pycpa Documentation

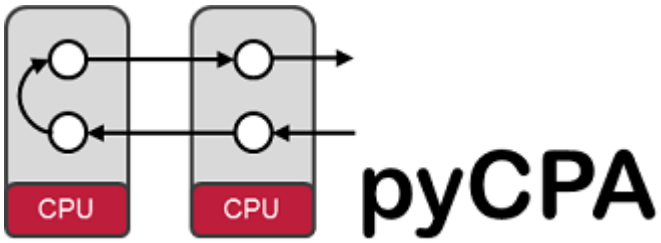
Release current

Diemer, Axer, Thiele, Schlatow

Jan 08, 2021

Contents

pyCPA is a pragmatic Python implementation of Compositional Performance Analysis (aka the SymTA/S approach provided by Symtavision (now: Luxoft)) used for research in worst-case timing analysis. Unlike the commercial SymTA/S tool, pyCPA is not intended for commercial-grade use and does not guarantee correctness of the implementation.



Contents:

CHAPTER 1

Installation

Before you can install pyCPA, you must set up Python (*Prerequisites*). If you already have a Python installation, you may directly proceed with *Quickstart*. Alternatively, if you want to modify the pyCPA code, please consult *For Developers*.

A brief introduction in how to test your installation is provided in *Using pyCPA*.

1.1 Prerequisites

First of all, you need a basic Python (2.7 or 3.x) environment. As a Linux user, you most probably have Python already installed. On Windows, we recommend to use Python(x,y), which includes a comprehensive set of scientific Python libraries and tools as well as related documentation. Python(x,y) comes with several interactive consoles (based on IPython), editors and applications. For your first hands-on experience, we recommend using `Spyder` as an IDE. You can also run a command prompt via the Python(x,y) icon on the Desktop and choosing `IPython (sh)` as an interactive console.

1.2 Quickstart

The easiest way to install pyCPA is by using pip. For a system-wide installation of the current pyCPA version, you simply run the following command:

```
$ pip install https://github.com/IDA-TUBS/pycpa/archive/master.zip
```

Alternatively, e.g. if you do not have admin privileges, you can install pyCPA for the current user:

```
$ pip install --user https://github.com/IDA-TUBS/pycpa/archive/master.zip
```

1.3 For Developers

pyCPA has the following dependencies:

- Required Python packages: setuptools, argparse, pygraphviz, matplotlib
- Optional Python packages: numpy, simpy, xlrd

Before proceeding, you might want to check the status of your Python installation, i.e. what version is installed (if at all) and what Python packages are already available, using the following commands:

```
$ python --version
$ pydoc modules
```

For downloading the pyCPA source code, you simply create a clone from the git git repository, e.g. by running the following command:

```
$ git clone https://github.com/IDA-TUBS/pycpa/
```

From within the pyCPA repository, execute the following command to install pyCPA in editable mode (i.e. changes to the source code do not require re-installation to be effective):

```
$ pip install --user -e .
```

1.4 Testing and using pyCPA

Congratulations, you have installed pyCPA!

In order to test pyCPA, you may want to run the examples which are provided with the distribution. The quickest way to do this is to run the following on the command prompt (e.g. `IPython (sh)` on Windows):

```
$ python /path/to/pycpa/examples/spp_test.py
```

If you want to know what this examples does and how it works checkout the *Static Priority Preemtive Example*.

Depending on your personal preferences, you may also use an IDE of which we give a more detailed account in the following sections.

1.4.1 Using an IDE: Spyder (Windows)

Spyder is installed with Python(x,y). Simply open one of the example files (e.g. spp_test.py) and click the `Run` button.

1.4.2 Using an IDE: PyDev

You may also use Eclipse with PyDev as IDE, which can be installed by the following steps:

1. Make sure that you have installed Python *BEFORE* you install Eclipse.
2. Download from http://www.eclipse.org/downloads/eclipse-packages/ the current Eclipse release for Windows 32 bit (!). Extract the zip-file, execute `eclipse.exe` and follow the installation instructions.
3. Open Eclipse and specify a workspace. If you open a workspace for the first time, you will have to close the Welcome tab, before proceeding to your workspace.

4. Select the menu item `Help -> Install New Software`, search for the site http://pydev.org/updates. Select and install the item "PyDev" which will be displayed in the list of available software.

Now, you can set up a pyCPA project as follows:

1. Open the PyDev-Perspective by selecting in the main menu `Window -> Open Perspective -> Other -> PyDev`
2. Select in the main menu `File -> New -> PyDev Project`.
3. In the PyDev-Project Window specify a project name; the project will be saved to your workspace unless specified otherwise.
4. Choose the project type "Python" and select the 2.7 interpreter version.
5. Click on "Please configure an interpreter before proceeding".
   i. Select `Manual Config` in the pop-up window.
   ii. In the settings for the Python interpreter click `New...` and specify an interpreter name, e.g. Python27, and the path to the interpreter executable (e.g. `C:\myPathToPython\python.exe`). In the appearing pop-up window select all options.
   iii. In the tab `Libraries`, select `New Folder` and specify the path to the pyCPA-folder (e.g. `C:\MyPathTo\pycpa`).
   iv. Close the preferences window by clicking ok.
6. Back in the PyDev-Project Window, click `add project directory to PYTHONPATH` and then the button `Finish`.
7. You may now add a Python file to your project (right-click on your project in the PyDev Package Explorer -> New... -> File) and write a Python program (e.g. test.py) which uses pyCPA.
8. To run test.py, right-click on `test.py` and select `Run as -> Python Run`. If you want to modify your run settings in order to e.g. specify arguments, select `Run as -> Run Configurations` and adapt the settings as needed before clicking `Run` in the Run Configurations Window.
9. You may also try out the examples which are provided with pyCPA such as the *Static Priority Preemtive Example*.

CHAPTER 2

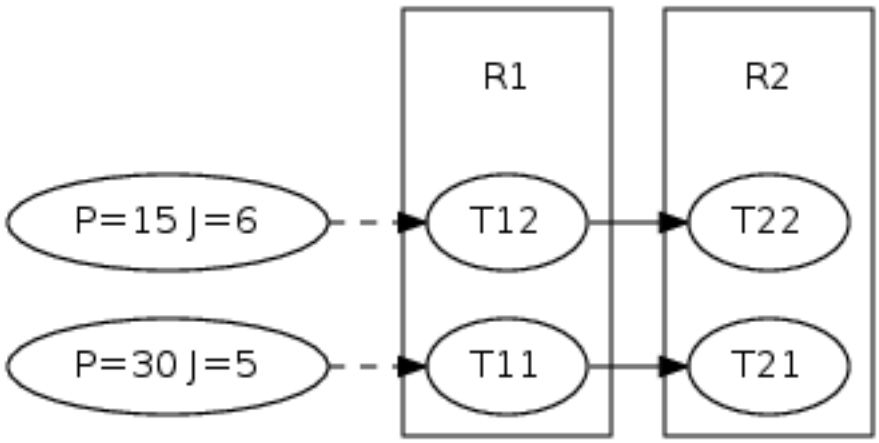
Examples

Contents:

2.1 Static Priority Preemtive Example

2.1.1 Introduction

In this section, we will dissect the SPP example which is representative for the ideas behind pyCPA. The full source code of the example is shown at the end of this section.



Before we begin some general reminder:

pyCPA
is NOT a
tool! It rather is a package of methods and classes which can be embedded into your pyhon application - the spp example is such an example application.

Each pyCPA program consists of three steps:

- initialization
- setting up the architecture
- one or multiple scheduling analyses

The architecture can be entered in two ways, either you provide it with the source code or you can use an XML loader such as the Symta or the SMFF loader. However, in most cases it is sufficient to code your architecture directly in a python file. For this example we assume that our architecture consists of two resources (e.g. CPUs) scheduled by an static-priority-preemptive (SPP) scheduler and four tasks of which some communicate by event-triggering. The environment stimulus (e.g. an sensor or input from another system) is assumed to be periodic with jitter. The application graph is shown on on the right.

2.1.2 Initialization

Now, let's look at the example. Before we actually start with the program, we import all pycpa modules which are needed for this example

```
from pycpa import model
from pycpa import analysis
from pycpa import schedulers
from pycpa import graph
from pycpa import options
```

The interesting module are `pycpa.spp` which contains scheduler specific algorithms, *pycpa.graph* which is used to plot a task graph of this example and *pycpa.options* which controls various modes in which pyCPA can be executed.

pyCPA can be initialized by *pycpa.options.init_pycpa()*. This will parse the pyCPA related options such as the propagation method, verbosity, maximum-busy window, etc. Conveniently, this also prints the options which will be used for your pyCPA session. This is handy, when you run some analyses in batch jobs and want are uncertain about the exact settings after a few weeks. However, the explicit call of this function is not necessary most of the time, as it is being implicitly called at the beginning of the analysis. It can be useful to control the exact time where the initialization happens in case you want to manually override some options from your code.

