Exception-Based Management of Timing Constraints Violations for Soft Real-Time Applications [1]

T. Cucinotta, **D. Faggioli**
ReTiS Lab, **Scuola Superiore Sant'Anna**, CEIIC
via G. Moruzzi 1,56124 Pisa (Italy)
{t.cucinotta, d.faggioli}@sssup.it
A. Evangelista
mail@evangelista.tv

30, June 2009

[1] This work has been supported in part by the EU Commission within the context of the IRMOS FP7/2008/ICT/214777 European Project.

Motivation for This Work

General Purpose Operating Systems are continuously being enriched with real-time capabilities.

However, especially on general purpose computing platforms:

- hardware is optimized for average performance,
- execution times may heavily vary between jobs.
- knowledge of detailed timing of applications is limited,

Some timing constraints violations should be expected, thus something is needed:

- to specify timing constraints inside the application,
- to help developers in design and timing overrun handling.

Motivation for This Work

General Purpose Operating Systems are continuously being enriched with real-time capabilities.

However, especially on general purpose computing platforms:

- hardware is optimized for average performance,
- execution times may heavily vary between jobs.
- knowledge of detailed timing of applications is limited,

Some timing constraints violations should be expected, thus something is needed:

- to specify timing constraints inside the application,
- to help developers in design and timing overrun handling.

Motivation for This Work

General Purpose Operating Systems are continuously being enriched with real-time capabilities.

However, especially on general purpose computing platforms:

- hardware is optimized for average performance,
- execution times may heavily vary between jobs.
- knowledge of detailed timing of applications is limited,

Some timing constraints violations should be expected, thus something is needed:

- to specify timing constraints inside the application,
- to help developers in design and timing overrun handling.

Motivation for This Work

General Purpose Operating Systems are continuously being enriched with real-time capabilities.

However, especially on general purpose computing platforms:

- hardware is optimized for average performance,
- execution times may heavily vary between jobs.
- knowledge of detailed timing of applications is limited,

Some timing constraints violations should be expected, thus something is needed:

- to specify timing constraints inside the application,
- to help developers in design and timing overrun handling.

Motivation for This Work

General Purpose Operating Systems are continuously being enriched with real-time capabilities.

However, especially on general purpose computing platforms:

- hardware is optimized for average performance,
- execution times may heavily vary between jobs.
- knowledge of detailed timing of applications is limited,

Some timing constraints violations should be expected, thus something is needed:

- to specify timing constraints inside the application,
- to help developers in design and timing overrun handling.

Motivation for This Work

General Purpose Operating Systems are continuously being enriched with real-time capabilities.

However, especially on general purpose computing platforms:

- hardware is optimized for average performance,
- execution times may heavily vary between jobs.
- knowledge of detailed timing of applications is limited,

Some timing constraints violations should be expected, thus something is needed:

- to specify timing constraints inside the application,
- to help developers in design and timing overrun handling.

Motivation for This Work

General Purpose Operating Systems are continuously being enriched with real-time capabilities.

However, especially on general purpose computing platforms:

- hardware is optimized for average performance,
- execution times may heavily vary between jobs.
- knowledge of detailed timing of applications is limited,

Some timing constraints violations should be expected, thus something is needed:

- to specify timing constraints inside the application,
- to help developers in design and timing overrun handling.

Motivation for This Work

General Purpose Operating Systems are continuously being enriched with real-time capabilities.

However, especially on general purpose computing platforms:

- hardware is optimized for average performance,
- execution times may heavily vary between jobs.
- knowledge of detailed timing of applications is limited,

Some timing constraints violations should be expected, thus something is needed:

- to specify timing constraints inside the application,
- to help developers in design and timing overrun handling.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Contribution of This Work

Two kind of of timing constraints:

- *deadline constraints*, whenever a software component needs to complete within a certain (wall-clock) time,
- *WCET constraints*, whenever a software component needs to not exceed an predetermined execution time.

Exception-based management approach:

- mechanism similar to exception management in C++, Java or Ada,
- available for the C language (widely used in embedded systems):
  - implemented by macros, i.e., no compiler modification needed,
  - usable directly from the program, i.e., no external component needed,
- support for arbitrary nesting.

Related Work

Need for predictable timing behavior of system components is paramount for real-time.

Existing –programming language level– solutions:

- *RTSJ*: specialized exceptions to deal with timing specification and enforcement,
- *ADA 2005*: Asynchronous Transfer of Control, usable in case of deadline and/or WCET violations,
- *RTC*: introduces new syntactic “real-time constructs” into C, but requires non-standard/non-existent compiler.

Nothing for C programs and standard C compiler exists.

Related Work

Need for predictable timing behavior of system components is paramount for real-time.

Existing –programming language level– solutions:

- *RTSJ*: specialized exceptions to deal with timing specification and enforcement,
- *ADA 2005*: Asynchronous Transfer of Control, usable in case of deadline and/or WCET violations,
- *RTC*: introduces new syntactic “real-time constructs” into C, but requires non-standard/non-existent compiler.

Nothing for C programs and standard C compiler exists.

Related Work

Need for predictable timing behavior of system components is paramount for real-time.

Existing –programming language level– solutions:

- *RTSJ*: specialized exceptions to deal with timing specification and enforcement,
- *ADA 2005*: Asynchronous Transfer of Control, usable in case of deadline and/or WCET violations,
- *RTC*: introduces new syntactic “real-time constructs” into C, but requires non-standard/non-existent compiler.

Nothing for C programs and standard C compiler exists.

Related Work

Need for predictable timing behavior of system components is paramount for real-time.

