rt-app`, but it is also affecting `cbs` since the CPU bandwidth reserved to `mplayer` is not enough for guaranteeing in-time frame processing and smooth playback, at least with the standard frame dropping policy. However, looking at the other graphs in Fig.7 the differences among `other` and `cbs` clearly emerge. In fact, with `other`, at least one of the instances of `rt-app-1`, in any of the 100 runs, misses its deadline, as denoted by Fig. 7(e) in which the minimum slack time over all the instances shows always as a negative value. Moreover, this is the case also for the $5^{th}$ percentile and in most of the cases for the $25^{th}$ percentile (7(f) and 7(g)) of the distribution of the slacks, while the average value of them across all the instances of each run (7(h)) is just slightly positive. On the other hand the strict temporal isolation that `cbs` is able to enforce among the three applications ensures that, although the slack time might be small, no deadline is ever missed by `rt-app-1` in any of the performed experiments. Note that `rt-app-2` never misses any deadline neither while running at `cbs` nor at `other` and that this is something that could have been expected. In fact, among the two `rt-app`s, the interference from the other applications (i.e., the other `rt-app` and `mplayer`) is more likely to cause a deadline miss in `rt-app-1` than in `rt-app-2`, since the former has a by far shorter period (i.e., deadline), and thus much shorter tolerance for timing anomalies.

Similar conclusions can be drawn by looking at the `fifo` and `rr` curves in all the cited graphs. Main differences are that the performance of `mplayer` is significantly improved, as clearly shown by the clusters of points, all near the origin, in 7(d). On the other hand, the suffering of `rt-app-1` is much more worse, since not only the minimum, $5^{th}$ and $25^{th}$ of the slack time record constant deadline misses, but also the average slack time across all the instances in each of the 100 experiments is always negative (`fifo` and `rr` curves in 7(h).

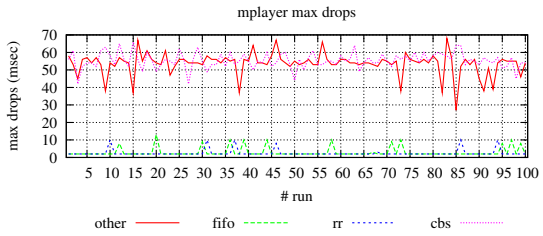
Summarizing, the standard unmodified `mplayer` run with `rt` (i.e., `fifo` or `rr`) policies achieves nearly perfect playback performance, but this comes at the cost of completely wasted behavior of other time-sensitive activities, especially if small and frequent. Similarly, the Linux most commonly used scheduling policy, `other`, behaves less disruptively with respect to `rt-app`s, but then it is the video playback which gets seriously affected. Finally, resource reservation scheduling, in this case `cbs`, is able to achieve isolation