2.1.3 System Model

Now, we create an empty system, which is just a container for all other objects:

```
    # generate an new system
    s = model.System()
```

The next step is to create two resources R1 and R2 and bind them to the system via *pycpa.model.System.bind_resource()*. When creating a resource via *pycpa.model.Resource()*, the first argument of the constructor sets the resource id (a string) and the second defines the scheduling policy. The scheduling policy is defined by a reference to an instance of a scheduler class derived from *pycpa.analysis.Scheduler*. For SPP, this is `pycpa.spp.SPPScheduler`. In this class, different functions are defined which for instance compute the multiple-event busy window on that resource or the stopping condition for that particular scheduling policy. The

stopping condition specifies how many activations of a task have to be considered for the analysis. The default implementations of these functions from *pycpa.analysis.Scheduler* can be used for certain schedulers, but generally should be overridden by scheduler-specific versions. For SPP we have to look at all activations which fall in the level-i busy window, thus we choose the spp stopping condition.

```
    r1 = s.bind_resource(model.Resource("R1", schedulers.SPPScheduler()))
    r2 = s.bind_resource(model.Resource("R2", schedulers.SPPScheduler()))
```

The next part is to create tasks via *pycpa.model.Resource()* and bind them to a resource via *pycpa.model.Resource.bind_task()*. For tasks, we pass some parameters to the constructor, namely the identifier (string), the scheduling_paramter denoting the priority, and the worst- and best-case execution times (wcet and bcet).

```
    # create and bind tasks to r1
    t11 = r1.bind_task(model.Task("T11", wcet=10, bcet=5, scheduling_parameter=1))
    t12 = r1.bind_task(model.Task("T12", wcet=3, bcet=1, scheduling_parameter=2))

    # create and bind tasks to r2
    t21 = r2.bind_task(model.Task("T21", wcet=2, bcet=2, scheduling_parameter=1))
    t22 = r2.bind_task(model.Task("T22", wcet=9, bcet=4, scheduling_parameter=2))
```

In case tasks communicate with each other through event propagation (e.g. one task fills the queue of another task), we model this through task links, which are created by *pycpa.model.Task.link_dependent_task()* A task link is abstract and does not consume any additional time. In case of communication-overhead it must be modeled by using other resources/tasks.

```
    # specify precedence constraints: T11 -> T21; T12-> T22
    t11.link_dependent_task(t21)
    t12.link_dependent_task(t22)
```

Lastly we must set the input event models for all tasks that are not activated by another task. An event model is an abstraction of possible activation patterns and are commonly parametrized by an activation period P, a jitter J (optional) and an minimum distance d (optional). Such an event model is created by *pycpa.model.PJdEventModel*.

```
    # register a periodic with jitter event model for T11 and T12
    t11.in_event_model = model.PJdEventModel(P=30, J=5)
    t12.in_event_model = model.PJdEventModel(P=15, J=6)
```

2.1.4 Plotting the Task-Graph

Then, we plot the taskgraph to a pdf file by using *pycpa.graph.graph_system()* from the graph module.

```
    # plot the system graph to visualize the architecture
    g = graph.graph_system(s, 'spp_graph.pdf', dotout='spp_graph.dot')
```

2.1.5 Analysis

The analysis is performed by calling *pycpa.analysis.analyze_system()*. This will will find the fixed-point of the scheduling problem and terminate if a result was found or if the system is not feasible (e.g. one busy window or the amount a propagations was larger than a limit or the load on a resource is larger one).

```
    # perform the analysis
    print("Performing analysis")
    task_results = analysis.analyze_system(s)
```

pycpa.analysis.analyze_system() returns a dictionary with results for each task in the form of instances to *pycpa.analysis.TaskResult*. Finally, we print out some of the results:

```
    # print the worst case response times (WCRTs)
    print("Result:")
    for r in sorted(s.resources, key=str):
        for t in sorted(r.tasks, key=str):
            print("%s: wcrt=%d" % (t.name, task_results[t].wcrt))
```

The output of this example is:

```
pyCPA - Compositional Performance Analysis in Python.

Version 1.2
Copyright (C) 2010-2017, TU Braunschweig, Germany. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.

invoked via: examples/spp_test.py

check_violations :       False
debug            :       False
e2e_improved     :       False
max_iterations   :        1000
max_wcrt         :         inf
nocaching        :       False
propagation      : busy_window
show             :       False
verbose          :       False



Performing analysis
Result:
T11: wcrt=10
    b_wcrt=q*WCET:1*10=10
T12: wcrt=13
    b_wcrt=T11:eta*WCET:1*10=10, q*WCET:1*3=3
T21: wcrt=2
    b_wcrt=q*WCET:1*2=2
T22: wcrt=19
    b_wcrt=T21:eta*WCET:1*2=2, q*WCET:2*9=18
```

As you can see, the worst-case response times of the tasks are 10, 13, 2 and 19.

This is the full spp-test file.

```
"""
| Copyright (C) 2010 Philip Axer
| TU Braunschweig, Germany
| All rights reserved.
| See LICENSE file for copyright and license details.

:Authors:
         - Philip Axer

Description
-----------

Simple SPP example
"""

from pycpa import model
from pycpa import analysis
from pycpa import schedulers
from pycpa import graph
from pycpa import options

def spp_test():
    # init pycpa and trigger command line parsing
    options.init_pycpa()

    # generate an new system
    s = model.System()

    # add two resources (CPUs) to the system
    # and register the static priority preemptive scheduler
    r1 = s.bind_resource(model.Resource("R1", schedulers.SPPScheduler()))
    r2 = s.bind_resource(model.Resource("R2", schedulers.SPPScheduler()))

    # create and bind tasks to r1
    t11 = r1.bind_task(model.Task("T11", wcet=10, bcet=5, scheduling_parameter=1))
    t12 = r1.bind_task(model.Task("T12", wcet=3, bcet=1, scheduling_parameter=2))

    # create and bind tasks to r2
    t21 = r2.bind_task(model.Task("T21", wcet=2, bcet=2, scheduling_parameter=1))
    t22 = r2.bind_task(model.Task("T22", wcet=9, bcet=4, scheduling_parameter=2))

    # specify precedence constraints: T11 -> T21; T12-> T22
    t11.link_dependent_task(t21)
    t12.link_dependent_task(t22)

    # register a periodic with jitter event model for T11 and T12
    t11.in_event_model = model.PJdEventModel(P=30, J=5)
    t12.in_event_model = model.PJdEventModel(P=15, J=6)

    # plot the system graph to visualize the architecture
    g = graph.graph_system(s, 'spp_graph.pdf', dotout='spp_graph.dot')

    # perform the analysis
    print("Performing analysis")
```

(continues on next page)

(continued from previous page)

```
    task_results = analysis.analyze_system(s)

    # print the worst case response times (WCRTs)
    print("Result:")
    for r in sorted(s.resources, key=str):
        for t in sorted(r.tasks, key=str):
            print("%s: wcrt=%d" % (t.name, task_results[t].wcrt))
            print("    b_wcrt=%s" % (task_results[t].b_wcrt_str()))


    expected_wcrt = dict()
    expected_wcrt[t11] = 10
    expected_wcrt[t12] = 13
    expected_wcrt[t21] = 2
    expected_wcrt[t22] = 19

    for t in expected_wcrt.keys():
        assert(expected_wcrt[t] == task_results[t].wcrt)

if __name__ == "__main__":
    spp_test()
```

2.2 Tutorial

2.2.1 Introduction

In this section, we will assemble several pyCPA examples step-by-step.

Before we begin some general reminder:

pyCPA is NOT a tool! It rather is a package of methods and classes which can be embedded into your python application.

Each pyCPA program consists of three steps:

- initialization
- setting up the architecture
- one or multiple scheduling analyses

The architecture can be entered in two ways, either you provide it with the source code or you can use an XML loader such as the SMFF loader, the Almathea parser or the task chain parser. However, in most cases it is sufficient to code your architecture directly in a python file on which we will focus in this tutorial.

2.2.2 Initialization

Now, let's look at the example. Before we actually start with the program, we import the all pycpa modules

```
from pycpa import *
```

Note that a few modules - such as the *pycpa.smff_loader*, *pycpa.cparpc* and `pycpa.simulation` - must be imported explicitly as they require additional third-party modules.

pyCPA can be initialized by *pycpa.options.init_pycpa()*. This will parse the pyCPA related options such as the propagation method, verbosity, maximum-busy window, etc. Conveniently, this also prints the options which will be used for your pyCPA session. This is handy, when you run some analyses in batch jobs and want are uncertain about the exact settings after a few weeks. However, the explicit call of this function is not necessary most of the time, as it is being implicitly called at the beginning of the analysis. It can be useful to control the exact time where the initialization happens in case you want to manually override some options from your code.

