# End-to-End Latency Analysis Framework

***Release 0.2-alpha***

**Jiri Kuncar <jiri.kuncar@gmail.com>**
**Rafia Inam <rafia.inam@mdh.se>**

March 04, 2013

# CONTENTS

Welcome to End-to-End Latency Analysis Framework's documentation. This documentation is divided into different parts. I recommend that you get started with *installation* and then head over to the *Quickstart*. If you'd rather dive into the internals of the library, check out the *API* documentation.

This framwork depends on two external libraries: the numpy and the argparse. These libraries are not documented here. If you want to dive into their documentation, check out the following links:

- Numpy Documentation
- Argparse Documentation

CHAPTER

ONE

# USER'S GUIDE

## 1.1 Quickstart

Eager to get started? This page gives a good introduction how the End-to-End Latency Analysis Framework works and how you can benefit from it. It assumes you already have it installed. If you do not, head over to the *installation* section.

### 1.1.1 Finding possible execution paths

The whole simulation is dependend on quick and effective algorithm for finding possible exectution paths of tasks in all system components. All latency types are calculated on specified data flow path that contains identifiers of tasks in analyzed system.

Our generator `generate_paths()` returns tuples with activation indexes of tasks accordingly to the analyzed execution path. The algorithm starts with finding closest activation indexes of the first task in path for defined interval. Following pseudocode shows simplified version of our algorithm using methods `alpha()` and `ialpha()` defined on `Task`.

```
function generate_paths(start, stop, tasks_in_path):
    paths <- list()
    task <- tasks_in_path.pop(0)  # assign and remove first task from the path
    loop i from task.ialpha(start) to task.ialpha(stop):
        # find a time range for next task
        if length(tasks_in_path) > 0:
            time <- task.alpha(i)
            next_task <- tasks_in_path[0]  # next task in path
            j <- next_task.ialpha(time)  # closest activation index
            new_start <- next_tasks.alpha(j)  # closest activation time
            # find posible paths for next task in path from new start.
            for all path in generate_paths(new_start, stop, tasks_in_path):
                # join current activation index with found tuple
                paths.append( path.prepend(i) )
            end for
        else:
            paths.append( list(i) )  # list with only one index
        end if
    end loop
    return paths
end function
```

CHAPTER

TWO

# API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

## 2.1 API

This part of the documentation covers all the interfaces of End-to-End Analyzer Framework.

### 2.1.1 Task Object

**class** `eelaf.`**`Task`**(*name*, *period*, *priority*, *exetime*)

System task.

**`C`**

Alias for task $t_i$ execution time ($C_i$).

**`HB`**

Set of tasks belonging to same `Component` *C* with priorities higher than itself one.

$$HB(s) = \{T_k \in C \mid P(k) > P(s)\}$$

**`P`**

Alias for task $t_i$ period ($P(i)$).

**static `alpha`**($i$)

Calculate time of *i*-th activation.

$$\alpha_r(i) = r_{start} + i * P(r)$$

**Parameters** **i** – Activation number.

**Returns** Activation time.

**`blocking_time`**

Task deadline ($b_i$).

> **Warning:** Currently always returns 0.

**`deadline`**

Task deadline.

**Note:** Currently *deadline==period*.

---

**static `delta`** (*i*)
Calculate response time of *i*-th activation.

**Parameters** **i** – Activation number.

**Returns** Response time of *i*-th activation.

---

**Note:** In our case we always return task response time.

---

**See Also:**

`response_time()`

**`freq`**
Task frequency $f(t_i) = 1/P(t_i)$.

**static `ialpha`** (*t*)
Calculate previous activation for given time.

$$\alpha_r^{-1}(t) = \lfloor \frac{t - r_{start}}{P(r)} \rfloor$$

**Parameters** **t** – Current time.

**Returns** int – activation number.

**`p`**
Alias for task $t_i$ priority ($p(i)$).

**`plan`** (*time*)
Returns True if the task should be scheduled at given *time*.

**static `rbf`** (*t*)
The request bound function.

It computes the maximum cumulative execution requests that could be generated from the time that task is released up to time *t*.

$$rbf_i(t) = C_i + b_i + \sum_{\tau_k \in HP(i)} \frac{t}{T_k} * C_k$$

**Parameters** **t** – Time limit for execution requests.

**See Also:**

[Inam2548:2011] formula (2).

