An Exception Based Approach to Timing Constraints Violations in Real-Time and Multimedia Applications.

Tommaso Cucinotta and Dario Faggioli
Real-Time Systems Lab, CEIICP
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 envisioned approach has been validated by modifying `mplayer`, one of the most widely used multimedia player for Linux, so as to use the introduced library. Experimental results demonstrate how the exception-based paradigm is effective in improving the audio/video delay exhibited by the player.

I. 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 ...`).

a) 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

The research leading to these results has received funding from the European Communitys Seventh Framework Programme FP7 under grant agreement n.214777 "IRMOS — Interactive Realtime Multimedia Applications on Service Oriented Infrastructures" and n.248465 "S(o)OS — Service-oriented Operating Systems".

showing results gathered modifying a real existing multimedia applications.

b) Organization of the Paper: After a brief overview of the related work in Section II, Section III describes some possible utilization scenarios for the framework. Section IV identifies the main technical requirements that need to be supported by the mechanism, and Section V describes the fundamentals characteristics of the library implementing such requirements. Section VI illustrates the POSIX-based implementation realized for the Linux OS. Finally, Section VII reports some performance evaluation measurements, highlighting the impact of the Linux kernel configuration on the mechanism precision, while Section VIII 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 IX along with directions for future work.

II. 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 for 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 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.

Focusing on the C language, the RTC approach proposed by Lee et al. [7] 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.

Finally, the concept of *time-scope* introduced in [8] 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.

III. 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 multimedia player, and (ii) an embedded control scenario making use of an "anytime algorithm".

A. Video Player

For (i), consider a multimedia 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 if the decoder

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

[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.

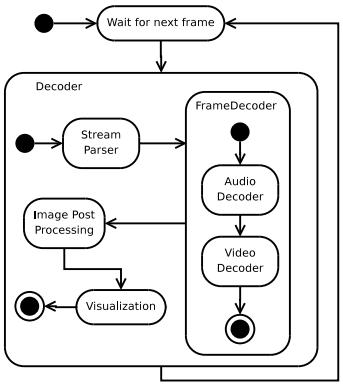


Fig. 1. UML Activity diagram for the example video decoder thread.

is lacking time. Therefore, the player described by Fig. 1 may be implemented as follows:

```
void FrameDecoder(raw_frame_t f, dec_frame_t *d) {
  AudioDecoder(f, d);
  VideoDecoder(f, d);
}

void Decoder(stream_t *s) {
  raw_frame_t f;
  dec_frame_t d;
  while (1) {
    f = StreamParser(s);
    try(T) {
      /* Aborted if still inside at time T */
      FrameDecoder(f, &d);
      ImagePostProcessing(&d);
    }
    /* If aborted, re-use last decoded frame */
    Visualization(d);
    wait_for_next_frame(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 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.

B. Anytime Algorithms

For what concerns anytime algorithms, they have been theorized in real-time systems from long time, for enhancing flexibility [9], [10]. 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. [11], 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 C; /* Computation time for the whole instance */
int D1, D2; /* Computation time of optional parts */
while (1) {
  set_computation_time(C);
  start_computation();
  res = mandatory_computation();
  try(C) { /* Aborted if execution exceeds C */
    res = optional_computation_1(res);
    res = optional_computation_2(res);
  } catch {
    end_computation(res);
  }
  wait_next_period();
}
```

Notice that in both cases the overall results comes from the "merge" of the intermediate results of the various computation phase, performed by each computation phase itself (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.

C. 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 always may be some special code segments the asynchronous interruption of which should

be avoided. For this reason, it would be useful to have a mechanism to temporarily stop the notification of an exception and 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.

IV. 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 some "protected" section of a code segment that will never be interrupted by a timing constraint violation notification. Thus, if that happens, the execution of recovery code would be delayed while inside such a section.
- Optional requirements:
  Requirement 10: the mechanism could provide a method for monitoring the time remaining before the specified constraint violation to occur;
  Requirement 11: the mechanism could provide a method for checking if a constraint violation has been notified with the correct timing and, if not, how much is the difference between the expected and the actual (i.e., the latency) notification of such violation;
  Requirement 12: the mechanism could provide support for gathering benchmarking data of the code segments, instead of enforcing their timing-constraints. This operational 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.