Existing –programming language level– solutions:

- *RTSJ*: specialized exceptions to deal with timing specification and enforcement,
- *ADA 2005*: Asynchronous Transfer of Control, usable in case of deadline and/or WCET violations,
- *RTC*: introduces new syntactic "real-time constructs" into C, but requires non-standard/non-existent compiler.

Nothing for C programs and standard C compiler exists.

Related Work

Need for predictable timing behavior of system components is paramount for real-time.

Existing –programming language level– solutions:

- *RTSJ*: specialized exceptions to deal with timing specification and enforcement,
- *ADA 2005*: Asynchronous Transfer of Control, usable in case of deadline and/or WCET violations,
- *RTC*: introduces new syntactic "real-time constructs" into C, but requires non-standard/non-existent compiler.

Nothing for C programs and standard C compiler exists.

Related Work

Need for predictable timing behavior of system components is paramount for real-time.

Existing –programming language level– solutions:

- *RTSJ*: specialized exceptions to deal with timing specification and enforcement,
- *ADA 2005*: Asynchronous Transfer of Control, usable in case of deadline and/or WCET violations,
- *RTC*: introduces new syntactic “real-time constructs” into C, but requires non-standard/non-existent compiler.

Nothing for C programs and standard C compiler exists.

A Simple Example

Component based multimedia application.

A Simple Example

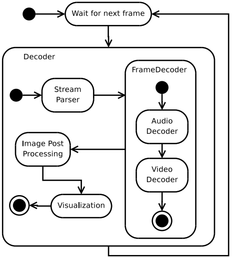
Component based multimedia application.

(Sub)Components may came from libraries and/or third party software packages.

A Simple Example

Component based multimedia application.

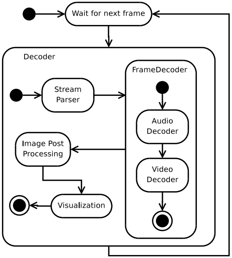
(Sub)Components may came from libraries and/or third party software packages.



A Simple Example

Component based multimedia application.

(Sub)Components may came from libraries and/or third party software packages.

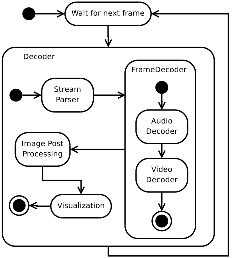


Highly desirable design-time features:

A Simple Example

Component based multimedia application.

(Sub)Components may came from libraries and/or third party software packages.



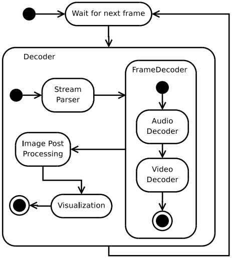
Highly desirable design-time features:

- deadline indication and enforcement for each (sub)component.

A Simple Example

Component based multimedia application.

(Sub)Components may came from libraries and/or third party software packages.



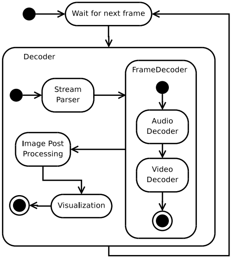
Highly desirable design-time features:

- deadline indication and enforcement for each (sub)component.
- WCET enforcement for each (sub)component:

A Simple Example

Component based multimedia application.

(Sub)Components may came from libraries and/or third party software packages.



Highly desirable design-time features:

- deadline indication and enforcement for each (sub)component.
- WCET enforcement for each (sub)component:

$$WCET_{Prsr} + WCET_{FrmDec} + WCET_{PostProc} + WCET_{Vis} = WCET_{Decoder}$$

Requirements

A mechanism of such kind should:

- support relative and absolute deadline constraint;
- support WCET constraint;
- support generic (timing) or specific recovery logic triggering on violation of those two;
- support both processes and threads;
- support nesting of timing constraints;
- support benchmarking timing behavior of components;
- support being "switched off" for some code segments.

Requirements

A mechanism of such kind should:

- support relative and absolute deadline constraint;
- support WCET constraint;
- support generic (timing) or specific recovery logic triggering on violation of those two;
- support both processes and threads;
- support nesting of timing constraints;
- support benchmarking timing behavior of components;
- support being "switched off" for some code segments.

Requirements

A mechanism of such kind should:

- support relative and absolute deadline constraint;
- support WCET constraint;
- support generic (timing) or specific recovery logic triggering on violation of those two;
- support both processes and threads;
- support nesting of timing constraints;
- support benchmarking timing behavior of components;
- support being "switched off" for some code segments.

Requirements

A mechanism of such kind should:

- support relative and absolute deadline constraint;
- support WCET constraint;
- support generic (timing) or specific recovery logic triggering on violation of those two;
- support both processes and threads;
- support nesting of timing constraints;
- support benchmarking timing behavior of components;
- support being "switched off" for some code segments.

Requirements

A mechanism of such kind should:

- support relative and absolute deadline constraint;
- support WCET constraint;
- support generic (timing) or specific recovery logic triggering on violation of those two;
- support both processes and threads;
- support nesting of timing constraints;
- support benchmarking timing behavior of components;
- support being "switched off" for some code segments.

Requirements

A mechanism of such kind should:

- support relative and absolute deadline constraint;
- support WCET constraint;
- support generic (timing) or specific recovery logic triggering on violation of those two;
- support both processes and threads;
- support nesting of timing constraints;
- support benchmarking timing behavior of components;
- support being "switched off" for some code segments.

Requirements

A mechanism of such kind should:

- support relative and absolute deadline constraint;
- support WCET constraint;
- support generic (timing) or specific recovery logic triggering on violation of those two;
- support both processes and threads;
- support nesting of timing constraints;
- support benchmarking timing behavior of components;
- support being "switched off" for some code segments.