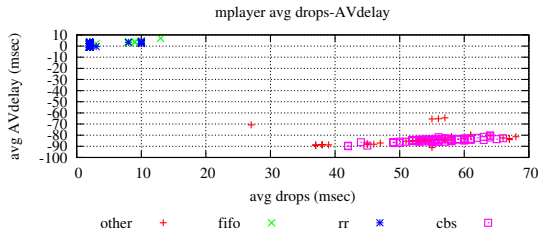

(a) Average A/V delay



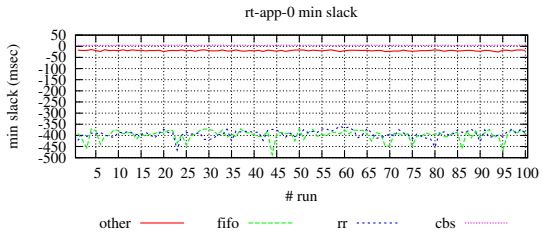

(b) Standard deviation of the A/V delay



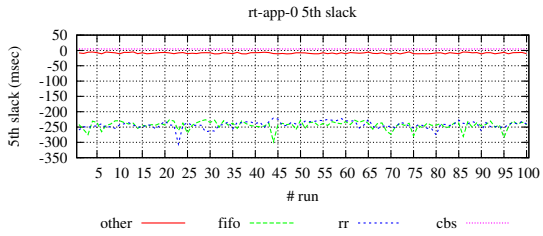

(c) Number of dropped frames



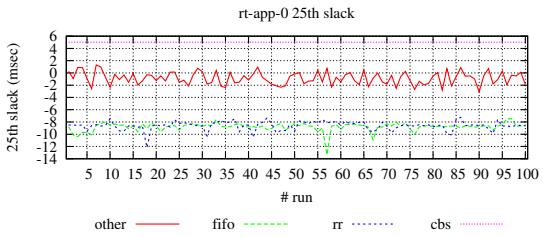

(d) A/V delay vs. dropped frames



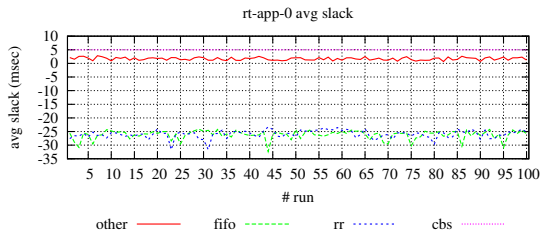

(e) $1^{st}$ `rtapp` minimum slack time



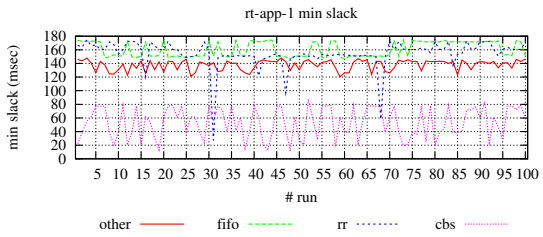

(f) $1^{st}$ `rtapp` $5^{th}$ percentile of the slack time



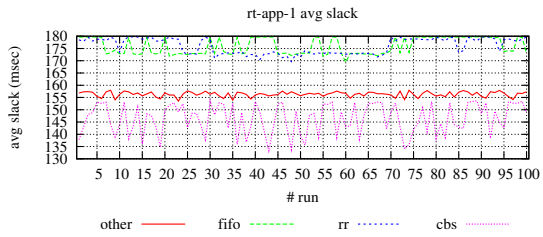

(g) $1^{st}$ `rtapp` $25^{th}$ percentile of the slack time




(h) $1^{st}$ `rtapp` average slack time




(i) $2^{nd}$ `rtapp` minimum slack time




(j) $2^{nd}$ `rtapp` average slack time

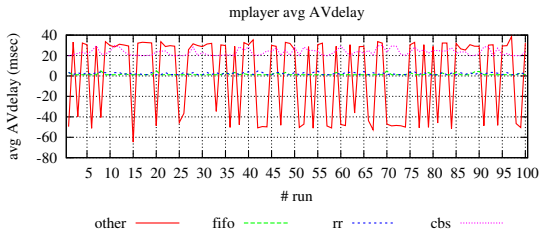
Figure 7: Performances of the original `mplayer` with the two `rt-app`s, all running at various scheduling policies.

among the different applications, allowing the `rt-app`s to meet their deadlines, but in this case this also means the CPU bandwidth that is then available for `mplayer` is not enough for decent HD video playback, at least with the standard frame-dropping heuristic in place.

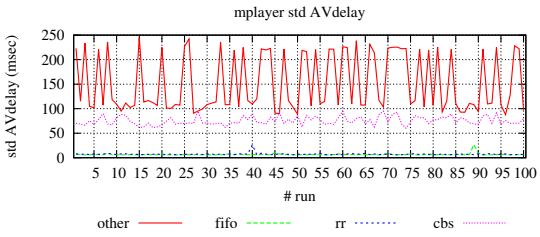
*Results for `mplayer-dlex`.* Figure 8 reports the very same evaluation, but in the case of the OML-modified version of the player, `mplayer-dlex`.

The plots related to the playback (i.e., 8(a) — 8(d)) are the most interesting ones, especially if compared with the matching ones in Fig. 7. In some more details, average A/V delay is, again, low and stable for all the runs at `fifo` and `rr`, and with almost no frame dropping, but then `rt-app-1` misses a great deal of its deadlines. At `other` the number of dropped frames is reasonably small, but there still are runs where the A/V delay is well noticeable and, as in the previous paragraph, `rt-app-1` misses some deadlines in this case too. Things get better when considering the `cbs`, since not only the two `rt-app`s make their deadlines, the A/V delay is acceptable and the number of dropped frames often stays below 10. Graphs about the behavior of the `rt-app`s are also reported in Figure 8, for the sake of completeness. However, as it could have been expected, they behave the same as in the `mplayer` case, although the fact that `mplayer-dlex` is more clever in deciding how to use the CPU it is entitled. The various schedulers seems to yield some beneficial effect to them too, as the various statistics of the slack time of both applications look more regular and stable.