V. Proposed approach

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

A. 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 throughout 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 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 [1].

B. 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;
- `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;

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

- `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 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.

```
#include <oml_exceptions.h>

int FrameDecoder(raw_frame_t f, dec_frame_t *d) {
  int rv = 0;
  oml_try_within_wcet(12000) {
    AudioDecoder(f, d);
    VideoDecoder(f, d);
  }
  oml_handle
    when (oml_ex_wcet_violation) {
      rv = -1;
    }
  oml_end;

  return rv;
}

void Decoder(stream_t *s) {
  raw_frame_t f;
  dec_frame_t d, d_old;
  while(1) {
    f = StreamParser(s);
    oml_try_within_rel(f->next_frame_pts) {
      if (FrameDecoder(f, &d) == 0)
        ImagePostProcessing(&d);
    }
    oml_handle
      when (oml_ex_deadline_violation) {
        d = d_old;
      }
    oml_end;
    Visualization(d);
    d_old = d;
    wait_for_next_frame(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;
oml_try_within_abs(next_dl) {
  /* safely interruptible computations */
  ...
  oml_try_within_disable();
  p_obj = malloc(sizeof(struct my_object));
  if (p_obj == NULL)
    throw(ENoMemoryException);
  my_object_init(p_obj); /* Constructor */
  oml_try_within_enable();
  /* safely interruptible computations */
  ...
} oml_finally {
  /* Free allocated resources */
  if (p_obj != NULL) {
    my_object_cleanup(p_obj); / * Destructor */
    free(p_obj);
    p_obj = NULL;
  }
}
oml_handle
  oml_when (oml_ex_deadline_exception) {
      /* Recovery logic */
  }
oml_end;
```

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

VI. Implementation

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

A. 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.

B. 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

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

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.

C. 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.

D. 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.

E. 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.

F. 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 [12]. 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.

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

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 1000 Hz tick frequency being probably the best choice.

VII. Performance Evaluation

The performance metric that it is interesting to evaluate, as identified by Requirement 8, is the *notification latency*, i.e., the time interval between the actual constraint violation and the instant the proper thread in the application is notified.

In fact — even if it is not strictly necessary to achieve an exact worst case case upper bound to such value, since the framework is mainly aimed at soft real-time systems — the developer must know that the latency is small, if compared to the execution times and deadlines of its code segments, otherwise the whole mechanism would become useless.

In order to perform this measurements, a standard distribution of the GNU/Linux OS, with kernel version 2.6.31, has been used, running on a commonly available desktop PC, with 3.0 GHz Intel CPU and 2 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 only one thread, and it was the only application running in the system (obviously, together with the minimum possible set of system daemons and maintenance programs), such that the measurements are not affected by "external" sources of latencies, e.g., determined by the scheduler, just to cite the most relevant one[6]. The WCET, relative deadline and period of the test application were equal to $(C, D, T) = (50, 50, 100)\ msec$, and 1000 consecutive instances of it have been run. All the experiments have been performed three times, with the OS periodic tick frequency set to 100, 250 and 1000 Hz, respectively.

In the first experiment, the latency of a deadline violation is measured 1000 times. This is done by forcing the thread to execute more than $50\ msec$ inside a `try_within_rel` block, and then subtracting the ideal deadline violation instance — $i \cdot T + D$ — from the actual time instant $\dot{D}$ at which the deadline miss signal handler is invoked. Results are shown in Table I and Fig. 2. They clearly demonstrate that the notification latency of a deadline constraint is both (i) small and (ii) independent from the tick frequency, thanks to the high resolution timers. This can be easily deducted by the fact that values in Table I are comparable, and the three CDF in Fig. 2 are completely superimposed. The measured latency values are in the order of the μs, what constitutes a more than acceptable performance.

