Exception-Based Management of Timing Constraints Violations for Soft Real-Time Applications. *

Tommaso Cucinotta, **Dario Faggioli**
Scuola Superiore Sant'Anna, Pisa (Italy)
{t.cucinotta, d.faggioli}@sssup.it
Alessandro Evangelista
mail@evangelista.tv

Abstract

This paper presents an open-source library for the C language supporting the specification and management of timing constraints within embedded soft real-time applications. The library provides a set of well-designed C macros that allow developers to associate timing constraints to code segments, and to deal with their violations through the well-established practise of exception-based management.

After a brief overview of the requirements motivating the work, the exceptions library is presented. Then, the paper focuses on the specific macros that deal with the specification of deadline and execution-time constraints, with a few notes on how the library has been implemented.

Finally, a few experimental results are shown in order to discuss the features and limitations of this approach, with the current implementation (on Linux) that relies almost completely on POSIX-compliant system calls.

1 Introduction

General Purpose Operating Systems (GOPSes) are being enriched with more and more support for soft real-time applications, allowing for an easier development of applications with stringent timing requirements, such as multimedia and interactive ones. Still, one of the challenges for developers is how to specify timing constraints within the application, and how to properly design the application so as to respect them.

Furthermore, this kind of systems differ from traditional hard real-time ones, on a number of different points. First, a GPOS with a monolithic kernel cannot provide a precise scheduling of processes. Second, 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 general-purpose hardware architectures (that are optimised for average-case performance, penalising 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.

Therefore, in such context, timing constraints violations should be expected to occur at run-time, and developers need to deal with these events by embedding appropriate recovery logic. This usually involves the correct use of timers and signals, something not always immediate.

This paper presents a framework that allows for adoption of the well-known *exception-based management* approach for dealing with timing constraints violations in C applications. The framework makes it possible to handle these events similarly to how exceptions are managed in languages that support such programming paradigm, e.g., C++, Java and 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. Also, the proposed solution allows for an arbitrary nesting of timing constraints. In fact, in the expected typical scenario, it is foreseen to have one deadline constraint at the outermost level, and one or more nested WCET constraints.

To the best of the authors' knowledge, no similar mechanism has been previously presented for the C language, with the same completeness of the one presented here, with no need to modify the C compiler, and only relying on standard POSIX features.

Paper outline After a brief overview of the related work in Section 2, Section 3 identifies the main technical requirements that need to be supported by the mechanism, then

*The research leading to these results has been supported by the European Commission under grant agreement n.214777, in the context of the IRMOS Project. More information at: http://www.irmosproject.eu.

Section 5 describes the POSIX-based implementation realised for the Linux OS, finally a few experimental results are presented in Section 6 highlighting the impact of the Linux kernel configuration on the mechanism precision. Finally, conclusions are drawn in Section 7 along with directions for future work.

2 Related Work

The need for having more and more predictable timing behaviour 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 [3], a paradigm for designing hardware systems that provide a high degree of predictability of the software behaviour. 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 [9].

Coming to software approaches relying on the services of the Operating System (OS) and standard libraries, the POSIX.1b standard [5] 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 behaviour. 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) [1] 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 specialised exceptions in order to deal with timing constraints specification, enforcement and violation.

Also, the Ada 2005 language [2] 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 C 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 theorised 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.

3 Requirements Definition

The basic requirements that drive the work of this paper are presented here as drawn out by a simple example: a multimedia, component based application, designed as a single thread of execution[1] activated periodically or sporadically. For example, consider the *Video Decoder* application, whose behaviour is outlined in the UML Activity Diagram of Figure 1.

From a design level perspective, as *Video Decoder* will be co-scheduled with other applications, it would be highly desirable to characterise each component with such typical information: (1) WCET (or an appropriate statistic of execution time distribution); (2) relative or absolute deadline; (3) minimum period of activation. Also, it might be desirable that *Video Decoder* actually respects both the declared WCET and the deadline constraint, also in cases of overload, e.g., when a frame is particularly difficult to decode.