**`response_time`**
Task response time.

$$RS(i) = t \mid \exists t : 0 < t <= D_i, rbf_i(t) <= sbf_C(t)$$

**`status`** (*time*)
Returns textual representation of task status at given *time*.

**`utilization`**
Calculates utilization.

### 2.1.2 Component (Subsystem) Object

**class** `eelaf.`**`Component`**(*name*, *period*, *priority*, *budget*, *scheduler='EDF'*, *payback=False*)

System servers / components.

**`Bl`**

The maximum blocking imposed to a this subsystem.

$$Bl(s) = max\{X(k) \mid S_k \in LPS(s)\}$$

**See Also:**

[Inam2548:2011] formula (10).

**`HPS`**

Set of subsystems of `System` *S* with priority higher than itself ($S_s$).

$$HSP(s) = \{S_k \in S \mid P(k) > P(s)\}$$

**See Also:**

Used in formula (9) in [Inam2548:2011].

**`LPS`**

Set of subsystems of `System` *S* with priority lower than itself ($S_s$).

$$LSP(s) = \{S_k \in S \mid P(k) < P(s)\}$$

**See Also:**

Used in formula (10) in [Inam2548:2011].

**`P`**

Alias for `Component` period.

**`Q`**

Alias for `Component` budget.

**`RBF`**(*t*)

Request Bound Function

**See Also:**

[Inam2548:2011] formula (9) and (11).

**`X`**

The maximum execution-time that any subsystem-internal task may lock a shared global resource.

> **Warning:** Currently always returns 0.

**`deadline`**

The function returns `Component` deadline.

Currently the *self.deadline==self.period*.

**`f1`**(*t*)

Implementation of helper formula *f1*.

**Parameters** **t** – time

**See Also:**

[Inam2548:2011] formula (5).

**f2**(*t*)
Implementation of helper formula *f2*.

**Parameters** **t** – time

**See Also:**

[Inam2548:2011] formula (6).

**freq**
The function calculates `Component` frequency.

**static sbf**(*t*)
Supply Bound Function.

If *payback* is *True* then this method depends on `f1()` and `f2()`:

```
return max(min(f1(t), f2(t)), 0)
```

**Parameters** **t** – time

**See Also:**

[Inam2548:2011] formula (3) and (4).

**schedulability**
`Component` schedulability condition.

```
for t in C_{tasks}:
    schedulable <- False
    for i in [1..P(t)]:
        if rbf_t(i) <= sbf_C(i):
            schedulable <- True
            break
        end if
    end for
    if not schedulable:
        return False
    end if
end for
return True
```

**See Also:**

[Inam2548:2011] formula (13).

**utilization**
Returns `Component` utilization as product of its frequency and budget.

### 2.1.3 System Object

**class** eelaf.**System**(*scheduler='FPS'*, *resolution=1000*, *components=None*)
Models a physical system with instances of `Component`.

**B** (*t*)

Blocking time left at given *time*.

> **Warning:** Currently always returns 0.

**static TP_first** (*possible_paths*)

Set of all non-duplicate, reachable timed paths, for which no timed path exists that shares the same start instance of the first task and has an earlier end instance of the last task.

$$\mathbb{TP}^{first} = \{\vec{tp} \in \mathbb{TP}^{reach} \mid \neg\exists \vec{tp'} \in \mathbb{TP}^{reach} : tp'_1 = tp_1 \wedge tp'_n < tp_n\}$$

```
function earlier(tp, tp'):
    # index -1 gets last element in array
    return tp'[0] == tp[0] and tp'[-1] < tp[-1]
end function

out <- list()
TP_reach_paths <- TP_reach(possible_paths)
for tp in TP_reach_paths:
    if not any(map(partial(earlier, tp), TP_reach_paths))):
        out.append(tp)
    end if
end for
return out
```

**See Also:**

[Feiertag:08] formula (11).