In the second experiment, the thread again executes more than $50\ msec$, this time from inside a `try_within_wcet`

[6]Notice that, if coping with that specific source of latency is necessary, proper actions have to be undertaken, such as adopting a real-time scheduler and application design strategy.

	max	mean	std. dev.
HZ=100	28.61	1.724	1.189
HZ=250	17.202	1.595	0.711
HZ=1000	33.394	1.603	1.023

TABLE I
DEADLINE LATENCY IN μs

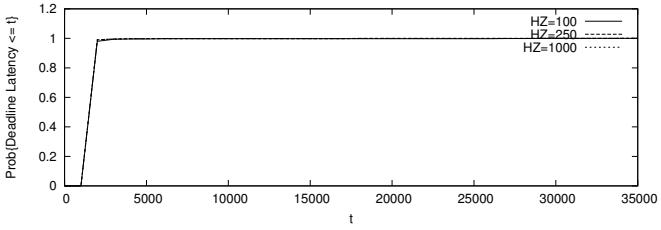


Fig. 2. Cumulative Distribution Function of the deadline violation latencies. Time on x-axis is in ns.

block, so to cause a WCET violation and measure its latency as well. Results are available in Table II and in Fig. 3. This time, what appears quite clear is that the precision of a WCET violation is tightly coupled with the system tick frequency HZ. Table II also shows how the mean WCET latency is close to $\frac{HZ}{2}$, as obviously expected. Therefore, for the mechanism to be useful in dealing with WCET violations, the value of $HZ = 1000$ is strongly recommended.

Minimum achieved latency values are not reported since they are highly dependant on how close to a system tick (or, in general, an accounting event) a timing violation event occurs. Thus, since they depend on the actual alignment of the task and the OS events, they turn out to be unrelated to the system configuration and provide few information about the performance of the mechanism this section wanted to examine.

	max	mean	std. dev.
HZ=100	18727.747	5748.948	4474.772
HZ=250	4423.164	1233.955	844.593
HZ=1000	1999.752	522.229	390.837

TABLE II
WCET LATENCY IN μs

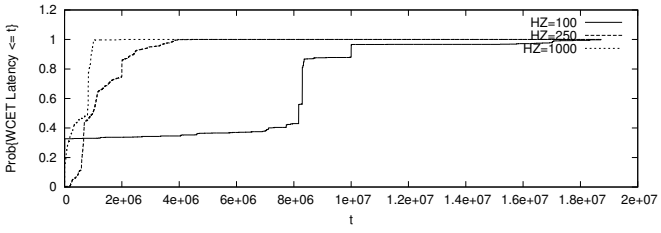


Fig. 3. Cumulative Distribution Function of the WCET violation latencies. Time on x-axis in ns

VIII. A CASE STUDY: THE `mplayer`

According to the project website[7] *"MPlayer is a movie player which runs on many systems. It plays most MPEG/VOB, AVI, Ogg/OGM, VIVO, ASF/WMA/WMV, QT/MOV/MP4, RealMedia, Matroska, NUT, NuppelVideo, FLI, YUV4MPEG, FILM, RoQ, PVA files, supported by many native, XAnim, and Win32 DLL codecs. You can watch VideoCD, SVCD, DVD, 3ivx, DivX 3/4/5, WMV and even H.264 movies"*. It is free and open-source 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 such audio data starts growing too large, something must be done 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 that can not be tolerated). Even if a frame has been decoded but it 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.

A. 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 much 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 $100ms$;
- dropping as few frames as possible, with special attention to avoiding bursts of dropped frames.

[7]More information at: http://www.mplayerhq.hu/.
[8]More information is available at: http://www.mpeg.org/MPEG/video/.

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 about the fact that the actual decoding of that frame will negatively affect the A/V delay or not.

Therefore, because of the latter point, there 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.

B. Frame Dropping in `mplayer-dlex`

Both audio and video data in a typical multimedia stream have some temporal information attached to them, the so-called *presentation timestamp* (a_pts and v_pts). Therefore, in order to introduce a deadline constraint for the decoding of a video frame, it seems reasonable to ask that frame i must be decoded by its pts_i. However, since `mplayer` is interested in controlling the A/V delay, if t is the current time when the decoding deadline d_i for the i-th frame is set, d_i may be defined as:

$$d_i = (v\_pts_i - t) + (a\_pts_i - t)$$

In order to avoid dropping of too many frames, it is useful to provider the decoder with some extra time, which in this work has been set to $50ms$.