Now, assume that a *Frame Decoder* is used in the main loop of *Video Decoder*. Due to the in-place timing requirements, it would be useful to characterise *Frame Decoder* invocations with the WCET to be expected at run-time. In fact, as shown in Figure 1, such information, plus the WCETs of the *Stream Parser*, *Filtering* and *Visualization* components, sum up to the WCET of the *Video Decoder* itself. However, 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.

[1] For example, the `fflay` player, part of the widely used open-source `ffmpeg` project, is designed as a single threaded application.

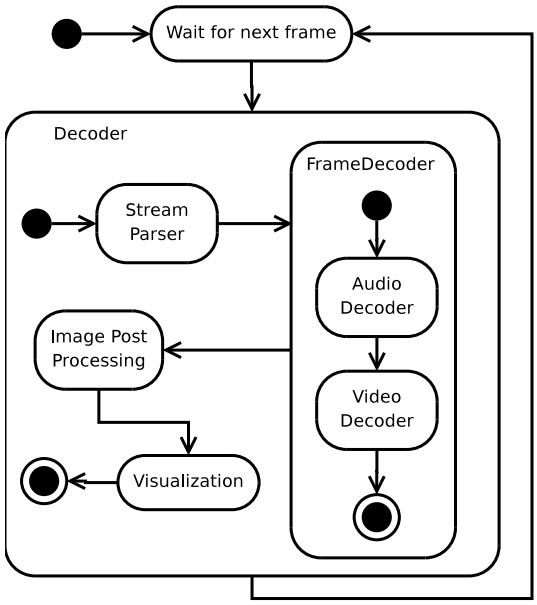

Figure 1. UML Activity diagram for the example video decoder thread.

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 a WCET estimation such that either: (1) the decoding operation terminates within the WCET limit, or (2) it is aborted.

The approach that is envisioned in this paper is aimed at simplifying design of such a complex software, and it is based on the adoption of an exception-based programming paradigm. A timing constraint violation is seen as an exceptional situation whose occurrence must be foreseen by the programmer, without necessarily subverting the flow of control that is normally realised.

However, it is clear that the possibility for the program to jump asynchronously to exception handling code segments is not something that may be seamlessly incorporated within an application. The latter should be designed so as to tolerate this kind of operation abortion, so as to not introduce memory leaks, and to properly cleanup any resources that might be associated with the aborting code segment. For example, a multimedia encoding/decoding library in which all the needed buffers are allocated at initialisation time, and the encoding/decoding functions only operate on these buffers (without any memory allocation nor mutex locks acquisition), the encoding/decoding functions may probably be safely asynchronously aborted. If this is not the case, then one should generally modify the code so as to catch the deadline exception at appropriate points in the code, so as to trigger the proper cleanup logic.

From the above sketched example, the following set of high-level requirements may be identified for the proposed mechanism.

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. In fact, this is a key feature for component based design of real-time applications. For example, not only* Video Decoder *should be associated with overall deadline and WCET constraints, but also* Frame Decoder *should be associated with its own WCET constraint;*

Requirement 6 *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 7 *the latency between the occurrence of the timing constraint violation and the activation of the application recovery code (from here on referred to as* handler activation latency*) should be known to the designer/developer, and it should be possibly negligible with respect to the task execution time;*

Requirement 8 *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;*

Requirement 9 *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 10 *the mechanism could be portable to as many Operating Systems as possible.*

4 Proposed approach

Here a mechanism complying with the above enumerated requirements is presented, with a focus on the programming paradigm and syntax. First, the generic framework for exceptions handling for the C language is presented. Then, the extensions for dealing with timing constraints violations are presented. Finally, for the sake of completeness, a few implementation details are discussed.

4.1 Exceptions for the C language

The framework for exception-management for the C language is distributed as part of the open-source project Open Macro Library (OML) [2], whose description is out of the scope of this paper. OML Exceptions supports hierarchical arrangement of exceptions, where all exceptions must derive from the common "type" `exception`. The syntax of such framework (from here on referred to as OML Exceptions) comprises the following macros:

`define_exception...extends`: this macro may be used to define new application-specific exceptions;

`try`: this macro delimits the code segment subject to exception handling;

`finally`: this macro identifies the code segment that will be executed both in case of exception and normal `try` termination;

`handle...handle_end`: these two macros must enclose the (optional) `when` clauses;

`when`: this macro identifies the code segment that is executed in reaction to each particular type of exception (or sub-type); the first matching `when` clause is the one that catches the exception; if no `when` clause catches the exception, then it is automatically re-thrown;

`re-throw`: this macro may be used to explicitly re-throw an exception from within a `when` clause, after it has been caught.