2.2.3 Step 1: Base Scenario

In the first step, we want to model and analyse our base scenario as depicted in the figure. It comprises two CPUs, a bus and two task chains. Task T11 and T12 execute on CPU1 and, once completed, activate the bus-communication tasks T21 and T22 respectively. On CPU2, T31 and T32 are activated by their preceding communication tasks.

System Model

First, we create an empty system, which is just a container for all other objects:

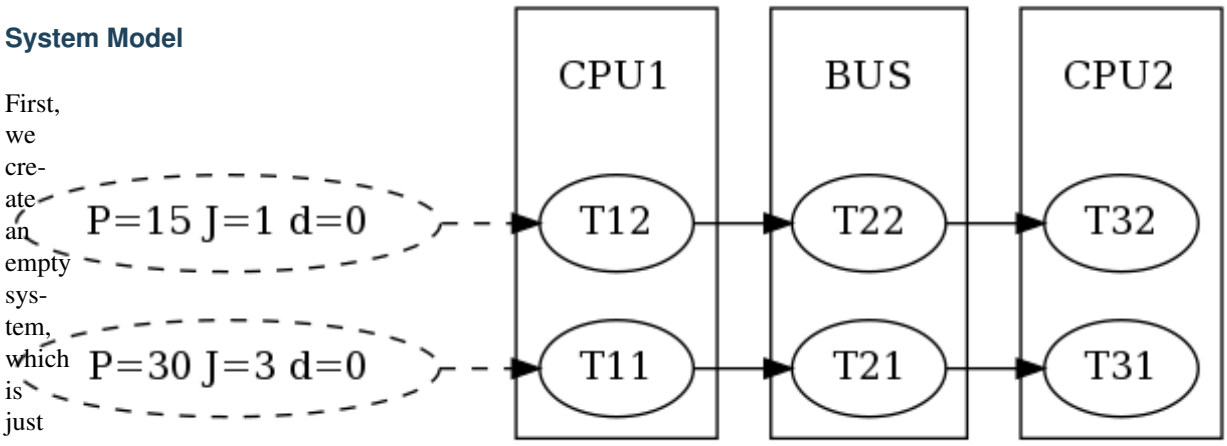


Fig. 1: Base scenario

```
# generate an new system
s = model.System('step1')
```

The next step is to create the three resources and bind them to the system via *pycpa.model.System.bind_resource()*. When creating a resource via *pycpa.model.Resource()*, the first argument of the constructor sets the resource id (a string) and the second defines the scheduling policy. The scheduling policy is defined by a reference to an instance of a scheduler class derived from *pycpa.analysis.Scheduler*. For SPP, this is *pycpa.schedulers.SPPScheduler* which we use for both processing resources. For the bus, we use *pycpa.schedulers.SPNPScheduler*.

```
    # add three resources (2 CPUs, 1 Bus) to the system
    # and register the SPP scheduler (and SPNP for the bus)
    r1 = s.bind_resource(model.Resource("CPU1", schedulers.SPPScheduler()))
    r2 = s.bind_resource(model.Resource("BUS",  schedulers.SPNPScheduler()))
    r3 = s.bind_resource(model.Resource("CPU2", schedulers.SPPScheduler()))
```

The next part is to create tasks and bind them to a resource via *pycpa.model.Resource.bind_task()*. For tasks, we pass some parameters to the constructor, namely the identifier (string), the scheduling_parameter, and the worst- and best-case execution times (wcet and bcet). The scheduling_parameter is evaluated by the scheduler which was assigned to the resource. For SPP and SPNP, it specifies the priority. By default higher numbers denote lower priorities.

```
    # create and bind tasks to r1
    t11 = r1.bind_task(model.Task("T11", wcet=10, bcet=5, scheduling_parameter=2))
    t12 = r1.bind_task(model.Task("T12", wcet=3, bcet=1, scheduling_parameter=3))

    # create and bind tasks to r2
    t21 = r2.bind_task(model.Task("T21", wcet=2, bcet=2, scheduling_parameter=2))
    t22 = r2.bind_task(model.Task("T22", wcet=9, bcet=5, scheduling_parameter=3))

    # create and bind tasks to r3
    t31 = r3.bind_task(model.Task("T31", wcet=5, bcet=3, scheduling_parameter=3))
    t32 = r3.bind_task(model.Task("T32", wcet=3, bcet=2, scheduling_parameter=2))
```

In case tasks communicate with each other through event propagation (e.g. one task fills the queue of another task), we model this through task links, which are created by *pycpa.model.Task.link_dependent_task()* A task link is abstract and does not consume any additional time. In case of communication-overhead it must be modeled by using other resources/tasks.

```
    # specify precedence constraints: T11 -> T21 -> T31; T12-> T22 -> T32
    t11.link_dependent_task(t21).link_dependent_task(t31)
    t12.link_dependent_task(t22).link_dependent_task(t32)
```

Last, we need to assign activation patterns (aka input event models) to the first tasks in the task chains, i.e. T11 and T12. We do this by assigning a periodic with jitter model, which is implemented by *pycpa.model.PJdEventModel()*.

```
    # register a periodic with jitter event model for T11 and T12
    t11.in_event_model = model.PJdEventModel(P=30, J=3)
    t12.in_event_model = model.PJdEventModel(P=15, J=1)
```

Plotting the Task-Graph

After creating the system model, we can use *pycpa.graph.graph_system()* from the graph module in order to visualize the task graph. Here, we create a DOT (graphviz) and PDF file.

```
    # graph the system to visualize the architecture
    g = graph.graph_system(s, filename='%s.pdf' % s.name, dotout='%s.dot' % s.name,␣
↪show=False, chains=chains)
```

Analysis

The analysis is performed by calling *pycpa.analysis.analyze_system()*. This will will find the fixed-point of the scheduling problem and terminate if a result was found or if the system is not feasible (e.g. one busy window or the amount a propagations was larger than a limit or a resource is overloaded).

```
    # perform the analysis
    print("\nPerforming analysis of system '%s'" % s.name)
    task_results = analysis.analyze_system(s)
```

pycpa.analysis.analyze_system() returns a dictionary with results for each task in the form of instances to *pycpa.analysis.TaskResult*. Finally, we print out the resulting worst-case response times and the corresponding details of the busy-window in which the worst-case response-time was found.

```
    # print the worst case response times (WCRTs)
    print("Result:")
    for r in sorted(s.resources, key=str):
        for t in sorted(r.tasks & set(task_results.keys()), key=str):
            print("%s: wcrt=%d" % (t.name, task_results[t].wcrt))
            print("    b_wcrt=%s" % (task_results[t].b_wcrt_str()))
```

The output of this example is:

```
pyCPA - Compositional Performance Analysis in Python.

Version 1.2
Copyright (C) 2010-2017, TU Braunschweig, Germany. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.

invoked via: examples/tutorial.py

check_violations :       False
debug            :       False
e2e_improved     :       False
max_iterations   :        1000
max_wcrt         :         inf
nocaching        :       False
propagation      : busy_window
show             :       False
verbose          :       False
```

(continues on next page)

(continued from previous page)

```
Performing analysis of system 'step1'
Result:
T21: wcrt=11
    b_wcrt=blocker:9, q*WCET:1*2=2
T22: wcrt=11
    b_wcrt=T21:eta*WCET:1*2=2, blocker:0, q*WCET:1*9=9
T11: wcrt=10
    b_wcrt=q*WCET:1*10=10
T12: wcrt=13
    b_wcrt=T11:eta*WCET:1*10=10, q*WCET:1*3=3
T31: wcrt=11
    b_wcrt=T32:eta*WCET:2*3=6, q*WCET:1*5=5
T32: wcrt=3
    b_wcrt=q*WCET:1*3=3
```

As you can see, the worst-case response times of the tasks are 11, 11, 10, 13, 11 and 3. We can also see, that for T21, a lower-priority blocker (T22) has been accounted as required for SPNP scheduling.

End-to-End Path Latency Analysis

After the WCRT analysis, we can additionally calculate end-to-end latencies of task chains. For this, we first need to define *pycpa.model.Path* objects and bind them to the system via *pycpa.model.System.bind_path()*. A path is created from a name and a sequence of tasks. Note that, the tasks will be automatically linked according to the given sequence if the corresponding task links are not already registered.

```
    # specify paths
    p1 = s.bind_path(model.Path("P1", [t11, t21, t31]))
    p2 = s.bind_path(model.Path("P2", [t12, t22, t32]))
```

The path analysis is invoked by *pycpa.path_analysis.end_to_end_latency()* with the path to analysis, the task_results dictionary and the number of events. It returns the minimum and maximum time that it may take on the given path to process the given number of events.

```
    # perform path analysis of selected paths
    for p in paths:
        best_case_latency, worst_case_latency = path_analysis.end_to_end_latency(p,␣
↪task_results, n=1)
        print("path %s e2e latency. best case: %d, worst case: %d" % (p.name, best_
↪case_latency, worst_case_latency))
```

The corresponding output is:

```
path P1 e2e latency. best case: 10, worst case: 32
path P2 e2e latency. best case: 8, worst case: 27
```

2.2.4 Step 2: Refining the Analysis

In this step, we show how analysis and propagation methods can be replaced in order to apply an improved analysis. More precisely, we want to exploit inter-event stream correlations that result from the SPNP scheduling on the bus as published in [Rox2010].