Requirements

A mechanism of such kind should:

- support relative and absolute deadline constraint;
- support WCET constraint;
- support generic (timing) or specific recovery logic triggering on violation of those two;
- support both processes and threads;
- support nesting of timing constraints;
- support benchmarking timing behavior of components;
- support being "switched off" for some code segments.

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (`gcc`);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (gcc);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (`gcc`);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (`gcc`);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- **`try`:** code segment subject to exception management;
- **`when`:** code segment executed in reaction to an –the first matching– exception;
- **`handle...end`:** code segments for one or more `when` clauses;
- **`finally`:** code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (`gcc`);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (gcc);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (gcc);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

`try`: code segment subject to exception management;

`when`: code segment executed in reaction to an –the first matching– exception;

`handle...end`: code segments for one or more `when` clauses;

`finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (gcc);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (gcc);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (`gcc`);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exceptions for the C language

Open Macro Library (author's former project, `http://oml.sourceforge.net`):

- `try`: code segment subject to exception management;
- `when`: code segment executed in reaction to an –the first matching– exception;
- `handle...end`: code segments for one or more `when` clauses;
- `finally`: code segment executed after the `try`, either any exception fired or not;

Implemented by means of:

- macros only, i.e., works with standard compilers (gcc);
- POSIX long jumps (`sigsetjmp()`, `siglongjmp()`).

Moreover:

- it is process and thread safe;
- it allows nested exception throwing/catching (up to impl. defined level).

Exception Example

```
define_exception(ENotReady) extends(EException);
void foo()
{
  if (cond)
    throw(ENotReady);
}
void bar()
{
  try { /* Potentially faulty code segment */
    f();
  } finally { /* Clean-up code */
  }
  handle
    when (ENotReady) { /* Handle the ENotReady exception */
    }
    when (EException) { /* Handle any other exception */
    }
  end;
}
```

Additions to OML:

`try_within`: code segment with relative deadline constraint;

`try_within_abs`: code segment with absolute deadline constraint;

`try_wcet`: code segment with maximum allowed execution time (WCET);

`ETimingConstraintViolation`: basic type for timing constraint exceptions;

`EDeadlineViolation`: occurring if `try_within` or `try_within_abs` segment do not make their deadlines;

`EWCETViolation`: occurring if `try_wcet` executes more than how it specified.

Timing Exceptions

Additions to OML:

`try_within`: code segment with relative deadline constraint;

`try_within_abs`: code segment with absolute deadline constraint;

`try_wcet`: code segment with maximum allowed execution time (WCET);

`ETimingConstraintViolation`: basic type for timing constraint exceptions;

`EDeadlineViolation`: occurring if `try_within` or `try_within_abs` segment do not make their deadlines;

`EWCETViolation`: occurring if `try_wcet` executes more than how it specified.

Timing Exceptions

Additions to OML:

try_within: code segment with relative deadline constraint;

try_within_abs: code segment with absolute deadline constraint;

try_wcet: code segment with maximum allowed execution time (WCET);

ETimingConstraintViolation: basic type for timing constraint exceptions;

EDeadlineViolation: occurring if try_within or try_within_abs segment do not make their deadlines;

EWCETViolation: occurring if try_wcet executes more than how it specified.

Timing Exceptions

Additions to OML:

`try_within`: code segment with relative deadline constraint;

`try_within_abs`: code segment with absolute deadline constraint;

`try_wcet`: code segment with maximum allowed execution time (WCET);

`ETimingConstraintViolation`: basic type for timing constraint exceptions;

`EDeadlineViolation`: occurring if `try_within` or `try_within_abs` segment do not make their deadlines;

`EWCETViolation`: occurring if `try_wcet` executes more than how it specified.

Timing Exceptions

Additions to OML:

try_within: code segment with relative deadline constraint;

try_within_abs: code segment with absolute deadline constraint;

try_wcet: code segment with maximum allowed execution time (WCET);

ETimingConstraintViolation: basic type for timing constraint exceptions;

EDeadlineViolation: occurring if try_within or try_within_abs segment do not make their deadlines;

EWCETViolation: occurring if try_wcet executes more than how it specified.

Timing Exceptions

Additions to OML:

try_within: code segment with relative deadline constraint;

try_within_abs: code segment with absolute deadline constraint;

try_wcet: code segment with maximum allowed execution time (WCET);

ETimingConstraintViolation: basic type for timing constraint exceptions;

EDeadlineViolation: occurring if try_within or try_within_abs segment do not make their deadlines;

EWCETViolation: occurring if try_wcet executes more than how it specified.

Timing Exceptions

Additions to OML:

`try_within`: code segment with relative deadline constraint;

`try_within_abs`: code segment with absolute deadline constraint;

`try_wcet`: code segment with maximum allowed execution time (WCET);

`ETimingConstraintViolation`: basic type for timing constraint exceptions;

`EDeadlineViolation`: occurring if `try_within` or `try_within_abs` segment do not make their deadlines;

`EWCETViolation`: occurring if `try_wcet` executes more than how it specified.