For example, Figure 2 shows an application that defines a custom exception, `ENotReady`, extending the `exception` basic "type", which is raised by using the `throw()` macro within the `foo()` function. Finally, the exception is caught by means of the `when()` macro within a `handle...end` block. The hierarchical arrangement of exceptions is useful for allowing a single `when` clause to specify a generic type and catch any exception descending from the specified one. As all exceptions derive from `exception`, a clause `when(exception)` may be used for catching any type of exception (similarly to the `catch(...)` syntax of C++).

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

```
define_exception(ENotReady) extends(exception);

void foo() {
  if (cond)
    throw(ENotReady);
}

void bar() {
  try {
    /* Potentially faulty code segment */
    foo();
  } finally {
    /* Clean-up code executed both on normal
     * termination _and_ on exception */
  }
  handle
    when (ENotReady) {
      /* Handle the ENotReady exception */
    }
    when (exception) {
      /* Handle any exception that is not
         of ENotReady type nor sub-type */
    }
  handle_end;
}
```

Figure 2. Example code segment

Notice that OML Exceptions is both process and thread safe. Moreover, it allows exception throwing/catching code to be nested. However, due to how the macros are defined, there are a set of limitations in the use of them. For example, within `try` blocks, it is forbidden to use any C mechanism that would cause an attempt to cross the block boundary, such as `return` statements, `goto` statements (and also `longjmp` calls) with destinations outside the block, and `continue` and `break` statements referring to iterative loops enclosing the `try` block. The full discussion of these aspects is omitted for the sake of brevity.

4.2 Timing Constraints Based Exceptions

OML Exceptions includes a support for notifying timing constraints violations by means of the following constructs[3]:

`try_within_abs`: this macro allows for starting a `try` block with an absolute deadline constraint;

`try_within_rel`: convenience macro macro useful for specifying a relative deadline constraint, however the effect of a relative deadline expiring is not distinguishable from the one of an absolute deadline expiring (see note at end of section);

`try_wcet`: this macro allows for starting a `try` block with a maximum allowed execution time (WCET);

[3] This support is available within the `dlexception` branch on the CVS repository of the OML project.

```
#include <oml_exceptions.h>

void Decoder() {
  next_deadline = current_time();
  for (;;) {
    next_deadline += period;

    /* absolute deadline constrained code */
    try_within_abs(next_deadline) {
      StreamParser();
      if (FrameDecoder() == 0)
        ImagePostProcessing();
      Visualization();
    }
    handle
      when (ex_deadline_violation) {
        /* e.g., re-use last decoded frame */
      }
    handle_end;
    /* Wait for next activation */
  }
}

int FrameDecoder() {
  int rv = 0;   /* Normal return code */

  try_wcet(12000) {
    DecodeAudioFrame();
    DecodeVideoFrame();
  }
  handle
    when (ex_wcet_violation) {
      /* Notify caller of incomplete decoding */
      rv = -1;
    }
  handle_end;

  return rv;
}
```

Figure 3. Example code of an video/audio player using the proposed mechanism.

ex_timing_constraint_violation: this is the common basic type for timing constraint violation exceptions; it may be used for the purpose of catching a generic timing constraint violation, without distinguishing between them;

ex_deadline_violation: this exception occurs as a result of a `try_within_rel` or `try_within_abs` segment not terminating within the specified relative or absolute deadline;

ex_wcet_violation: this exception occurs as a result of a `try_wcet` segment not terminating within the specified execution time constraint.

A simple example of how to use these macros is shown in Figure 3. The `Decoder` main body is a typical periodic thread where each activation has the next activation time as absolute deadline. Also, the `FrameDecoder()` function has a nested WCET constraint of $12ms$.