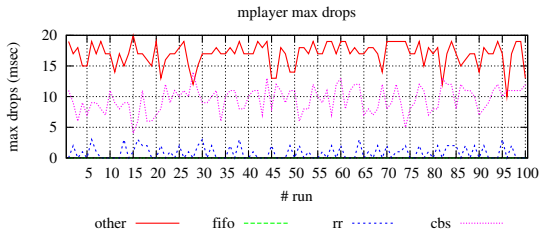
*Comparing `mplayer` and `mplayer-dlex`.* Looking at Fig. 7 and Fig. 8 all together gives the idea of whether or not the adoption of the deadline driven dropping logic brings any benefits to the playback, at the various scheduling policies. The average and the standard deviation of the A/V delay were already very good in Fig. 7(a) and 7(b) for `fifo` and `rr`, and still they improve in Fig. 8(a) and 8(b), where they look even more flat and closer to the x axis (especially the standard deviation). The same happens to the number of dropped frames, since it stays below 5 frames for any of the runs in 8(c), while it was reaching 10 for some of the runs in 7(c). The very same thing occurs in the `cbs` case, since the A/V delay dances around $20 \pm 65$ in case of `mplayer-dlex`, versus the $-85 \pm 200$ of `mplayer`, in the various runs; the number of dropped frames lowers itself up to $\sim 8$ from $\sim 55$, respectively. `other` shows, in the `mplayer-dlex` case, quite some variability among the various runs, but the benefits of optimistic, deadline based dropping are still visible. In fact, the number of dropped frames stays well below 20, while it reached 70 for `mplayer`; on its hand, the A/V delay turned into something varying between 30 and $-50$, with a standard deviation between 100 and 250, while it was $-90 \pm 200$ for almost all the runs in the `mplayer` case. All the above is confirmed by 8(d), where it is even more evident that, by switching to the deadline exception based frame dropping logic, the points move closer to $(0, 0)$ with respect to 7(d), for all the considered scheduling policies, in particular for `cbs` and `other`. Furthermore, given the performance of the `rt-app`s, it appears quite clear that the proposed timing exception framework, in combination with a resource reservation enabled real-time scheduling (`cbs`) is what gives the best results, for the purposes of this paper. In fact, `mplayer-dlex`+`cbs` achieves reasonably small