System Model

We use the same system model as before but replace the scheduler on CPU2 by *pycpa.schedulers.SPPSchedulerCorrelatedRox*.

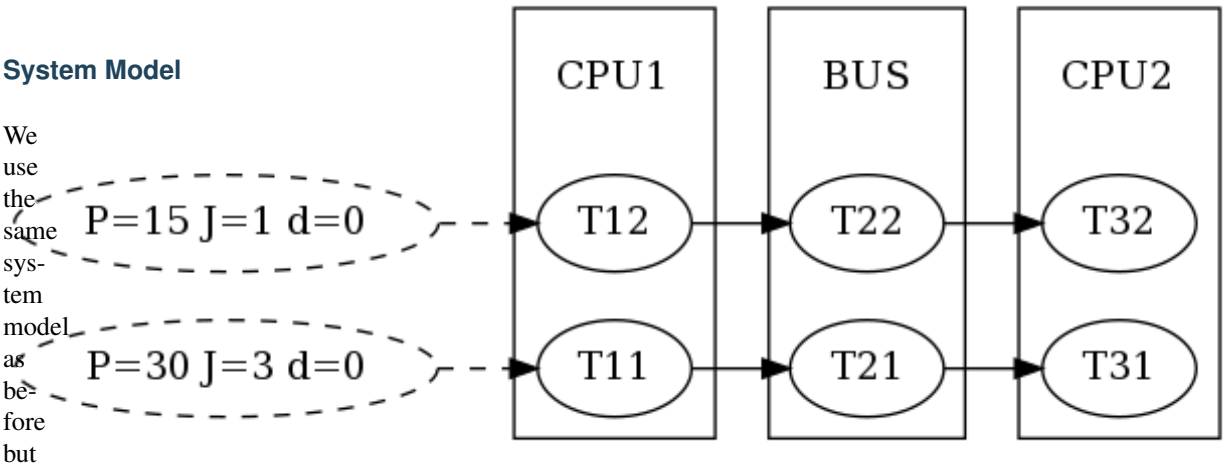


Fig. 2: Refining the Analysis

```
    r3.scheduler = schedulers.SPPSchedulerCorrelatedRox()
```

This scheduler exploits inter-event stream correlations that are accessed via the `correlated_dmin()` function of the input event models. It therefore requires this function to be present for all event models on this resource (CPU2). We achieve this by replacing the propagation method by *pycpa.propagation.SPNPBusyWindowPropagationEventModel* for alls tasks on the bus.

```
    for t in r2.tasks:
        t.OutEventModelClass = propagation.SPNPBusyWindowPropagationEventModel
```

This results in the following analysis output:

```
Performing analysis of system 'step2'
Result:
T21: wcrt=11
    b_wcrt=blocker:9, q*WCET:1*2=2
T22: wcrt=11
    b_wcrt=T21:eta*WCET:1*2=2, blocker:0, q*WCET:1*9=9
T11: wcrt=10
    b_wcrt=q*WCET:1*10=10
T12: wcrt=13
    b_wcrt=T11:eta*WCET:1*10=10, q*WCET:1*3=3
T31: wcrt=5
    b_wcrt=q*WCET:1*5=5
T32: wcrt=3
    b_wcrt=q*WCET:1*3=3
path P1 e2e latency. best case: 10, worst case: 26
path P2 e2e latency. best case: 8, worst case: 27
```

You can see that the WCRT of T31 improved from 11 to 5.

2.2.5 Step 3: Junctions and Forks

In this step, we illustrate the use of junctions and forks. Junctions need to be inserted to allow combining multiple input event streams according to a given strategy. Forks can be used if a task has multiple dependent tasks (successors). A customized fork strategy can be used if different event models shall be propagated to these tasks (e.g. in case of hierarchical event streams [Rox2008]). In this example (see figure), we model the scenario that T12 and T13 produce data that is transmitted by the same bus message which is received by the RX task on CPU2. Depending on whether T12 or T13 issued the message, T32 or T33 will be activated respectively. Hence, junctions and forks enable modelling complex scenarios such as multiplexing and demultiplexing of messages as in [Thiele2015].

System Model

Again, we start with the same system model as presented in the base scenario but remove the dependencies between T12, T22 and T32 by resetting `next_tasks` attribute. Note that modifying a system model in such a way is not recommended but used for the sake of brevity in this tutorial.

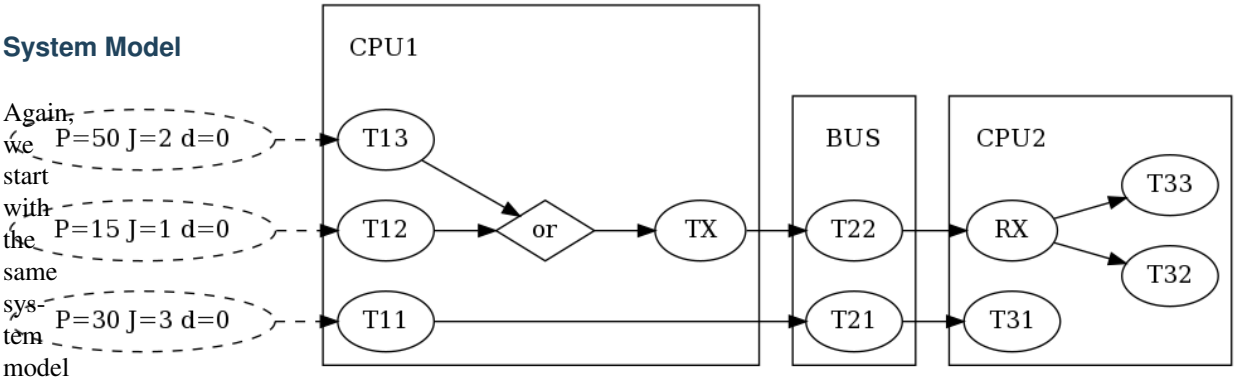


Fig. 3: Junctions and forks

```
    # remove links between t12, t22 and t32
    t12.next_tasks = set()
    t22.next_tasks = set()
```

Next, we add the additional tasks T31 and TX to CPU1 and specify an input event model for T31.

```
    # add one more task to R1
    t13 = r1.bind_task(model.Task("T13", wcet=5, bcet=2, scheduling_parameter=4))
    t13.in_event_model = model.PJdEventModel(P=50, J=2)
    # add TX task on R1
    ttx = r1.bind_task(model.Task("TX", wcet=2, bcet=1, scheduling_parameter=1))
```

Now, we need a junction in order to combine the output event models of T12 and T13 into and provide the input event model of TX. This is achieved by registering a *pycpa.model.Junction* to the system via *pycpa.model.System.bind_junction()*. A junction is assigned a name and a strategy that derives from *pycpa.analysis.JunctionStrategy*. Some strategies are already defined in the *pycpa.junctions* module. Here, we use the *pycpa.junctions.ORJoin* in order to model that TX will be activated whenever T12 or T13 produce an output event.

```
    # add OR junction to system
    j1 = s.bind_junction(model.Junction(name="J1", strategy=junctions.ORJoin()))
```

Of course, we also need to add the corresponding task links.

```
    # link t12 and t13 to junction
    t12.link_dependent_task(j1)
    t13.link_dependent_task(j1)

    # link junction to tx
    j1.link_dependent_task(ttx)
```

On CPU2, we also need to add new tasks: T31 and RX. More specifically, we add RX as a *pycpa.model.Fork* which inherits from *pycpa.model.Task*. A fork also requires a strategy. Here, we use `PathJitterForkStrategy` that we explain later in *Writing a Fork Strategy*.