Timing Exception Example (I)

```
#include <oml_exceptions.h>

void Decoder {
  next_dl = now;
  for (;;) {
    next_dl = next_dl + period;

    try_within_abs(next_dl) {
      StreamParser();
      if (FrameDecoder() == 0)
        ImagePostProcessing();
      Visualization();
    }
    handle
      when (EDeadlineViolation) {
        /* e.g., re-use last decoded frame */
      }
    end;
  }
}
```

Timing Exception Example (II)

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

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

  return rv;
}
```

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference–– for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference–– for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference–– for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference–– for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference— for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference— for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference— for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference— for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Something About Implementation

Implementation for POSIX standard systems:

- `sigsetjmp()` and `siglongjmp()` for the base mechanism of exceptions;
- interval timers (`itimers`) with:
  - `CLOCK_MONOTONIC` –non decreasing time reference– for deadline enforcement;
  - `CLOCK_THREAD_CPUTIME_ID` –thread execution time reference— for wcet enforcement.
- real-time signal delivery to "faulting threads" on timer firing:
  - signals can be temporary blocked, but no delivery is lost;
  - signals are delivered in the order they have been sent;

Implementation is portable to any really OS providing support for POSIX real-time extensions.

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer` based timers resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

hrtimer based timers resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer` based timers resolution is:
- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:
- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer based timers` resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer based timers` resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer` based timers resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer` based timers resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer based timers` resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer` based timers resolution is:
- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:
- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer` based timers resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Violation-Notification Precision and Latency

Maximum achievable precision is subject to time-keeping precision of the underlying OS. On Linux (at least since 2.6.21 kernels) we have:

- `hrtimers` for `CLOCK_MONOTONIC` based `itimers`
- accounting based for `CLOCK_THREAD_CPUTIME_ID` based `itimers`.

Which means:

`hrtimer` based timers resolution is:

- not related to periodic tick frequency;
- only dependant on underlying hardware (e.g., availability of TSC, etc.);

accounting based timers resolution is dependant on accounting events, i.e. at:

- each periodic tick (every 10, 4 or 1 *msec*);
- each scheduling event (en/de-queue, preemption, etc.)

Benchmarking and Non-Interruptible Modes

Benchmarking operational mode: Compile-time switch are provided for gathering information on the duration of `try...handle` segments, instead of constraint enforcement.

Non-Interruptible code sections: Simply temporary blocking signal delivery is all it is needed to protect a code segment from being interrupted by a constraint violation notification.

Benchmarking operational mode: Compile-time switch are provided for gathering information on the duration of `try...handle` segments, instead of constraint enforcement.

Non-Interruptible code sections: Simply temporary blocking signal delivery is all it is needed to protect a code segment from being interrupted by a constraint violation notification.

Benchmarking and Non-Interruptible Modes

Benchmarking operational mode: Compile-time switch are provided for gathering information on the duration of `try...handle` segments, instead of constraint enforcement.

Non-Interruptible code sections: Simply temporary blocking signal delivery is all it is needed to protect a code segment from being interrupted by a constraint violation notification.

Sending the signal to the *Correct* Thread (I)

Direct signal delivery to a *specific* thread is not covered by POSIX:

- signals reach a whole process,
- they may be delivered to anyone of the threads that does not block it,
- *impossible* to know in advance which thread will receive and handle it.

Two solutions are possible:

POSIX compliant: have a special thread responsible for getting and distributing it to its legitimate recipient.

Linux specific: *non-standard* direct signal delivery to a specific thread is possible.

Sending the signal to the *Correct* Thread (I)

Direct signal delivery to a *specific* thread is not covered by POSIX:

- signals reach a whole process,
- they may be delivered to anyone of the threads that does not block it,
- *impossible* to know in advance which thread will receive and handle it.

Two solutions are possible:

POSIX compliant: have a special thread responsible for getting and distributing it to its legitimate recipient.

Linux specific: *non-standard* direct signal delivery to a specific thread is possible.

Sending the signal to the *Correct* Thread (I)

Direct signal delivery to a *specific* thread is not covered by POSIX:

- signals reach a whole process,
- they may be delivered to anyone of the threads that does not block it,
- *impossible* to know in advance which thread will receive and handle it.

Two solutions are possible:

POSIX compliant: have a special thread responsible for getting and distributing it to its legitimate recipient.

Linux specific: *non-standard* direct signal delivery to a specific thread is possible.

Sending the signal to the *Correct* Thread (I)

Direct signal delivery to a *specific* thread is not covered by POSIX:

- signals reach a whole process,