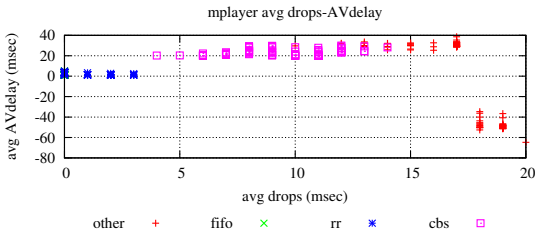

(a) Average A/V delay



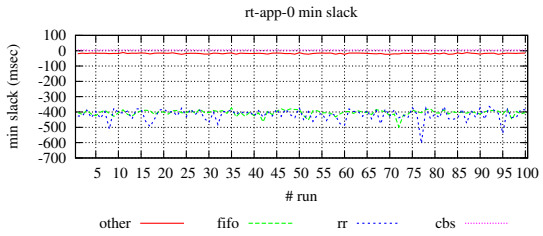

(b) Standard deviation of the A/V delay



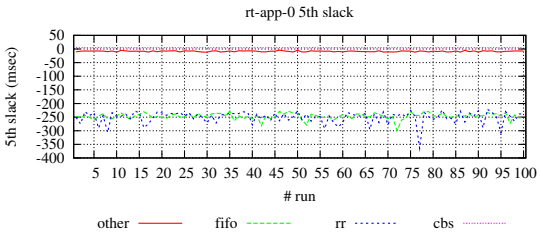

(c) Number of dropped frames



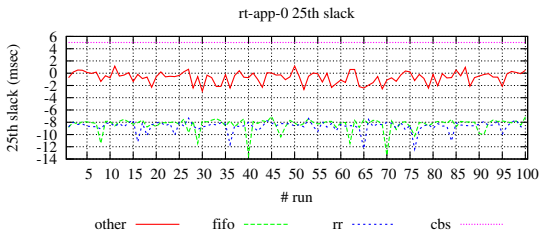

(d) A/V delay vs. dropped frames



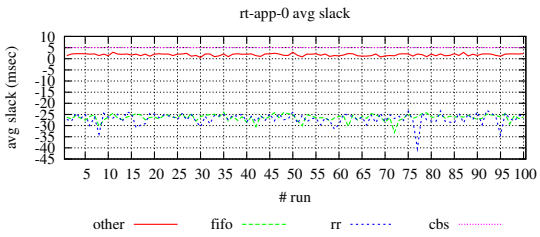

(e) $1^{st}$ `rtapp` minimum slack time



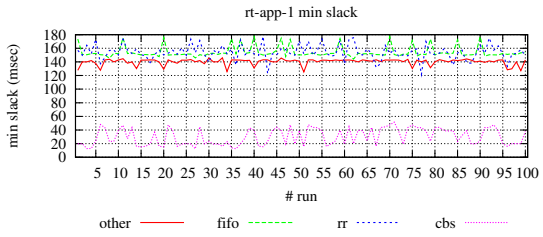

(f) $1^{st}$ `rtapp` $5^{th}$ percentile of the slack time



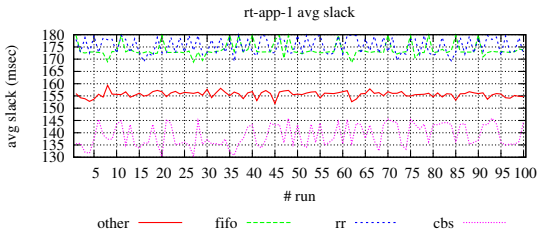

(g) $1^{st}$ `rtapp` $25^{th}$ percentile of the slack time




(h) $1^{st}$ `rtapp` average slack time




(i) $2^{nd}$ `rtapp` minimum slack time




(j) $2^{nd}$ `rtapp` average slack time

Figure 8: Performances of the OML, deadline constrained, `mplayer-dlex` and the two `rt-app`s, all running at various scheduling policies.

frame dropping and low and fairly stable (as stated by 8(b)) A/V delay. A summary of the metrics for the original mplayer and the modified one are also reported in Table 3 for the reader's convenience.

| | mplayer | | mplayer-dlex | |
|---|---|---|---|---|
| **experiment** | A/V desync | Dropped frames | A/V desync | Dropped frames |
| **other** | $-83.805 \pm 4.072$ | $44.032 \pm 4.76$ | $0.806 \pm 38.508$ | $9.87 \pm 1.027$ |
| **fifo** | $0.88 \pm 1.82$ | $2.64 \pm 1.97$ | $1.6694 \pm 0.97$ | $0 \pm 0$ |
| **rr** | $1.018 \pm 1.63$ | $2.45 \pm 1.82$ | $2.33 \pm 1.105$ | $0.48 \pm 0.62$ |
| **cbs** | $-84.53 \pm 1.49$ | $45.27 \pm 3.28$ | $23.25 \pm 3.16$ | $6.39 \pm 1.42$ |

Table 3: Summary of the most relevant metrics.

*Quality of Experience (QoE).* Measuring the number of dropped frames sometimes may not be representative of the impact of the missing frames on the observer experience. For example, in one case the player may drop more frames which are more similar to the ones that remain on the screen, as compared to another case, in which less frames with significant differences might be dropped. Therefore, we also compared the various cases by computing one of the quantitative Quality-of-Experience (QoE) metrics widely used in video processing evaluation [17], namely the Root Mean-Square Error (RMSE). The RMSE between two images is the average over all the color channels of the root mean square error between the color value over all the image pixels. The RMSE, or the Peak Signal to Noise Ratio (PSNR) that can be derived from it, may be conveniently used to evaluate different encoding/compression options in lossy encoders or unreliable real-time video transmission protocols. In such a case, in order to obtain the RMSE for the whole video, one would compare, image by image, the video obtained after decompression with the original one as available before compression (e.g., the raw video just output by a camera), then sum up the RMSE values for all the images.

In our case, as we only modified the frame dropping logic of the player, we compared the images that should have been displayed on the screen by the player if no frame dropping had occurred with the images that have actually been displayed. In such a case, when a frame drop was detected, the image to compare with was obtained as the last non-dropped frame before the dropped one (for non-dropped frames, the contribution to the overall video RMSE is null in our case). We preferred the simpler RMSE metric to the PSNR merely because we compared videos that were almost identical except for a few possibly dropped frames (the RMSE over all the frames can easily be summed up, as it is null on identical frames, but the PSNR would go to infinity on such frames). A more extensive comparison of the impact on different QoE metrics, including more sophisticated approaches such as the Structure Similarity Index [18] (SSI) or others is out of the scope of the present work and may be deferred to future work.