Before that, let us register the missing task links.

```
    # link rx to t32 and t33
    trx.link_dependent_task(t32)
    trx.link_dependent_task(t33)

    # link tx to t22 to rx
    ttx.link_dependent_task(t22).link_dependent_task(trx)
```

A fork also allows adding a mapping from its dependent tasks to for instance an identifier or an object that will be used by the fork strategy to distinguish the extracted event models. We use this in order to map the tasks T32 and T33 to T12 and T13 respectively.

```
    # map source and destination tasks (used by fork strategy)
    trx.map_task(t32, t12)
    trx.map_task(t33, t13)
```

Writing a Fork Strategy

In this example, we want to extract separate input event models for T32 and T33 as T32 (T33) will only be activated by messages from T12 (T13). This can be achieved by encapsulating several inner event streams into an outer (hierarchical) event streams as presented in [Rox2008]. Basically, the jitter that the outer event stream experiences can be applied to the inner event streams. Hence, we need to write a fork strategy that extract the inner (original) event streams before their combination by the junction and applies the path jitter that has been accumulated from the junction to the fork. A fork strategy must implement the function `output_event_model()` which returns the output event model for a given fork and one of its dependent tasks. Our fork strategy uses the previously specified mapping to get the corresponding source task (i.e. T12 and T13) and creates the path object via *pycpa.util.get_path()*. The jitter propagation is then implemented by inheriting from *pycpa.propagation.JitterPropagationEventModel* but using the path jitter (worst-case latency - best-case latency) instead of the response-time jitter (wcrt-bcrt).

```
    class PathJitterForkStrategy(object):

        class PathJitterPropagationEventModel(propagation.
↪JitterPropagationEventModel):
            """ Derive an output event model from an in_event_model of the given task␣
↪and
                the end-to-end jitter along the given path.
            """
            def __init__(self, task, task_results, path):
                self.task = task
                path_result = path_analysis.end_to_end_latency(path, task_results, 1)
                self.resp_jitter = path_result[1] - path_result[0]
                self.dmin = 0
                self.nonrecursive = True
```

(continues on next page)

(continued from previous page)

```
                name = task.in_event_model.__description__ + "+J=" + \
                    str(self.resp_jitter) + ",dmin=" + str(self.dmin)

                model.EventModel.__init__(self,name,task.in_event_model.container)

                assert self.resp_jitter >= 0, 'response time jitter must be positive'

        def __init__(self):
            self.name = "Fork"

        def output_event_model(self, fork, dst_task, task_results):
            src_task = fork.get_mapping(dst_task)
            p = model.Path(src_task.name + " -> " + fork.name, util.get_path(src_task,
↪ fork))
            return PathJitterForkStrategy.PathJitterPropagationEventModel(src_task,␣
↪task_results, p)
```

Analysis

When running the analysis, we obtain the following output:

```
Performing analysis of system 'step3'
Result:
T21: wcrt=11
    b_wcrt=blocker:9, q*WCET:1*2=2
T22: wcrt=11
    b_wcrt=T21:eta*WCET:1*2=2, blocker:0, q*WCET:1*9=9
T11: wcrt=20
    b_wcrt=TX:eta*WCET:5*2=10, q*WCET:1*10=10
T12: wcrt=26
    b_wcrt=T11:eta*WCET:2*10=20, TX:eta*WCET:7*2=14, q*WCET:2*3=6
T13: wcrt=55
    b_wcrt=T11:eta*WCET:2*10=20, T12:eta*WCET:4*3=12, TX:eta*WCET:9*2=18, q*WCET:1*5=5
TX: wcrt=6
    b_wcrt=q*WCET:4*2=8
RX: wcrt=2
    b_wcrt=q*WCET:1*2=2
T31: wcrt=43
    b_wcrt=RX:eta*WCET:9*2=18, T32:eta*WCET:6*3=18, q*WCET:2*5=10
T32: wcrt=15
    b_wcrt=RX:eta*WCET:3*2=6, q*WCET:3*3=9
T33: wcrt=76
    b_wcrt=RX:eta*WCET:11*2=22, T31:eta*WCET:4*5=20, T32:eta*WCET:8*3=24,␣
↪q*WCET:2*5=10
```

Plotting Event Models

Now, we are interested in the event models that were extracted by the fork. For this, we use the *pycpa.plot* module to plot and compare the input event model of T12 and T32.

```
    plot_in = [t12, t32, ttx]
```

```
    # plot input event models of selected tasks
    for t in plot_in:
        plot.plot_event_model(t.in_event_model, 7, separate_plots=False, file_format=
↪'pdf', file_prefix='event-model-%s'
                % t.name, ticks_at_steps=False)
```

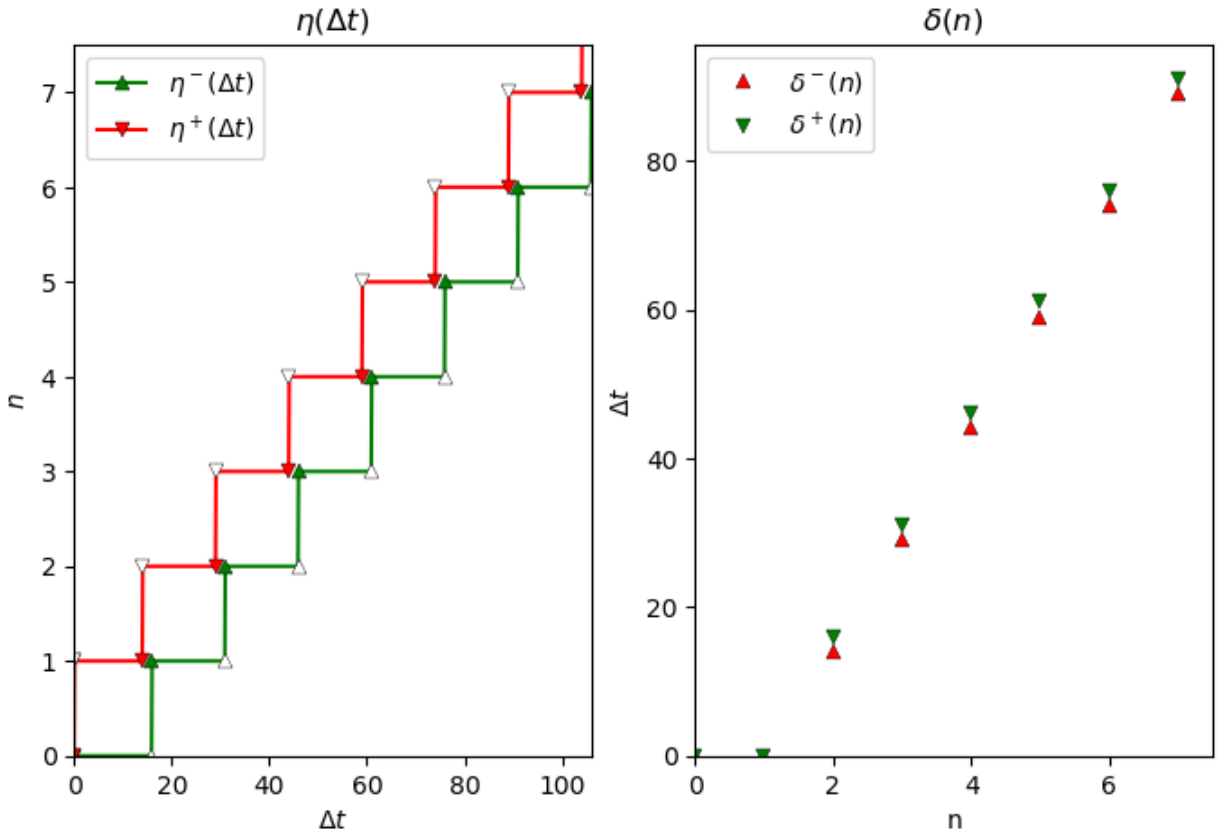


Fig. 4: Input event model of T12.

2.2.6 Step 4: Cause-Effect Chains

In this step, we demonstrate how we can compute end-to-end latencies for cause-effect chains. In contrast to a path, which describes an event stream across a chains of (dependent) tasks, a cause-effect chain describes a sequence of independently (time-)triggered tasks. In both cases, data is processed by a sequence of tasks but with different communication styles between the tasks.