- they may be delivered to anyone of the threads that does not block it,
- *impossible* to know in advance which thread will receive and handle it.

Two solutions are possible:

POSIX compliant: have a special thread responsible for getting and distributing it to its legitimate recipient.

Linux specific: *non-standard* direct signal delivery to a specific thread is possible.

Sending the signal to the *Correct* Thread (I)

Direct signal delivery to a *specific* thread is not covered by POSIX:

- signals reach a whole process,
- they may be delivered to anyone of the threads that does not block it,
- *impossible* to know in advance which thread will receive and handle it.

Two solutions are possible:

POSIX compliant: have a special thread responsible for getting and distributing it to its legitimate recipient.

Linux specific: *non-standard* direct signal delivery to a specific thread is possible.

Sending the signal to the *Correct* Thread (I)

Direct signal delivery to a *specific* thread is not covered by POSIX:

- signals reach a whole process,
- they may be delivered to anyone of the threads that does not block it,
- *impossible* to know in advance which thread will receive and handle it.

Two solutions are possible:

POSIX compliant: have a special thread responsible for getting and distributing it to its legitimate recipient.

Linux specific: *non-standard* direct signal delivery to a specific thread is possible.

Sending the signal to the *Correct* Thread (I)

Direct signal delivery to a *specific* thread is not covered by POSIX:

- signals reach a whole process,
- they may be delivered to anyone of the threads that does not block it,
- *impossible* to know in advance which thread will receive and handle it.

Two solutions are possible:

POSIX compliant: have a special thread responsible for getting and distributing it to its legitimate recipient.

Linux specific: *non-standard* direct signal delivery to a specific thread is possible.

Sending the signal to the *Correct* Thread (II)

POSIX recommended mechanism:
creating a special handling thread at each signal delivery.

Advantages:

- standard compliance, i.e. portability.

Drawbacks:

- constraint violations are delayed by thread creation latency;
- constraint violations entail creation and rapid destruction of quite a number of threads.
- constraint violations may result in up to 3 context switches by itself to be accomplished;
- constraint violations notification involves scheduling, i.e. its latency is dependant on system status actual load.

Implementation is widely portable, but latency between violation and its notification may be not-negligible.

Sending the signal to the *Correct* Thread (II)

POSIX recommended mechanism:
creating a special handling thread at each signal delivery.

Advantages:

- standard compliance, i.e. portability.

Drawbacks:

- constraint violations are delayed by thread creation latency;
- constraint violations entail creation and rapid destruction of quite a number of threads.
- constraint violations may result in up to 3 context switches by itself to be accomplished;
- constraint violations notification involves scheduling, i.e. its latency is dependant on system status actual load.

Implementation is widely portable, but latency between violation and its notification may be not-negligible.

Sending the signal to the *Correct* Thread (II)

POSIX recommended mechanism:
creating a special handling thread at each signal delivery.

Advantages:

- standard compliance, i.e. portability.

Drawbacks:

- constraint violations are delayed by thread creation latency;
- constraint violations entail creation and rapid destruction of quite a number of threads.
- constraint violations may result in up to 3 context switches by itself to be accomplished;
- constraint violations notification involves scheduling, i.e. its latency is dependant on system status actual load.

Implementation is widely portable, but latency between violation and its notification may be not-negligible.

Sending the signal to the *Correct* Thread (II)

POSIX recommended mechanism:
creating a special handling thread at each signal delivery.

Advantages:

- standard compliance, i.e. portability.

Drawbacks:

- constraint violations are delayed by thread creation latency;
- constraint violations entail creation and rapid destruction of quite a number of threads.
- constraint violations may result in up to 3 context switches by itself to be accomplished;
- constraint violations notification involves scheduling, i.e. its latency is dependant on system status actual load.

Implementation is widely portable, but latency between violation and its notification may be not-negligible.

Sending the signal to the *Correct* Thread (II)

POSIX recommended mechanism:
creating a special handling thread at each signal delivery.

Advantages:

- standard compliance, i.e. portability.

Drawbacks:

- constraint violations are delayed by thread creation latency;
- constraint violations entail creation and rapid destruction of quite a number of threads.
- constraint violations may result in up to 3 context switches by itself to be accomplished;
- constraint violations notification involves scheduling, i.e. its latency is dependant on system status actual load.

Implementation is widely portable, but latency between violation and its notification may be not-negligible.

Sending the signal to the *Correct* Thread (II)

POSIX recommended mechanism:
creating a special handling thread at each signal delivery.

Advantages:

- standard compliance, i.e. portability.

Drawbacks:

- constraint violations are delayed by thread creation latency;
- constraint violations entail creation and rapid destruction of quite a number of threads.
- constraint violations may result in up to 3 context switches by itself to be accomplished;
- constraint violations notification involves scheduling, i.e. its latency is dependant on system status actual load.

Implementation is widely portable, but latency between violation and its notification may be not-negligible.

Sending the signal to the *Correct* Thread (III)

Linux specific mechanism:
Exploit non-standard –enabled on Linux– possibility of specific thread `itimer` signal delivery.

Advantages:

- time between violation and notification is as tight as possible.