The obtained RMSE values for all the considered cases are reported in Table 4. As we performed 100 repetitions of the same experiment for each case, we reported the average and standard deviation of the

obtained RMSE values across the 100 runs, for each considered case.

| **experiment** | mplayer RMSE | mplayer-dlex RMSE |
|---|---|---|
| **other** | $172100 \pm 34963$ | $65297 \pm 12412$ |
| **fifo** | $1805.06 \pm 7877.44$ | $0 \pm 0$ |
| **rr** | $547.942 \pm 949.696$ | $2989.2 \pm 4369.48$ |
| **cbs** | $177444 \pm 23894$ | $27388 \pm 5587.80$ |

Table 4: Obtained root mean square error (RMSE) figures as due to frame dropping. The average and standard deviation of the values across the 100 runs are shown.

As it can be seen, the modified frame-dropping logic results almost always in reduced RMSE values. Specifically, when scheduling the player with the default Linux policy (first row), `mplayer-dlex` achieves an average RMSE of 65297 against the value of 172100 achieved by the original `mplayer`, i.e., a reduction of the 62.06% of the RMSE. With a deadline-based scheduler (bottommost row), we obtained an average RMSE of 27388 instead of 177444, i.e., a reduction of the 84.56%. Only with the round robin scheduler (third row) we obtained a higher average RMSE with the modified player. However, in this case the RMSE value is very small, due to the very low number of dropped frames (the average number of dropped frames is reported in Table 3). Therefore, the result may be considered as less accurate (and this consideration applies also to the 100% reduction of the RMSE for the FIFO scheduling case).

*Concluding remarks.* It is therefore possible to conclude that `mplayer-dlex` behaves better than `mplayer`, yielding similar A/V delay average values, but with smaller standard deviation (meaning more regular playback) and a noticeably reduced number of dropped frames with all the scheduling policies. Also, in almost all the cases `mplayer-dlex` achieved highly reduced RMSE values as compared to the original unmodified player. This proves the point that the deadline-exception based frame dropping, investigated in this study, succeeded in better capturing the specific timing behavior of the video player. More specifically, it helped in correcting a significant amount of the frame-dropping decisions that are mistaken by the original mplayer heuristic.

## 9. Conclusions and Future Works

In this paper, a mechanism for the management of timing constraints violations according to the well-known exception-based paradigm has been envisioned. A set of basic requirements has been identified, inspired by real-world multimedia and control scenarios, and a framework that fulfils all of them has been presented, along with its implementation as a part of an open-source library for C applications. This constitutes a valuable support for developers of embedded soft real-time and control applications, since it allows them to concentrate on the main application flow of control which will be executed most of the

times. The code that deals with anomalies, even in the domain of the timing behavior of the application, will be then provided in the form of an exception handler, with the framework responsible for jumping and executing it whenever it is the case.

An implementation of the proposed mechanism for POSIX compliant Operating Systems has been presented. Also, a performance evaluation has been carried out under Linux, where the latency involved in the activation of the management code has also been measured. Moreover, a real multimedia player (the `mplayer`) for Linux has been instrumented to use the framework. The gathered experimental results demonstrate that the proposed mechanism allows for better adherence to the intrinsic timing of the application, bringing an increase in the performance in the form of: reduced audio-video delay, reduced number of dropped frames and reduced root mean square error figures. This demonstrates that our technique has the potential to bring significant improvements in terms of users' Quality of Experience while watching videos with `mplayer`.

Concerning possible directions for future work, a kernel-level mechanism is being investigated for Linux, that will lead to a further reduction of the notification latency. Furthermore, a more ambitious macro-based framework for C is under design that will enrich OML with generic constructs for threads creation, management, synchronization etc. Finally, investigation is in progress on how to integrate the proposed framework with the many existing real-time schedulers available for the Linux kernel, such as SCHED_DEADLINE [16], the hybrid EDF/FP scheduler presented in [19], the POSIX compliant Sporadic Server [20] or the Adaptive QoS Architecture for Linux [21].