We modify the base scenario by

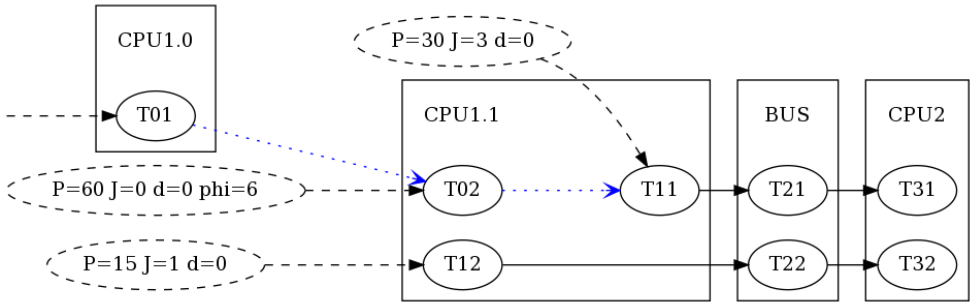


Fig. 6: Cause-Effect Chains

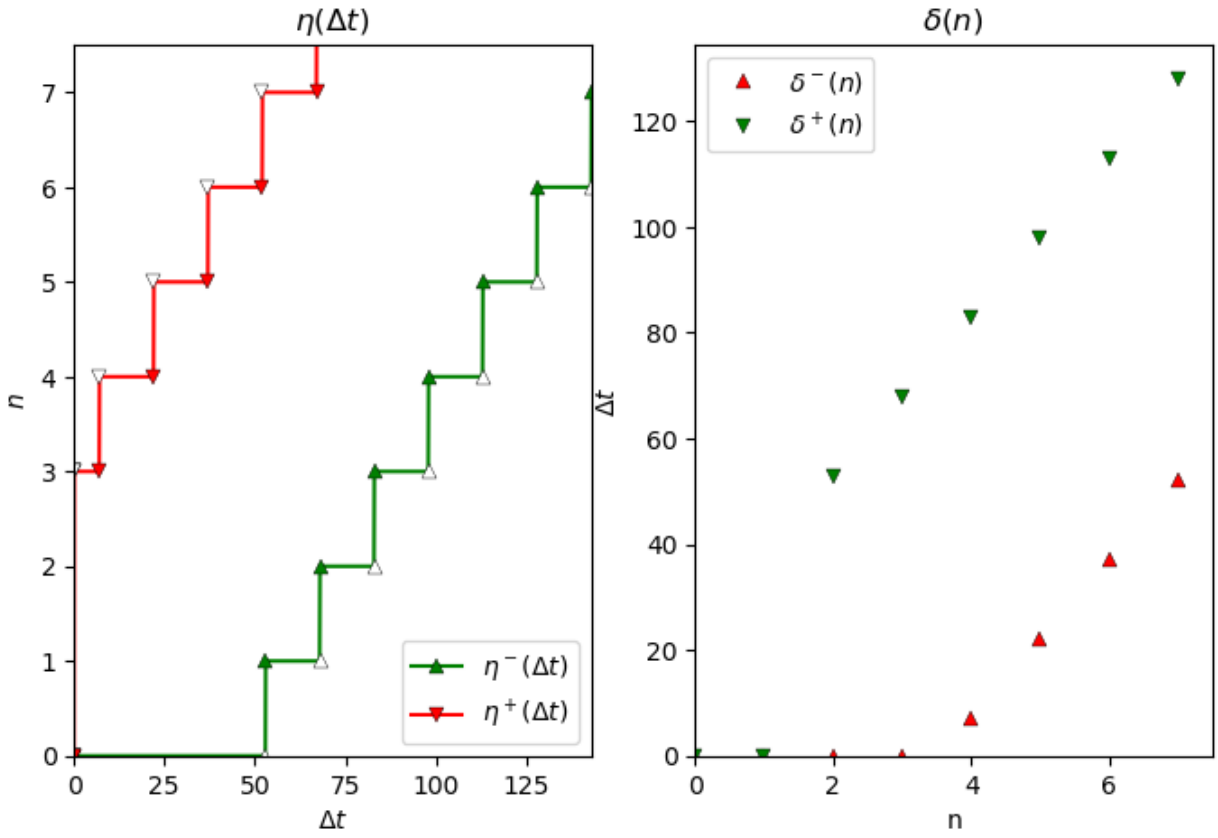


Fig. 5: Input event model of T32.

mov-
ing
from
single-
core
CPUs
to multiple cores per CPU. More precisely, we added one core to CPU1 as illustrated in the figure to the right:

```
    r1.name = 'CPU1.1'
    r10 = s.bind_resource(model.Resource("CPU1.0", schedulers.SPPScheduler()))
```

We also a new task to both cores:

```
    t01 = r10.bind_task(model.Task("T01", wcet=5, bcet=2, scheduling_parameter=1))
    t02 =  r1.bind_task(model.Task("T02", wcet=5, bcet=2, scheduling_parameter=4))

    t01.in_event_model = model.PJdEventModel(P=10, phi=0)
    t02.in_event_model = model.PJdEventModel(P=60, phi=6)
```

```
    t01 = r10.bind_task(model.Task("T01", wcet=5, bcet=2, scheduling_parameter=1))
    t02 =  r1.bind_task(model.Task("T02", wcet=5, bcet=2, scheduling_parameter=4))

    t01.in_event_model = model.PJdEventModel(P=10, phi=0)
    t02.in_event_model = model.PJdEventModel(P=60, phi=6)
```

Now we define an effect chain comprising T01, T02 and T11.

```
    chains = [ model.EffectChain(name='Chain1', tasks=[t01, t02, t11]) ]
```

Note that every task in the effect chain has its own periodic input event model. In contrast to activation dependencies (solid black arrows in the figure), the data dependencies within the effect chain are illustrated by blue dotted arrows.

Analysis

The effect chain analysis is performed similar to the path analysis. Note that there are two different latency semantics: reaction time and data age. Here, we are interested in the data age.

```
    # perform effect-chain analysis
    for c in chains:
        details = dict()
        data_age = path_analysis.cause_effect_chain_data_age(c, task_results, details)
        print("chain %s data age: %d" % (c.name, data_age))
        print("  %s" % str(details))
```

When running the analysis, we obtain the following output:

```
Performing analysis of system 'step4'
Result:
T21: wcrt=11
    b_wcrt=blocker:9, q*WCET:1*2=2
T22: wcrt=11
    b_wcrt=T21:eta*WCET:1*2=2, blocker:0, q*WCET:1*9=9
T01: wcrt=5
    b_wcrt=q*WCET:1*5=5
T02: wcrt=21
    b_wcrt=T11:eta*WCET:1*10=10, T12:eta*WCET:2*3=6, q*WCET:1*5=5
```

(continues on next page)

(continued from previous page)

```
T11: wcrt=10
    b_wcrt=q*WCET:1*10=10
T12: wcrt=13
    b_wcrt=T11:eta*WCET:1*10=10, q*WCET:1*3=3
T31: wcrt=11
    b_wcrt=T32:eta*WCET:2*3=6, q*WCET:1*5=5
T32: wcrt=3
    b_wcrt=q*WCET:1*3=3
path P1 e2e latency. best case: 10, worst case: 32
path P2 e2e latency. best case: 8, worst case: 27
chain Chain1 data age: 116
  {'T01-PHI+J': 0, 'T01-T02-delay': 23, 'T01-WCRT': 5, 'T02-T11-delay': 57, 'T02-WCRT
↪': 21, 'T11-WCRT': 10}
```

2.2.7 Step 5: Complex Run-Time Environments

It has been shown that CPA may provide very conservative results if a lot of task dependencies are present on a single resource [Schlatow2016]. The general idea to mitigate this is to only use event model propagation at resource boundaries as illustrated in the figure to the right. On the resource itself, we end up with task chains that can be analysed as a whole with the busy-window approach (see [Schlatow2016], [Schlatow2017]).

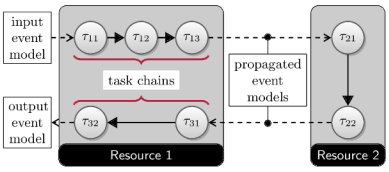


Fig. 7: Task Chains

The implementation of this approach is available as an extension to the pyCPA core at https://github.com/IDA-TUBS/pycpa_taskchain. It replaces the *pycpa.model.Resource* with a *TaskchainResource* and also the Scheduler with an appropriate implementation.

Hence, we need to import the modules as follows:

```
from taskchain import model as tc_model
from taskchain import schedulers as tc_schedulers
```

We then model the scenario depicted in the figure as follows:

```
    s = model.System(name='step5')

    # add two resources (CPUs) to the system
    # and register the static priority preemptive scheduler
    r1 = s.bind_resource(tc_model.TaskchainResource(
↪"Resource 1", tc_schedulers.SPPSchedulerSync()))
    r2 = s.bind_resource(tc_model.TaskchainResource(
↪"Resource 2", tc_schedulers.SPPSchedulerSync()))

    # create and bind tasks to r1
    t11 = r1.bind_task(model.
↪Task("T11", wcet=10, bcet=1, scheduling_parameter=1))
    t12 = r1.bind_task(model.
↪Task("T12", wcet=2, bcet=2, scheduling_parameter=3))
    t13 = r1.bind_task(model.
↪Task("T13", wcet=4, bcet=2, scheduling_parameter=6))

    t31 = r1.bind_task(model.
↪Task("T31", wcet=5, bcet=3, scheduling_parameter=4))
    t32 = r1.bind_task(model.
↪Task("T32", wcet=5, bcet=3, scheduling_parameter=2))
```

(continues on next page)

(continued from previous page)

```
    t21 = r2.bind_task(model.
↪Task("T21", wcet=3, bcet=1, scheduling_parameter=2))
    t22 = r2.bind_task(model.
↪Task("T22", wcet=9, bcet=4, scheduling_parameter=2))

    # specify precedence constraints
    t11.link_dependent_task(t12).
↪link_dependent_task(t13).link_dependent_task(t21).\
            link_dependent_task(t22).
↪link_dependent_task(t31).link_dependent_task(t32)

    # register a periodic with jitter event model for T11
    t11.in_event_model = model.PJdEventModel(P=50, J=5)

    # register task chains
    c1 = r1.bind_
↪taskchain(tc_model.Taskchain("C1", [t11, t12, t13]))
    c2 =␣
↪r2.bind_taskchain(tc_model.Taskchain("C2", [t21, t22]))
    c3 =␣
↪r1.bind_taskchain(tc_model.Taskchain("C3", [t31, t32]))

    # register a path
    s1 = s.bind_path(model.
↪Path("S1", [t11, t12, t13, t21, t22, t31, t32]))
```