As a final remark, consider the particular erroneous usage of the framework shown in Figure 4. If the

```
try_within(10) {
  ...
  try_within(50) {
    ...
  }
  handle
    when (ex_deadline_violation) {
      /* handle violation of try_within(50)  *
       *  and not the one of try_within(10)  *
       *  which is the first that is raised. */
    }
  handle_end;
}
handle
  when (ex_deadline_violation) {
    /* handle the violation of try_within(10) *
     *  which has to be captured here and not *
     *  in the previous when clause.          */
  }
```

Figure 4. Typical example where relating each `try` clause with its `when` is needed.

`try_within` statements are used in different components, then such a situation may occur during development. OML Exceptions includes a special exception matching rule for the `when` clauses involving timing constraint exceptions: if the raised exception is associated to a `try` block that is external (in the run-time sense) to the current `try` block, then the exception is propagated instead of being stopped. In the example, this mechanism allows the outer handler to correctly detect the deadline violation, because it is not stopped by the nested handler.

OML Exceptions complies with all of the requirements introduced in Section 3, with the few notes outlined in the following section.

5 Implementation

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

5.1 Time-Scoped Segment Implementation

OML Exceptions has been realized by means of the POSIX `sigsetjmp()` and `siglongjmp()` 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. 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 POSIX signal with the property of being 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.

5.2 Deadline and WCET Signal Handling

In case one (or more) specified constraint is violated, a signal has to be sent to the *correct* thread, in order to fulfil Requirement 4. However, signal delivery to a specific thread is not covered by POSIX. In fact, when a signal is sent, it reaches the whole process, and it is not possible to determine in advance which thread will receive and handle it. Therefore, the standard suggests to have one only thread receiving the signal, and all the others ignoring it, so that the receiving thread may notify the correct thread by means of other inter-thread synchronization primitives. 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, OML Exceptions is implemented by using this extension, which, at the cost of sacrificing Requirements 10, allows for a much more efficient implementation of the mechanism on Linux (see also Figure 5). However, a version of OML Exceptions perfectly compliant with POSIX is being implemented as well, so that the framework will be capable of choosing the best implementation at compile-time.

5.3 Benchmarking Operational Mode

In order to cope with requirement 9, a compile-time switch has been provided 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.

5.4 Non-Interruptible Code Sections

Requirement 8 is achieved by providing two additional macros, oml_within_disable and oml_within_enable, within which developers may enclose *atomic* code segments that cannot be asynchronously interrupted by a timing constraint violation. These two macros simply disable and enable, respectively, delivery of the time constraint violation real-time signals. If a signal occurs in the middle of such a protected code region, then it is enqueued by the OS, and delivered immediately at the end of the section.

5.5 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 Operating System. This is true also for the preliminary implementation on Linux, and thus a description of how timers and task execution time accounting are dealt with in the Linux kernel follows.

From mainstream kernel version 2.6.21, the kernel has been enriched by the 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. Therefore, if a Linux process or thread posts a timer to fire at a certain instant, it could expect to be woken up quite close to that point in time.

Despite this, there still exist Linux kernel subsystems depending on the periodic system tick. With respect to the presented work, the most relevant one is the time accounting mechanism, i.e., how the system tracks how much a thread is executing. In fact, this is done by the kernel at each occurrence of the following events:

- at each periodic system tick;
- at each task scheduling event, i.e., enqueue, dequeue or preemption.

Thus, the time accounting resolution is limited by the system tick frequency, which can be configured by the user at kernel compile time. Typical values are 100, 250 and 1000 Hertz, which means, respectively, 10, 4 and 1 millisecond resolution. This is also important, since the CPU-clock based timers used to implement the WCET timing constraints are not based on high resolution timers, and rely only on standard Linux accounting.

6 Experimental Evaluation

The proposed mechanism is effective and useful only if the latency between the occurrence of the timing constraint violation and the activation of application recovery logic (handler activation latency) is relatively small (with respect to the job execution times of the application), and if its value is known to the designer (Requirement 7).