**static TP_reach** (*possible_paths*)

It obtains the set of all paths and returns only all reachable timed path ($\mathbb{TP}^{reach}$).

```
out <- list()
for all path in possible_paths:
    if reach_path(path):
        out.append(path)
    end if
end for
return out
```

**See Also:**

[Feiertag:08] formula (9).

**addComponent** (*component*)

Adds new `Component` instance to the system.

It stores reference of the system to added component.

**Parameters** **component** (`Component`) – New system subsystem.

**static att** (*w*, *i*, *r*, *j*)

It returns *True* if "activation time travel" occurs ($att(t_w(i) \rightarrow t_r(j))$).

The activation time travel occurs when the reader is activated before the writer ($\alpha_r(i)$ is equivalent to `alpha()` on `t()`).

$$att(t_w(i) \rightarrow t_r(j)) = \alpha_r(j) < \alpha_w(i)$$

**Parameters**

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

**See Also:**

[Feiertag:08] formula (3)

**static** **crit**(*w*, *i*, *r*, *j*)

The "critical function" determines if writer and reader overlap in execution even in case of non-activation time travel ($crit(t_w(i) \rightarrow t_r(j))$).

$$crit(t_w(i) \rightarrow t_r(j)) = \alpha_r(j) < \alpha_w(i) + \delta_w(i)$$

**Parameters**

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

**See Also:**

[Feiertag:08] formula (4)

**delta_FF**(*ls*)

Find maximum of First-to-First path delays.

**See Also:**

[Feiertag:08] formula (17).

**delta_FF_path**(*path*, *tp_reach*)

Calculate First-to-First path delay.

$$\Delta^{FF}(\vec{tp}) = \Delta^{LF}(\vec{tp}) + \alpha_1(tp_1) - \alpha_1(pred(\vec{(tp)}))$$

**Parameters**

- **path** – Array with task activation numbers.
- **tp_reach** – List of reachable paths.

**Fixme** remove dependency on *tp_reach* parameter.

**See Also:**

[Feiertag:08] formula (16).

**delta_FL**(*ls*)

Find maximum of First-to-Last path delays.

**See Also:**

[Feiertag:08] formula (15).

**delta_FL_path**(*path*, *tp_reach*)
Calculate First-to-Last path delay (uses `pred()`).

$$\Delta^{FL}(\vec{tp}) = \Delta^{LL}(\vec{tp}) + \alpha_1(tp_1) - \alpha_1(pred(\vec{(tp)}))$$

**Parameters**

- **path** – Array with task activation numbers.
- **tp_reach** – List of reachable paths.

**Fixme** remove dependency on *tp_reach* parameter.

**See Also:**

[Feiertag:08] formula (14).

static **delta_LF**(*ls*)
The maximum "Last-to-First" timed path delay.

$$\Delta^{LF}(p) = max\{\Delta(\vec{(tp)}) \mid \vec{(tp)} \in \mathbb{TP}^{first}\}$$

**See Also:**

[Feiertag:08] formula (12).

static **delta_LL**(*possible_paths*)
Returns maximum latency over all reachable paths (`TP_reach()`).

$$\Delta^{LL}(possible_paths) = max\{\Delta(path) \mid path \in \mathbb{TP}^{reach}\}$$

```
# map .. calls function for each element in list
# max .. returns maximal element from list
return max(map(delta_path, TP_reach(possible_paths)))
```

**See Also:**

[Feiertag:08] formula (10).

**delta_LL_path**(*path*)
Calculate Last-to-Last *path* delay using `delta_path()`.

**delta_path**(*path*)
Calculate end-to-end *path* delay.

$$\Delta(path) = \alpha_n(path_n) + \delta_n(path_n) - \alpha_1(path_1)$$

**See Also:**

[Feiertag:08] formula (2).

static **forw**(*w*, *i*, *r*, *j*)
It determines the forward reachability of the two task instances $t_w$ and $t_r$.

$$forw(t_w(i) \rightarrow t_r(j)) = \neg att(t_w(i) \rightarrow t_r(j)) \wedge (\neg crit(t_w(i) \rightarrow t_r(j)) \vee wait(t_w(i) \rightarrow t_r(j)))$$

**Parameters**

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

**See Also:**

[Feiertag:08] formula (6)

**generate_paths** (*index*, *start*, *stop*)

Generator of possible paths for given task in path.