When running the analysis, we get the task-chain response time results as the results for the last task in each chain:

```
Performing analysis of system 'step5'
Result:
T13: wcrt=26
    b_wcrt=T11:q*WCET:1*10=10, T12:q*WCET:1*2=2,␣
↪T13:q*WCET:1*4=4, T31:eta*WCET:1*5=5, T32:eta*WCET:1*5=5
T32: wcrt=22
    b_wcrt=T11:WCET:10, T12:WCET:2, T31:q*WCET:1*5=5,␣
↪T32:q*WCET:1*5=5
T22: wcrt=12
    b_wcrt=T21:q*WCET:1*3=3, T22:q*WCET:1*9=9
Warning: no task_results for task T11
Warning: no task_results for task T12
Warning: no task_results for task T21
Warning: no task_results for task T31
path S1 e2e latency. best case: 16, worst case: 60
```

CHAPTER 3

Command line switches

These are the default command line switches, available in every pyCPA application. The example pyCPA applications and your own application potentially add some more options.

--max_iterations <int>
: Maximum number of iterations in a local analysis

--max_window <int>
: Maximum busy window length in a local analysis

--backlog
: Compute the worst-case backlog.

--e2e_improved
: enable improved end to end analysis (experimental)

--nocaching
: disable event-model caching.

--show
: Show plots (interactive).

--propagation <method>
: Event model propagation method (jitter, jitter_dmin, jitter_offset, busy_window). default is busy_window

--verbose
: be more talkative.

CHAPTER 4

List of Modules

This is a subset of modules contained in pyCPA. The modules are loosely divided into the following categories:

4.1 pyCPA core modules

4.1.1 Model Module

Authors

- Jonas Diemer
- Philip Axer
- Johannes Schlatow

Description

It should be imported in scripts that do the analysis. We model systems composed of resources and tasks. Tasks are activated by events, modeled as event models. The general System Model is described in Section 3.6.1 in [Jersak2005] or Section 3.1 in [Henia2005].

class pycpa.model.**CTEventModel**(*c*, *T*, *dmin=1*, *name='min'*, ***kwargs*)

c events every T time event model.

set_c_in_T(*c*, *T*, *dmin=1*)

Sets the event-model to a periodic Task with period T and c activations per period. No minimum arrival rate is assumed (delta_plus = infinity)! Cf. Equation 1 in [Diemer2010].

class pycpa.model.**ConstraintsManager**
This class manages all system-wide constraints such as deadlines, buffersizes and more.

add_backlog_constraint(*task*, *size*)
adds a buffer size constraint backlog must be less or equal than size

add_load_constraint(*resource*, *load*)
adds a resource load constraint actual load on the specified resource must be less or equal than load

add_path_constraint(*path*, *deadline*, *n=1*)
adds a path latency constraint latency for n events must be less or equal than deadline

add_wcrt_constraint(*task*, *deadline*)
adds a local task deadline constraint wcrt must be less or equal than deadline

class pycpa.model.**EffectChain**(*name*, *tasks=None*)
An cause-effect chain describes a (functional) chain of independent tasks. All tasks within a chain are time-triggered and hence sample their input data independently.

task_sequence(*writers_only=False*)
Generates and returns the sequence of reader/writer tasks in the form of [reader_0, writer_0, reader_1, writer_1,...].

A task in this sequence therefore acts either as a reader or a writer. Tasks at odd positions in this sequence are readers while tasks at even positions are writers.

Parameters **writers_only** (*boolean*) – if true, only include writer tasks in sequence (omit readers)

class pycpa.model.**EventModel**(*name='min'*, *container={}*, ***kwargs*)
The event model describing the activation of tasks as described in [Jersak2005], [Richter2005], [Henia2005]. Internally, we use $\delta^-(n)$ and $\delta^+(n)$, which represent the minimum/maximum time window containing n events. They can be transformed into $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ which represent the maximum/minimum number of events arriving within Δt.

delta_min(*n*)
Delta-minus Function Return the minimum time interval between the first and the last event of any series of n events. This is actually a wrapper to allow caching of delta functions.

static delta_min_from_eta_plus(*etaplus_func*)
Delta-minus Function Return the minimum time window containing n activations. The delta_minus-function is derived from the eta_plus-function. This function is rarely needed, as EventModels are represented by delta-functions internally. Equation 3.7 from [Schliecker2011].

delta_plus(*n*)
Delta-plus Function Return the maximum time interval between the first and the last event of any series of n events. This is actually a wrapper to allow caching of delta functions.

static delta_plus_from_eta_min(*etamin_func*)
Delta-plus Function Return the maximum time window containing n activations. The delta_plus-function is derived from the eta_minus-function. This function is rarely needed, as EventModels are represented by delta-functions internally. Equation 3.8 from [Schliecker2011].

eta_min(*w*)
Eta-minus Function Return the minimum number of events in a time window w. Derived from Equation 3.6 from [Schliecker2011], but different, as Eq. 3.6 is wrong.

eta_min_closed(*w*)
Eta-minus Function Return the minimum number of events in a time window w. Using CLOSED intevals

eta_plus(*w*)
Eta-plus Function Return the maximum number of events in a time window w. Derived from Equation 3.5 from [Schliecker2011], but assuming half-open intervals for w as defined in [Richter2005].

eta_plus_closed(*w*)
Eta-plus Function Return the maximum number of events in a time window w. Derived from Equation 3.5 from [Schliecker2011], but assuming CLOSED intervals for w as defined in [Richter2005].

This is technically identical to eta_plus(w + EPSILON), but the use of epsilon has issues with float precision, as w+EPSILON == w for large w and small Epsilon (e.g. 40000000+1e-9)

load(*accuracy=1000*)
Returns the asymptotic load, i.e. the avg. number of events per time

class pycpa.model.**Fork**(*name*, *strategy=<pycpa.model.StandardForkStrategy object>*, **args*, ***kwargs*)
A Fork allows the modification (determined by the assigned strategy) of output event models dependent on the destination task.

clean()
Cleans all intermediate analysis results

get_mapping(*dst_task*)
returns the identifier mapped to dst_task (or raises KeyError)

map_task(*dst_task*, *identifier*)
maps an identifier to dst_task

class pycpa.model.**Junction**(*name='unknown'*, *strategy=None*)
A junction combines multiple event models into one output event model This is used to model multi-input tasks. Typical semantics are "and" and "or" strategies. See Chapter 4 in [Jersak2005] for definitions and details.

clean()
mark event models as invalid

map_task(*src_task*, *identifier*)
maps an identifier to src_task

class pycpa.model.**LimitedDeltaEventModel**(*limited_delta_min_func=None*, *limited_delta_plus_func=None*, *limit_q_min=inf*, *limit_q_plus=inf*, *min_additive=<function recursive_min_additive>*, *max_additive=<function recursive_max_additive>*, *name='min'*, ***kwargs*)
User supplied event model on a limited delta domain.

set_limited_delta(*limited_delta_min_func*, *limited_delta_plus_func*, *limit_q_min=inf*, *limit_q_plus=inf*, *min_additive=<function recursive_min_additive>*, *max_additive=<function recursive_max_additive>*)
Sets the event model to an arbitrary function specified by limited_delta_min_func and limited_delta_plus_func. Contrary to directly setting deltamin_func and deltaplus_func, the given functions are only valid in a limited domain [0, limit_q_min] and [0, limit_q_plus] respectively. For values of q beyond this range, a conservative extension (additive extension) is used. You can also supply a list() object to this function by using lambda x: limited_delta_min_list[x]

class pycpa.model.**Mutex**(*name=None*)
A mutually-exclusive shared Resource. Shared resources create timing interferences between tasks which may be executed on different resources (e.g. multi-core CPU) but require access to a common resource (e.g. shared main memory) to execute. See e.g. Chapter 5 in [Schliecker2011].

class pycpa.model.**PJdEventModel**(*P=0*, *J=0*, *dmin=0*, *phi=0*, *name='min'*, ***kwargs*)
A periodic, jitter, min-distance event model.

set_PJd(*P*, *J=0*, *dmin=0*, *phi=0*, *early_arrival=False*)
Sets the event model to a periodic activation with jitter and minimum distance. Equations 1 and 2 from [Schliecker2008].

class pycpa.model.**Path**(*name*, *tasks=None*)
A Path describes a (event) chain of tasks. Required for path analysis (e.g. end-to-end latency). The information stored in Path classes could be derived from the task graph (see Task.next_tasks and Task.prev_task), but having redundancy here is more flexible (e.g. path analysis may only be interesting for some task chains).

print_all()
Print all tasks in Path. Uses __str__()

class pycpa.model.**Resource**(*name=None*, *scheduler=None*, ***kwargs*)
A Resource provides service to tasks.

bind_task(*t*)
Bind task t to resource Returns t

load(*accuracy=10000*)
returns the asymptotic load

unmap_tasks()
unmap all tasks from this resource

class pycpa.model.**StandardForkStrategy**
Standard fork strategy: propagates unmodified output event model to all tasks.

output_event_model(*fork*, *dst_task=None*, *task_results=None*)
This strategy does not distinguish between destination tasks.