Therefore, the exceptions mechanism described in the previous sections has been leveraged to modify the A/V control loop inside `mplayer` as follows. First, the frame decoding function has been enclosed into a `try_within_rel` 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., e 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 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, and if the deadline expired during the atomicity period, the violation is immediately notified.

Therefore, here it is how the most important code segments of `mplayer-dlex` look like:

```
static double update_video() {
  ...
  t = GetTimer();
  audio_pts = written_audio_pts(mpctx->sh_audio) -
              mpctx->delay + audio_delay;
  video_pts = sh_video->pts : mpctx->d_video->pts;
  dl = (video_pts - dt) + (audio_pts - dt)
       + VA_THRESHOLD;
  dl_ts = usec_to_timespec(dl * 1E6);
  oml_try_within_rel(dl_ts) {
    oml_try_within_disable();
    decoded_frame = decode_video(sh_video,
                                 sh_video->pts);
    ...
  }
  oml_handle
    oml_when(oml_ex_deadline_violation) {
      decoded_frame = NULL;
      ++drop_frame_cnt;
    }
  oml_end;
  ++total_frame_cnt;
  ...
}

  ...
  /* Then, in the H.264 decoding function */
  if (hx->slice_type_nos==FF_B_TYPE)
    oml_try_within_enable();
  ...
}
```

Notice how the code above looks similar to the one already shown as an example in Sec. III.

C. Experimental Evaluation

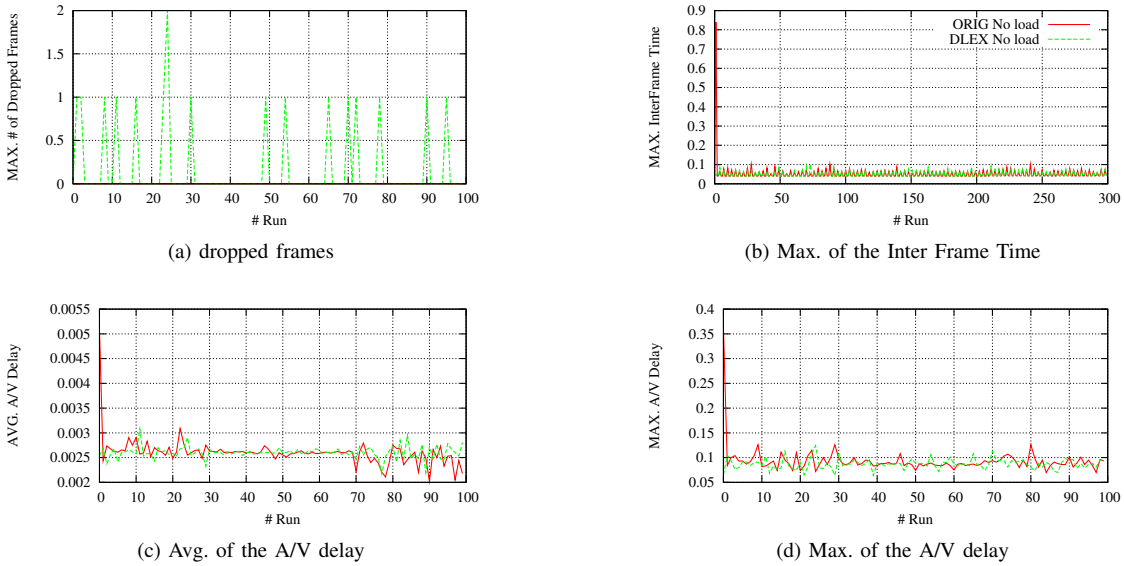
An experimental evaluation of the performance of `mplayer` and `mplayer-dlex` has been done by playing 100 times a video with a $33s$ duration, with different configuration with respect to frame dropping policy, and in different system load conditions. This resulted in more than 55 hours of playback, the most notable and representative results of which are summarized in this section.