Drawbacks:

- exploits a non-standard Linux specific extension.

Constraint violation latency is kept low, but the resulting code only runs on Linux.

In this *preliminary* work, only this solution is being analyzed.

Sending the signal to the *Correct* Thread (III)

Linux specific mechanism:
Exploit non-standard –enabled on Linux– possibility of specific thread `itimer` signal delivery.

Advantages:

- time between violation and notification is as tight as possible.

Drawbacks:

- exploits a non-standard Linux specific extension.

Constraint violation latency is kept low, but the resulting code only runs on Linux.

In this *preliminary* work, only this solution is being analyzed.

Sending the signal to the *Correct* Thread (III)

Linux specific mechanism:
Exploit non-standard –enabled on Linux– possibility of specific thread `itimer` signal delivery.

Advantages:

- time between violation and notification is as tight as possible.

Drawbacks:

- exploits a non-standard Linux specific extension.

Constraint violation latency is kept low, but the resulting code only runs on Linux.

In this *preliminary* work, only this solution is being analyzed.

Sending the signal to the *Correct* Thread (III)

Linux specific mechanism:
Exploit non-standard –enabled on Linux– possibility of specific thread `itimer` signal delivery.

Advantages:

- time between violation and notification is as tight as possible.

Drawbacks:

- exploits a non-standard Linux specific extension.

Constraint violation latency is kept low, but the resulting code only runs on Linux.

In this *preliminary* work, only this solution is being analyzed.

Sending the signal to the *Correct* Thread (III)

Linux specific mechanism:
Exploit non-standard –enabled on Linux– possibility of specific thread `itimer` signal delivery.

Advantages:

- time between violation and notification is as tight as possible.

Drawbacks:

- exploits a non-standard Linux specific extension.

Constraint violation latency is kept low, but the resulting code only runs on Linux.

In this *preliminary* work, only this solution is being analyzed.

Sending the signal to the *Correct* Thread (III)

Linux specific mechanism:
Exploit non-standard –enabled on Linux– possibility of specific thread `itimer` signal delivery.

Advantages:

- time between violation and notification is as tight as possible.

Drawbacks:

- exploits a non-standard Linux specific extension.

Constraint violation latency is kept low, but the resulting code only runs on Linux.

In this *preliminary* work, only this solution is being analyzed.

Sending the signal to the *Correct* Thread (III)

Linux specific mechanism:
Exploit non-standard –enabled on Linux– possibility of specific thread `itimer` signal delivery.

Advantages:

- time between violation and notification is as tight as possible.

Drawbacks:

- exploits a non-standard Linux specific extension.

Constraint violation latency is kept low, but the resulting code only runs on Linux.

In this *preliminary* work, only this solution is being analyzed.

Experimental Set-Up (I)

Proposed mechanism is effective if occurrence-to-notification of a violation is small –at least compared with timing characteristics of the application–.

Results gathered from preliminary implementation on Linux:

- common desktop PC: 3.0 GHz Intel CPU, 2 GB RAM, hand-tailored 2.6.28 Linux kernel;
- one task implemented by a Linux thread running 1000 instances;
- task τ parameters: $(C, D, T) = (50, 50, 100)$ $msec$;
- kernel periodic tick frequency configured to 100, 250 and 1000 Hz;
- 1000 independent runs for each tick frequency value;

Experimental Set-Up (I)

Proposed mechanism is effective if occurrence-to-notification of a violation is small –at least compared with timing characteristics of the application–.

Results gathered from preliminary implementation on Linux:

- common desktop PC: 3.0 GHz Intel CPU, 2 GB RAM, hand-tailored 2.6.28 Linux kernel;
- one task implemented by a Linux thread running 1000 instances;
- task τ parameters: $(C, D, T) = (50, 50, 100)$ *msec*;
- kernel periodic tick frequency configured to 100, 250 and 1000 *Hz*;
- 1000 independent runs for each tick frequency value;

Experimental Set-Up (I)

Proposed mechanism is effective if occurrence-to-notification of a violation is small –at least compared with timing characteristics of the application–.

Results gathered from preliminary implementation on Linux:

- common desktop PC: 3.0 GHz Intel CPU, 2 GB RAM, hand-tailored 2.6.28 Linux kernel;
- one task implemented by a Linux thread running 1000 instances;
- task τ parameters: $(C, D, T) = (50, 50, 100)$ *msec*;
- kernel periodic tick frequency configured to 100, 250 and 1000 *Hz*;
- 1000 independent runs for each tick frequency value;

Experimental Set-Up (I)

Proposed mechanism is effective if occurrence-to-notification of a violation is small –at least compared with timing characteristics of the application–.

Results gathered from preliminary implementation on Linux:

- common desktop PC: 3.0 GHz Intel CPU, 2 GB RAM, hand-tailored 2.6.28 Linux kernel;
- one task implemented by a Linux thread running 1000 instances;
- task τ parameters: $(C, D, T) = (50, 50, 100)$ *msec*;
- kernel periodic tick frequency configured to 100, 250 and 1000 *Hz*;
- 1000 independent runs for each tick frequency value;

Experimental Set-Up (I)

Proposed mechanism is effective if occurrence-to-notification of a violation is small –at least compared with timing characteristics of the application–.

Results gathered from preliminary implementation on Linux:

- common desktop PC: 3.0 GHz Intel CPU, 2 GB RAM, hand-tailored 2.6.28 Linux kernel;
- one task implemented by a Linux thread running 1000 instances;
- task τ parameters: $(C, D, T) = (50, 50, 100)$ *msec*;
- kernel periodic tick frequency configured to 100, 250 and 1000 *Hz*;
- 1000 independent runs for each tick frequency value;

Experimental Set-Up (I)

Proposed mechanism is effective if occurrence-to-notification of a violation is small –at least compared with timing characteristics of the application–.

Results gathered from preliminary implementation on Linux:

- common desktop PC: 3.0 GHz Intel CPU, 2 GB RAM, hand-tailored 2.6.28 Linux kernel;
- one task implemented by a Linux thread running 1000 instances;