## 10. Acknowledgements

We would like to thank the European Community's Seventh Framework Programme FP7, having funded the research leading to these results under grant agreement n.214777 "IRMOS — Interactive Realtime Multimedia Applications on Service Oriented Infrastructures" and n.248465 "S(o)OS — Service-oriented Operating Systems". Also, we would like to thank Luca Abeni for his continuous and passionate support for multimedia and especially video-related issues. Finally, we would like to thank the anonymous reviewers who helped us improve this manuscript with their insightful comments.

## References

[1] T. Cucinotta, D. Faggioli, An exception based approach to timing constraints violations in real-time and multimedia applications, in: Proceedings of the IEEE Symposium on Industrial Embedded Systems (SIES 2010), Trento, Italy, pp. 136–145.

[2] S. A. Edwards, E. A. Lee, The case for the precision timed (pret) machine, in: Proceedings of the $44^{th}$ annual conference on Design automation (DAC'07), ACM, New York, NY, USA, 2007, pp. 264–265.

[3] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, E. A. Lee, Predictable programming on a precision timed architecture, in: Proceedings of the International Conference on Compilers, Architecture, Synthesis for Embedded Systems (CASES), Atlanta, Georgia, United States, pp. 137–146.

[4] IEEE, Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions., 2004.

[5] G. Bollella, J. Gosling, The real-time specification for java, Computer 33 (2000) 47–54.

[6] A. Burns, A. Wellings, Concurrent and Real-Time Programming in Ada 2005, Cambridge University Press, 2007.

[7] H. Winroth, Exception Handling in ANSI C, Technical Report ISRN KTH NA/P–93/15–SE, Royal Institute of Technology (KTH), Stockholm, Sweden, 1993.

[8] I. Lee, S. Davidson, V. Wolfe, RTC: language support for real-time concurrency, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS 91), IEEE, San Antonio, TX, USA, 1991, pp. 43–52.

[9] I. Lee, V. Gehlot, Language Constructs for Distributed Real-Time Programming, Technical Report, University of Pennsylvania, 1985.

[10] J. W. S. Liu, K.-J. Lin, R. Bettati, D. Hull, A. Yu, Foundations of Dependable Computing, volume 284 of *The International Series in Engineering and Computer Science*, Springer US, pp. 157–182.

[11] D. Hull, W. chun Feng, J. W. Liu, Operating system support for imprecise computation, in: In Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities, pp. 96–99.

[12] A. Quagli, D. Fontanelli, L. Greco, L. Palopoli, A. Bicchi, Designing real-time embedded controllers using the anytime computing paradigm, in: Proc. of the IEEE International Conference on Emerging Technologies Factory Automation (ETFA 2009), pp. 1–8.

[13] T. Cucinotta, D. Faggioli, A. Evangelista, Exception-based management of timing constraints violations for soft real-time applications, in: Proceedings of the $5^{th}$ International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009), Dublin, Ireland, pp. 40–48.

[14] Intel, IA-PC HPET (High Precision Event Timers) Specification (revision 1.0a), October 2004.

[15] L. Abeni, G. Buttazzo, Integrating multimedia applications in hard real-time systems, in: Proceedings of the IEEE Real-Time Systems Symposium, Madrid, Spain, pp. 4–13.

[16] D. Faggioli, F. Checconi, M. Trimarchi, C. Scordino, An edf scheduling class for the linux kernel, in: Proceedings of the $11^{th}$ Real-Time Linux Workshop (RTLWS 2009), Dresden, Germany.

[17] C. Kiraly, L. Abeni, R. Lo Cigno, Effects of p2p streaming on video quality, in: ICC 2010 - 2010 IEEE International Conference on Communications, IEEE, IEEE, Cape Town, South Africa, 2010, pp. 1 – 5.

[18] Z. Wang, A. Bovik, H. Sheikh, E. Simoncelli, Image quality assessment: from error visibility to structural similarity, Image Processing, IEEE Transactions on 13 (2004) 600 –612.

[19] F. Checconi, T. Cucinotta, D. Faggioli, G. Lipari, Hierarchical multiprocessor CPU reservations for the linux kernel, in: Proceedings of the $5^{th}$ International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009), Dublin, Ireland, pp. 15–22.

[20] D. Faggioli, G. Lipari, T. Cucinotta, An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the linux kernel, in: Proceedings of the $4^{th}$ International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2008), Prague, Czech Republic, pp. 5–14.

[21] L. Palopoli, T. Cucinotta, L. Marzario, G. Lipari, AQuoSA — adaptive quality of service architecture, Software – Practice and Experience 39 (2009) 1–31.