In Figure 5 the various components contributing to the total amount of latency introduced between an actual con-

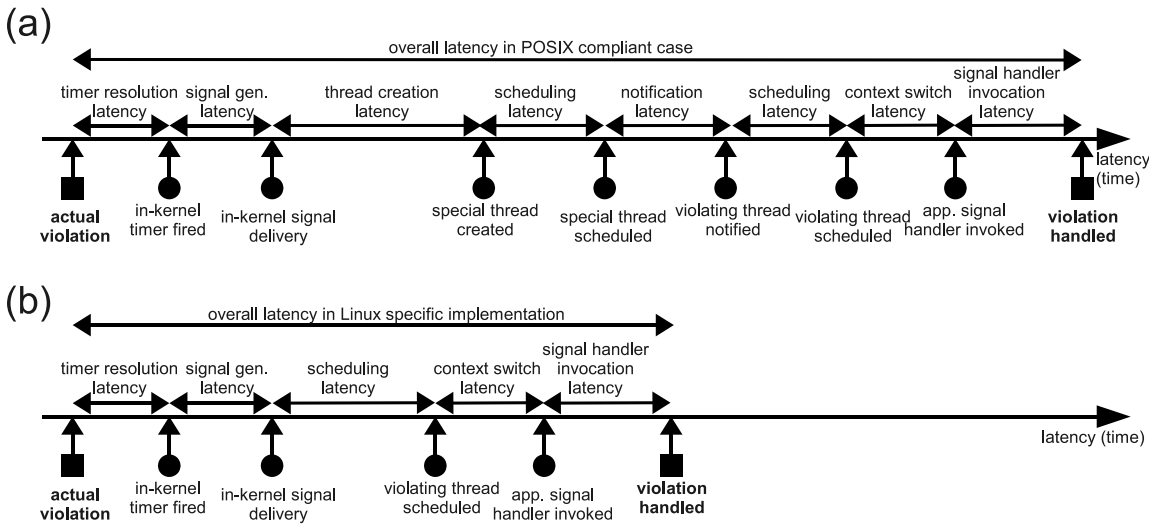

Figure 5. Various components contributing to the handler activation latency for the POSIX compliant (a) and Linux specific (b) implementations.

straint violation and its notification to the application are shown.

The handler activation latency has been measured by means of two experiments, made on the Linux Operating System (OS), that also highlight how the latency is affected by the kernel configuration.

6.1 Experiments Set-Up

A simple test application, built as a single thread of execution, has been used in both experiments, and no other applications have been launched concurrently. This way, the measurements were not affected by other components of the handler activation latency, such as the scheduling latency (the latter should be addressed by adopting a real-time design strategy). Thus, a task τ with WCET, relative deadline and period equal to $(C, D, T) = (50, 50, 100)\ msec$ is used, and run for 1000 consecutive instances. Experiments have been performed on commonly available desktop PC hardware, with 3.0 GHz Intel CPU and 2 GB of RAM. Debian GNU/Linux version 4.0, with hand-tailored kernel version 2.6.28 was the Operating System used. Kernel configuration includes the high-resolution timer subsystem, with support for high precision hardware timing sources. The one used by the system while running the experiments was the HPET [6].

In the first experiment, the latency of a deadline violation is measured 1000 times. This is done by forcing τ to execute more than $50\ msec$ inside a `try_within` 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.
In the second experiment, the task again executes more than $50\ msec$, this time from inside a `try_wcet` block, so to cause a WCET violation and measure its latency as well.

Both experiments have been run on three different configurations of the Linux kernel, i.e., with 100, 250 and 1000 Hertz as the periodic tick frequency, to study if and how this affects the latency.

Common statistics on the measured latency figures have been computed for both experiments on each configuration, and the corresponding cumulative distribution functions (CDF) are reported below. Minimum achieved latency values have not been 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, thus they provide few information about the performance of the mechanism.

6.2 Experiments Results

Results of the experiments are shown in Tables 1 and 2 and in Figures 6 and 7. They show that the latency of the notification of a deadline violation is both small and independent from the system tick frequency. In fact, values in Table 1 are comparable, and the three CDF in Figure 6 are completely superimposed. The measured latency values are in the order of the μs, what constitutes a more than acceptable performance.