- task τ parameters: $(C, D, T) = (50, 50, 100)$ *msec*;
- kernel periodic tick frequency configured to 100, 250 and 1000 *Hz*;
- 1000 independent runs for each tick frequency value;

Experimental Set-Up (I)

Proposed mechanism is effective if occurrence-to-notification of a violation is small –at least compared with timing characteristics of the application–.

Results gathered from preliminary implementation on Linux:

- common desktop PC: 3.0 GHz Intel CPU, 2 GB RAM, hand-tailored 2.6.28 Linux kernel;
- one task implemented by a Linux thread running 1000 instances;
- task τ parameters: $(C, D, T) = (50, 50, 100)$ *msec*;
- kernel periodic tick frequency configured to 100, 250 and 1000 *Hz*;
- 1000 independent runs for each tick frequency value;

Experimental Set-UP (II)

Experiments description:

Deadline violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_within` block. *Difference between ideal and actual violation notification is measured.*

WCET violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_wcet` block. *Difference between ideal and actual violation notification is measured.*

Results presentation:

- tables with maximum, average and standard deviation of latency values for both experiments. (Minimum values are not interesting;)
- cumulative distribution function (CDF) of latency values for both experiments.

Experimental Set-UP (II)

Experiments description:

Deadline violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_within` block. *Difference between ideal and actual violation notification is measured.*

WCET violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_wcet` block. *Difference between ideal and actual violation notification is measured.*

Results presentation:

- tables with maximum, average and standard deviation of latency values for both experiments. (Minimum values are not interesting;)
- cumulative distribution function (CDF) of latency values for both experiments.

Experimental Set-UP (II)

Experiments description:

Deadline violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_within` block. *Difference between ideal and actual violation notification is measured.*

WCET violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_wcet` block. *Difference between ideal and actual violation notification is measured.*

Results presentation:

- tables with maximum, average and standard deviation of latency values for both experiments. (Minimum values are not interesting;)
- cumulative distribution function (CDF) of latency values for both experiments.

Experimental Set-UP (II)

Experiments description:

Deadline violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_within` block. *Difference between ideal and actual violation notification is measured.*

WCET violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_wcet` block. *Difference between ideal and actual violation notification is measured.*

Results presentation:

- tables with maximum, average and standard deviation of latency values for both experiments. (Minimum values are not interesting;)
- cumulative distribution function (CDF) of latency values for both experiments.

Experimental Set-UP (II)

Experiments description:

Deadline violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_within` block. *Difference between ideal and actual violation notification is measured.*

WCET violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_wcet` block. *Difference between ideal and actual violation notification is measured.*

Results presentation:

- tables with maximum, average and standard deviation of latency values for both experiments. (Minimum values are not interesting;)
- cumulative distribution function (CDF) of latency values for both experiments.

Experimental Set-UP (II)

Experiments description:

Deadline violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_within` block. *Difference between ideal and actual violation notification is measured.*

WCET violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_wcet` block. *Difference between ideal and actual violation notification is measured.*

Results presentation:

- tables with maximum, average and standard deviation of latency values for both experiments. (Minimum values are not interesting;)
- cumulative distribution function (CDF) of latency values for both experiments.

Experimental Set-UP (II)

Experiments description:

Deadline violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_within` block. *Difference between ideal and actual violation notification is measured.*

WCET violation latency meas.: τ forced to execute more than 50 *msec* inside a `try_wcet` block. *Difference between ideal and actual violation notification is measured.*

Results presentation:

- tables with maximum, average and standard deviation of latency values for both experiments. (Minimum values are not interesting;)
- cumulative distribution function (CDF) of latency values for both experiments.

Deadline violation notification latency measuring results (in *nsec*):

Experimental Results (I)

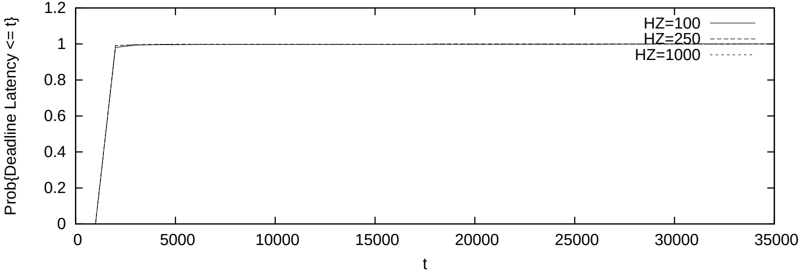
Deadline violation notification latency measuring results (in *nsec*):

	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

Experimental Results (I)

Deadline violation notification latency measuring results (in *nsec*):

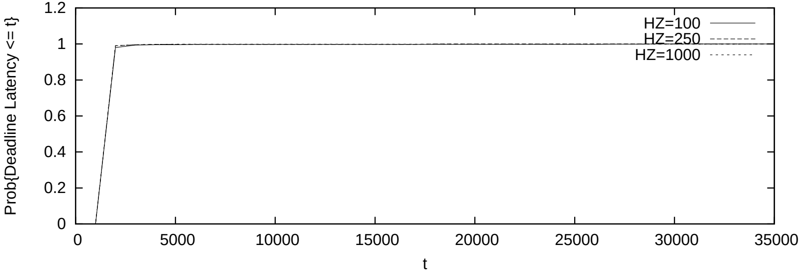
	max	mean	std. dev.
HZ=100	28610	1724.418	1187.854
HZ=250	17202	1595.095	711.1304
HZ=1000	33394	1602.544	1023.255



Experimental Results (I)

Deadline violation notification latency measuring results (in *nsec*):

	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

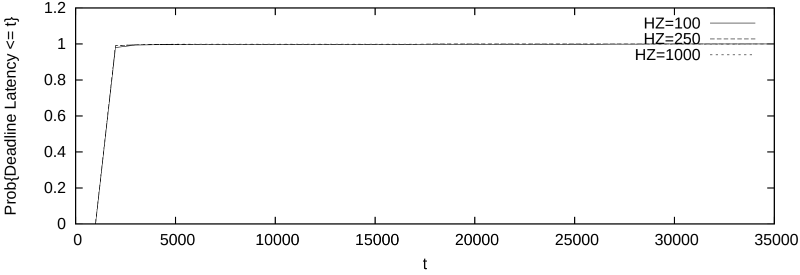


- deadline latency is always quite small (1.5 μs over 50 *ms*);

Experimental Results (I)

Deadline violation notification latency measuring results (in *nsec*):

	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



- deadline latency is always quite small (1.5 μs over 50 *ms*);
- deadline latency is independent on tick frequency.

WCET violation notification latency measuring results (in *nsec*):

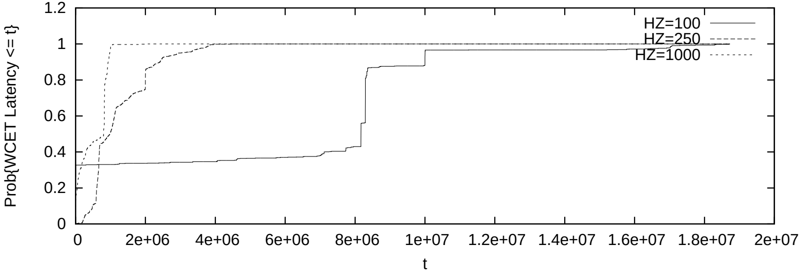
Experimental Results (II)

WCET violation notification latency measuring results (in *nsec*):

	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