The platform utilized is the same already described in Sec. VII, while the results reported here have been obtained playing the "Big Buck Bunny" movie trailer[9] in H.264 format at 1920x1080 resolution. The total number of frame of such video is 812.

In the experiments, the number of dropped frames, the maximum and average A/V delay, and the maximum inter-frame time (*IFT*) exhibited by both `mplayer` and `mplayer-dlex` have been measured, for each of the 100 runs and in different system load conditions. What would be desirable is the number of dropped frames and the A/V delay to be as small as possible — ideally 0, and the IFT, given the video is 25fps, to stay around $40ms$ for a regular playback.

In the first set of graphs in Fig. 4, for each one of the 100 run (i.e., each point represents one of them), the total number of dropped frames, the maximum IFT, and the maximum and average A/V delay are reported. In all the plots, the two curves are quite similar, if not super-imposed, which means the behavior of the original and the modified version matches, and thus that introducing the exception-based handling of frame dropping does not entail misbehavior of `mplayer-dlex` as compared to `mplayer` in this scenario.

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



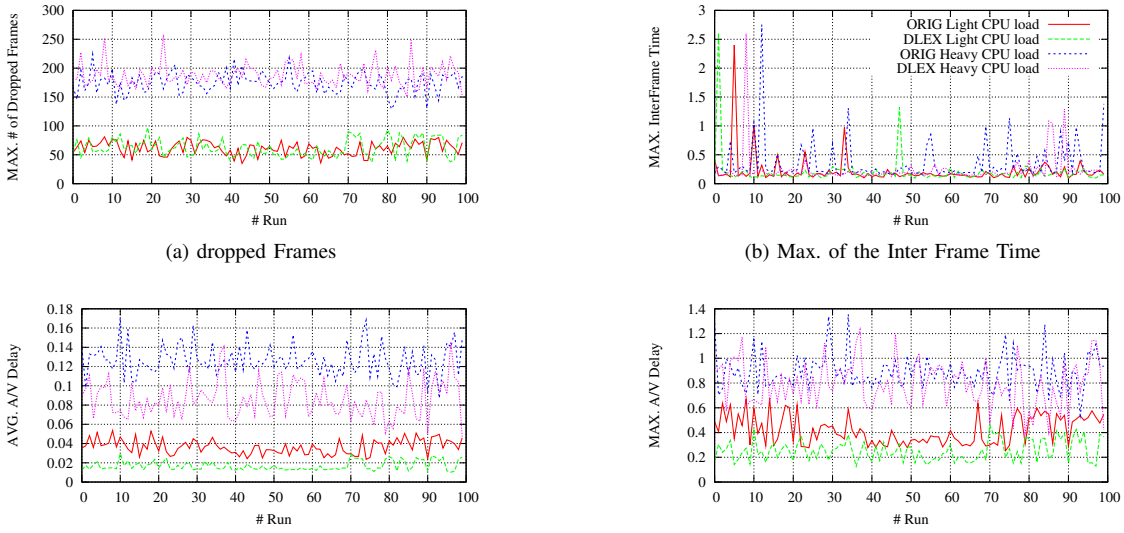
(a) dropped frames

(b) Max. of the Inter Frame Time

(c) Avg. of the A/V delay

(d) Max. of the A/V delay

Fig. 4. Total number of dropped frames; maximum inter-frame time; average and maximum of the A/V delay. Each for all the 100 runs of `mplayer`, *ORIG* curve, and `mplayer-dlex`, *DLEX* curve. No system load present except the video player.



(a) dropped Frames

(b) Max. of the Inter Frame Time

(c) avg. of the A/V delay

(d) max. of the A/V delay

Fig. 5. Total number of dropped frames; maximum inter-frame time; average and maximum of the A/V delay. Each for all the 100 runs of `mplayer`, *ORIG* curve, and `mplayer-dlex`, *DLEX* curve. Light or heavy system load were present.

The second set of results follows the same scheme, with the following differences:

- "ORIG and DLEX *Light CPU Load*" refers to runs of `mplayer` and `mplayer-dlex` disturbed by the `stress` benchmarking program[10] running 1 CPU and memory intensive thread plus 1 HDD massive reader/writer thread;
- "ORIG and DLEX *Heavy CPU Load*" refers to runs of `mplayer` and `mplayer-dlex` disturbed by, this time, 2 CPU and memory intensive threads and still 1 HDD massive reader/writer thread.