It yields tuples with task activation index starting from 'index'th task in defined path.

```
paths <- list()
task <- tasks_in_path.pop(0)  # assign and remove first task from the path
loop i from task.ialpha(start) to task.ialpha(stop):
    # find a time range for next task
    if length(tasks_in_path) > 0:
        time <- task.alpha(i)
        next_task <- tasks_in_path[0]  # next task in path
        j <- next_task.ialpha(time)  # closest activation index
        new_start <- next_tasks.alpha(j)  # closest activation time
        # find posible paths for next task in path from new start.
        for all path in generate_paths(new_start, stop, tasks_in_path):
            # join current activation index with found tuple
            paths.append( path.prepend(i) )
        end for
    else:
        paths.append( list(i) )  # list with only one index
    end if
end loop
return paths
```

**pred** (*path*, *tp_reach*)

Temporal distance to the start of the latest previous "last-to-x" path.

**See Also:**

[Feiertag:08] formula (13).

**static reach** (*w*, *i*, *r*, *j*)

The output of an instance $t_w(i)$ is overwritten by instance $t_w(i+1)$ when both instances can forward reach the same reading task instance $t_r(j)$. In other words, $t_w(i)$ can reach $t_r(j)$ if and only if the following function returns *True*:

$$reach(t_w(i) \rightarrow t_r(j)) = (forw(t_w(i) -> t_r(j)) \wedge \neg forw(t_w(i+1) -> t_r(j)))$$

**Parameters**

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

**See Also:**

[Feiertag:08] formula (7).

**static reach_path** (*path*)
Check *path* rechability.

```
path_length <- length(path)
for i in [0..path_length-1):
    tp_i  <- path[i]
    tp_i1 <- path[i+1]
    if reach(t_w(tpi) -> t_{w+1}(tp_i1)):
        return False
    end if
end for
return True
```

**See Also:**

[Feiertag:08] formula (8).

**schedulability**
This method checks the global schedulability condition.

```
for C in components:
    # P(C) .. period of component C
    schedulable <- False
    for t in [0..P(C)]:
        if RBF(C, t) <= t:
            schedulable <- True
            break
        end if
    end for
    if not schedulable:
        return False
    end if
end for
return True
```

**See Also:**

[Inam2548:2011] formula (8).

**t** (*i*)
Get *i*-th `Task` instance ($t_i$).

**Parameters** **i** – The index of system task starting from 0.

**Returns** Instance of `Task`.

**tasks**
All system tasks.

**tasks_in_path**
List of tasks in data path.

**utilization**
Returns system utilization calculated as sum of component utilizations.

**static wait** (*w*, *i*, *r*, *j*)
It determines if the writer finishes first, because the reader has to wait due to its priority in case of over-

lapped but not time-traveling execution ($wait(t_w(i) \rightarrow t_r(j))$).

$$wait(t_w(i) \rightarrow t_r(j)) = p(t_r) < p(t_w)$$

**Parameters**

- **w** – Index of writer task in data path.
- **i** – Activation index of writer task.
- **r** – Index of reader task in data path.
- **j** – Activation index of reader task.

**See Also:**

[Feiertag:08] formula (5)

CHAPTER

THREE

# ADDITIONAL NOTES

Design notes, legal information and changelog are here for the interested.

## 3.1 Licence

Source code is distributed under following GNU/GPLv2 licence.

```
EELAF is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the Free Software Foundation; either version 2 of the
License, or (at your option) any later version.

Analysis framework is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with Invenio; if not, write to the Free Software Foundation, Inc.,
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.
```

## 3.2 Bibliography

CHAPTER

FOUR

# INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

# BIBLIOGRAPHY

[Feiertag:08] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, “A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics”, In Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS‘08).

[Inam2548:2011] R. Inam, J. Maki-Turja, M. Sjodin, and M. Behnam, “Hard Real-time Support for Hierarchical Scheduling in FreeRTOS”, 7th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT‘11).