WCET violation notification latency measuring results (in *nsec*):

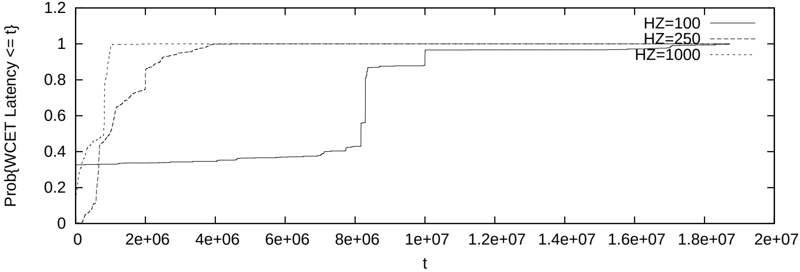
	max	mean	std. dev.
HZ=100	18727747	5748948.344	4474771.769
HZ=250	4423164	1233955.255	844593.486
HZ=1000	1999752	522228.673	390837.341



Experimental Results (II)

WCET violation notification latency measuring results (in *nsec*):

	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

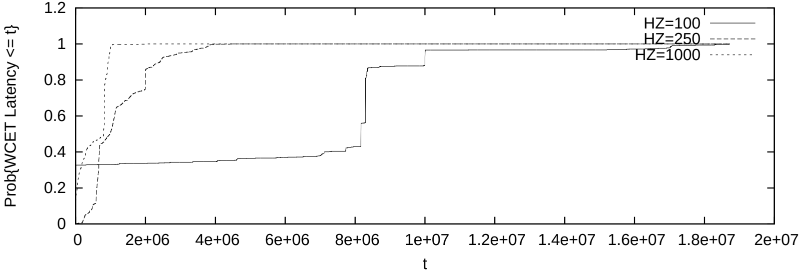


- WCET latency is much more bigger (0.5 *ms* over 50 *ms*);

Experimental Results (II)

WCET violation notification latency measuring results (in *nsec*):

	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



- WCET latency is much more bigger (0.5 *ms* over 50 *ms*);
- WCET latency is very dependent on tick frequency.

Conclusions

A set of linguistic constructs for the management of timing constraints during application design and implementation has been proposed. applications has been introduced.

Developers can thus focus on the main application flow: timing violation are catched dynamically by the framework.

Preliminary implementation of the framework on Linux.

Conclusions:

- dealing with timing violation as exceptions is viable an effective approach;
- the latency introduced is pretty small, especially for deadline constraints.

Conclusions

A set of linguistic constructs for the management of timing constraints during application design and implementation has been proposed. applications has been introduced.

Developers can thus focus on the main application flow: timing violation are catched dynamically by the framework.

Preliminary implementation of the framework on Linux.

Conclusions:

- dealing with timing violation as exceptions is viable an effective approach;
- the latency introduced is pretty small, especially for deadline constraints.

A set of linguistic constructs for the management of timing constraints during application design and implementation has been proposed. applications has been introduced.

Developers can thus focus on the main application flow: timing violation are catched dynamically by the framework.

Preliminary implementation of the framework on Linux.

Conclusions:

- dealing with timing violation as exceptions is viable an effective approach;
- the latency introduced is pretty small, especially for deadline constraints.

Conclusions

A set of linguistic constructs for the management of timing constraints during application design and implementation has been proposed. applications has been introduced.

Developers can thus focus on the main application flow: timing violation are catched dynamically by the framework.

Preliminary implementation of the framework on Linux.

Conclusions:

- dealing with timing violation as exceptions is viable an effective approach;
- the latency introduced is pretty small, especially for deadline constraints.

Conclusions

A set of linguistic constructs for the management of timing constraints during application design and implementation has been proposed. applications has been introduced.

Developers can thus focus on the main application flow: timing violation are catched dynamically by the framework.

Preliminary implementation of the framework on Linux.

Conclusions:

- dealing with timing violation as exceptions is viable an effective approach;
- the latency introduced is pretty small, especially for deadline constraints.

Conclusions

A set of linguistic constructs for the management of timing constraints during application design and implementation has been proposed. applications has been introduced.

Developers can thus focus on the main application flow: timing violation are catched dynamically by the framework.

Preliminary implementation of the framework on Linux.

Conclusions:

- dealing with timing violation as exceptions is viable an effective approach;
- the latency introduced is pretty small, especially for deadline constraints.

Future Work

Future work:

- investigate on combined user-kernel mechanism to further lower the introduced latency, especially for the WCET case (on Linux);
- thoroughly compare the POSIX variant of the framework implementation with the Linux-specific one (on Linux!);
- test the performance of the POSIX variant of the framework on OSes different than Linux;
- realize thorough performance comparison between our framework and the existing ones in RTSJ and Ada 2005.

Future Work

Future work:

- investigate on combined user-kernel mechanism to further lower the introduced latency, especially for the WCET case (on Linux);
- thoroughly compare the POSIX variant of the framework implementation with the Linux-specific one (on Linux!);
- test the performance of the POSIX variant of the framework on OSes different than Linux;
- realize thorough performance comparison between our framework and the existing ones in RTSJ and Ada 2005.

Future Work

Future work:

- investigate on combined user-kernel mechanism to further lower the introduced latency, especially for the WCET case (on Linux);
- thoroughly compare the POSIX variant of the framework implementation with the Linux-specific one (on Linux!);
- test the performance of the POSIX variant of the framework on OSes different than Linux;
- realize thorough performance comparison between our framework and the existing ones in RTSJ and Ada 2005.

Future Work

Future work:

- investigate on combined user-kernel mechanism to further lower the introduced latency, especially for the WCET case (on Linux);
- thoroughly compare the POSIX variant of the framework implementation with the Linux-specific one (on Linux!);
- test the performance of the POSIX variant of the framework on OSes different than Linux;
- realize thorough performance comparison between our framework and the existing ones in RTSJ and Ada 2005.

Future Work

Future work:

- investigate on combined user-kernel mechanism to further lower the introduced latency, especially for the WCET case (on Linux);
- thoroughly compare the POSIX variant of the framework implementation with the Linux-specific one (on Linux!);
- test the performance of the POSIX variant of the framework on OSes different than Linux;
- realize thorough performance comparison between our framework and the existing ones in RTSJ and Ada 2005.

Thank You for Your Time...

Questions?