Situation is different for WCET violation results. In fact,

	max	mean	std. dev.
HZ=100	28610	1724.418	1187.854
HZ=250	17202	1595.095	711.1304
HZ=1000	33394	1602.544	1023.255

Table 1. Deadline latency in ns

	max	mean	std. dev.
HZ=100	18727747	5748948.344	4474771.769
HZ=250	4423164	1233955.255	844593.486
HZ=1000	1999752	522228.673	390837.341

Table 2. WCET latency in ns

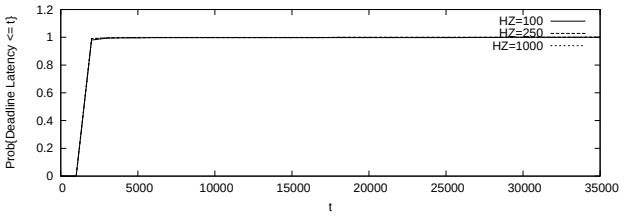

Figure 6. Cumulative Distribution Function of the deadline violation latencies. Time on x-axis is in ns.

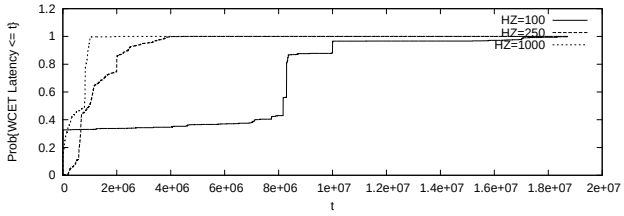

Figure 7. Cumulative Distribution Function of the WCET violation latencies. Time on x-axis in ns

as shown by both the values in Table 2 and the three CDF of Figure 7, the precision achieved in case of a WCET violation is tightly coupled with the system tick frequency HZ. Table 2 also shows how the mean WCET latency is close to $\frac{HZ}{2}$, which is exactly what was expected. As it can be easily seen, for the mechanism to be useful, the value of $HZ = 1000$ is strongly suggested.

7 Conclusions

In this paper, an open-source library has been presented for the management of timing constraints violations according to the well-known exception-based paradigm. This constitutes a valuable support for developers of embedded soft real-time applications.

A set of basic requirements have been identified, and a mechanism has been presented fulfilling them. The result is a framework designed as a set of macros for the C language, integrated in an open-source project that enables exception management.

Thanks to the proposed framework, developers may focus on one hand on the main application flow of control, which will be executed most of the times (or at least with a high probability, if the system is properly designed). On the other hand, the framework allows to catch dynamically possible violations of the timing constraints associated to critical parts of the code, due either to particular input data, or to the non-perfectly predictable behavior of applications on a soft real-time platform, such as Linux. The code in compensation or recovery of this is provided in the form of an exception handler.

An implementation of the proposed mechanism has been presented for the Linux Operating System, based on standard POSIX-class primitives. A few experimental results have been presented highlighting latencies that the applications using the framework experience in the activation of the exception management code, compared to the ideal time of fire of the exception, and the limitations on the precision of the current solution have been discussed.

Of course, such a framework should be used in combination with a real-time design methodology, and advanced real-time scheduling features, available in various forms for the Linux kernel, such as the POSIX Sporadic Server [4] or the Adaptive QoS Architecture for Linux [10].

Concerning possible directions for future work, a kernel-level mechanism is being investigated for the Linux OS, leading to a reduction in the handler activation latency. Furthermore, a more ambitious macro-based framework for C is under design that will enrich OML with generic constructs for threads management, synchronization and real-time scheduling.

References

[1] G. Bollella and J. Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

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

[3] 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)*, pages 264–265, New York, NY, USA, 2007. ACM.

[4] 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 Operat-*

ing Systems Platforms for Embedded Real-Time Applications (OSPERT 2008), Prague, Czech Republic, July 2008.

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

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

[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, December 1991. IEEE *** FIXME ***.

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

[9] 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)*, pages 137–146, Atlanta, Georgia, United States, October 2008.

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