[10]http://weather.ou.edu/~apw/projects/stress/

2 applications that are always trying to execute on the CPU and continuously `malloc()`-ing and `free()`-ing memory are considered — together with the movie player — a heavy load since the machine only has a single processor. Therefore, given the best effort, fairness oriented, scheduling policy under which all these tasks are scheduled, and with 2 such CPU hogs and the resulting stress on memory, the player task can only access quite a small share of CPU time. This said, Fig. 5 shows, again, the total number of dropped frames, the maximum IFT, the average and the maximum A/V delay of each run. What emerges from the plot in Fig. 5a is that using

the original heuristic or the exception based approach does not make much difference in the number of dropped frames for the various runs. In fact, there is no clear domination of one curve on the other spanning all the runs, in neither of the two scenarios, and the number of frames dropped by the two versions is always comparable. On the contrary, Fig. 5b shows that, if the maximum IFT observed for each run is considered, DLEX curves exhibits fewer and smaller peaks, under both light and heavy system load. This means that `mplayer-dlex` is able to achieve more regular behavior than original `mplayer`. Finally, looking at Fig 5c and 5d, it appears quite clear that `mplayer-dlex` is often performing better, especially in the lightly loaded case, where the DLEX curve is practically always below the ORIG curve, in both graphs.

Therefore, it is possible to conclude that `mplayer-dlex` behaves better than `mplayer`, since it is capable of a more regular playback with smaller A/V delay while dropping almost the same number of frames. Thus, it can be said that the deadline-exception based frame dropping — that this case study attempted — succeeded in capturing the specific timing behavior of the video player, and it helps in dropping the frames that would otherwise be detrimental for the synchronous playback of audio and video, as well as in "saving" the others.

IX. 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 have 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 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 a reduced audio-video delay with similar number of dropped frames, which is something tightly related to the actual user experience while watching a video.

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 [13], the hybrid EDF/FP scheduler presented in [14], the POSIX compliant Sporadic Server [15] or the Adaptive QoS Architecture for Linux [16].

References

[1] T. Cucinotta, D. Faggioli, and 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, June 2009.

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

[3] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and 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, October 2008, 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 and J. Gosling, "The real-time specification for java," *Computer*, vol. 33, no. 6, pp. 47–54, 2000.

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

[7] I. Lee, S. Davidson, and V. Wolfe, "RTC: language support for real-time concurrency," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS 91)*. San Antonio, TX, USA: IEEE, December 1991. [Online]. Available: http://repository.upenn.edu/cis_reports/368

[8] I. Lee and V. Gehlot, "Language Constructs for Distributed Real-Time Programming ," University of Pennsylvania, Tech. Rep., May 1985.

[9] J. W. S. Liu, K.-J. Lin, R. Bettati, D. Hull, and A. Yu, *Foundations of Dependable Computing*, ser. The International Series in Engineering and Computer Science. Springer US, November 1994, vol. 284, ch. Use of Imprecise Computation to Enhance Dependability of Real-Time Systems, pp. 157–182.

[10] D. Hull, W. chun Feng, and J. W. Liu, "Operating system support for imprecise computation," in *In Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities*, 1996, pp. 96–99.

[11] A. Quagli, D. Fontanelli, L. Greco, L. Palopoli, and A. Bicchi, "Designing real-time embedded controllers using the anytime computing paradigm," sept. 2009, pp. 1 –8.

[12] Intel, *IA-PC HPET (High Precision Event Timers) Specification (revision 1.0a)*, October 2004. [Online]. Available: http://www.intel.com/hardwaredesign/hpetspec_1.pdf

[13] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *Proceedings of the Eleventh Real-Time Linux Workshop*, Dresden, Germany, September 2009.

[14] F. Checconi, T. Cucinotta, D. Faggioli, and 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, June 2009.

[15] D. Faggioli, G. Lipari, and 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, July 2008.

[16] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA — adaptive quality of service architecture," *Software – Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.