Parameters

- **fork** (model.Task) – Fork from which to take the output event model.
- **dst_task** – destination task

class pycpa.model.**System**(*name=''*)
The System is the top-level entity of the system model. It contains resources, junctions, tasks and paths.

bind_junction(*j*)
Registers a junction object in the System. Logically, the junction neither belongs to a system nor to a resource, for sake of convenience we associate junctions with the system.

bind_path(*path*)
Add a Path to the System

bind_resource(*r*)
Add a Resource to the System

print_subgraphs()
enumerate all subgraphs of the application graph. if a subgraph is not well-formed (e.g. a source is missing), this algorithm may not work correctly (it will eventually produce to many subgraphs)

class pycpa.model.**Task**(*name*, **args*, ***kwargs*)
A Task is an entity which is mapped on a resource and consumes service. Tasks are activated by events, which are described by EventModel. Events are queued in FIFO order at the input of the task, see Section 3.6.1 in [Jersak2005] or Section 3.1 in [Henia2005].

bind_mutex(*m*)
Bind a Task t to a Mutex r

bind_resource(*r*)
Bind a Task t to a Resource/Mutex r

clean()
Cleans all intermediate analysis results

get_mutex_interferers()
returns the set of tasks sharing the same Mutex as Task ti excluding ti itself

get_resource_interferers()
returns the set of tasks sharing the same Resource as Task ti excluding ti itself

link_dependent_task(*t*)
Link a dependent task t to the task The dependent task t is activated by the completion of the task.

This method returns the t argument, which enables elegant task linking. E.g. to link t0 -> t1 -> t2, call: t0.link_dependent_task(t1).link_dependent_task(t2)

load(*accuracy=100*)
Returns the load generated by this task

unbind_mutex()
Remove a task fromk its mutex

unbind_resource()
Remove a task from its resource

class pycpa.model.**TraceEventModel**(*trace_points=[]*, *min_sample_size=20*, *min_additive=<function recursive_min_additive>*, *max_additive=<function recursive_max_additive>*, *name='min'*, ***kwargs*)

set_limited_trace(*trace_points*, *min_sample_size=20*, *min_additive=<function recursive_min_additive>*, *max_additive=<function recursive_max_additive>*)
Compute a pseudo-conservative event model from a given trace (e.g. from SymTA/S TraceAnalyzer or similar). trace_points must be a list of integers encoding the arrival time of an event. The algorithm will compute delta_min and delta_plus based on the trace by evaluating all candidates. min_sample_size is the minimum amount of candidates that must be available to derive a representative deltamin/deltaplus

4.1.2 Junctions Module

Authors

- Johannes Schlatow

Description

Local model propagation functions (junctions)

class pycpa.junctions.**ANDJoin**
Compute output event models for an AND junction. This corresponds to Lemma 4.2 in [Jersak2005].

class pycpa.junctions.**OREventModel**(*in_event_models*)
Compute output event model for an OR junction. This corresponds to Section 4.2, Equations 4.11 and 4.12 in [Jersak2005].

eta_min(*w*)
Eta-minus Function Return the minimum number of events in a time window w. Derived from Equation 3.6 from [Schliecker2011], but different, as Eq. 3.6 is wrong.

eta_min_closed(*w*)
Eta-minus Function Return the minimum number of events in a time window w. Using CLOSED intevals

eta_plus(*w*)
Eta-plus Function Return the maximum number of events in a time window w. Derived from Equation 3.5 from [Schliecker2011], but assuming half-open intervals for w as defined in [Richter2005].

eta_plus_closed(*w*)
Eta-plus Function Return the maximum number of events in a time window w. Derived from Equation 3.5 from [Schliecker2011], but assuming CLOSED intervals for w as defined in [Richter2005].

This is technically identical to eta_plus(w + EPSILON), but the use of epsilon has issues with float precision, as w+EPSILON == w for large w and small Epsilon (e.g. 40000000+1e-9)

class pycpa.junctions.**ORJoin**
Compute output event models for an OR junction (see [Jersak2005]).

class pycpa.junctions.**SampledInput**
Uses a fixed event model (trigger) as output event model. Serves as a workaround for defining a Path over time-triggered tasks. The sampling delay is conservatively computed and automatically added to the path latency.

4.1.3 Analysis Module

Generic Compositional Performance Analysis Algorithms

Authors

- Jonas Diemer
- Philip Axer
- Johannes Schlatow

Description

This module contains methods for real-time scheduling analysis. It should be imported in scripts that do the analysis.

class pycpa.analysis.**GlobalAnalysisState**(*system*, *task_results*)
Everything that is persistent during one analysis run is stored here. At the moment this is only the list of dirty tasks. Half the anlysis context is stored in the Task class itself!

clean()
Clear all intermediate analysis data

clean_analysis_state()
Clean the analysis state

get_dependent_tasks(*task*)
Return all tasks which immediately depend on task.

class pycpa.analysis.**JunctionStrategy**
This class encapsulates the junction-specific analysis

propagate(*junction*, *task_results*)
Propagate event model over a junction

reload_in_event_models(*junction*, *task_results*, *non_cycle_prev*)
Helper function, reloads input event models of junction from tasks in non_cycle_prev

exception pycpa.analysis.**NotSchedulableException**(*value*)
Thrown if the system is not schedulable

class pycpa.analysis.**Scheduler**
This class encapsulates the scheduler-specific analysis

b_min(*task*, *q*)
Minimum Busy-Time for q activations of a task.

This default implementation should be conservative for all schedulers but can be overridden for improving the results with scheduler knowledge.

Parameters

- **task** (model.Task) – the analyzed task
- **q** (*integer*) – the number of activations

Return type integer (max. busy-time for q activations)

b_plus(*task*, *q*, *details=None*, ***kwargs*)
Maximum Busy-Time for q activations of a task.

This default implementation assumes that all other tasks disturb the task under consideration, which is the behavior of a "random priority preemptive" scheduler or a "least-remaining-load-last" scheduler. This is a conservative bound for all work-conserving schedulers.

Warning: This default implementation should be overridden for any scheduler.

Parameters

- **task** (model.Task) – the analyzed task
- **q** (*boolean*) – the number of activations
- **details** – reference to a dict of details on the busy window (instead of busy time)

Return type integer (max. busy-time for q activations)

compute_bcrt(*task*, *task_results=None*)
Return the best-case response time for q activations of a task. Convenience function which calls the minimum busy-time. The bcrt is also stored in task_results.

compute_max_backlog(*task*, *task_results*, *output_delay=0*)
Compute the maximum backlog of Task t. This is the maximum number of outstanding activations. Implemented as shown in Eq.17 of [Diemer2012].

compute_service(*task*, *t*)
Computes the worst-case service a Task receives within an interval of t, i.e. how many activations are at least computed within t.

Call System.analyze() first if service depends on other resources to make sure all event models are up-to-date! This service is higher than the maximum arrival curve (requested service) of the task if the task is schedulable.

compute_wcrt(*task*, *task_results=None*)
Compute the worst-case response time of Task

> **Warning:** This default implementation works only for certain schedulers and must be overridden otherwise.

Parameters

- **task** (model.Task) – the analyzed task
- **task_results** (*dict (*analysis.TaskResult*)*) – dictionary which stores analysis results

Return type integer (worst-case response time)

For this, we construct busy windows for q=1, 2, ... task activations (see [Lehoczky1990]) and iterate until a stop condition (e.g. resource idle again). The response time is then the maximum time difference between the arrival and the completion of q events. See also Equations 2.3, 2.4, 2.5 in [Richter2005]. Should not be called directly (use System.analyze() instead).

stopping_condition(*task*, *q*, *w*)
Return true if a sufficient number of activations q have been evaluated for a task during the busy-time w.

This default implementation continues analysis as long as there are new activations of the task within its current busy window.

> **Warning:** This default implementation works only for certain schedulers (e.g. SPP) and must be overridden otherwise.

Parameters

- **task** (model.Task) – the analyzed task
- **q** (*integer*) – the number of activations
- **w** (*integer*) – the current busy-time

Return type integer (max. busy-time for q activations)

class pycpa.analysis.**TaskResult**
This class stores all analysis results for a single task

b_wcrt_str()
Returns a string with the components of b_wcrt sorted alphabetically

clean()
Clean up

exception pycpa.analysis.**TimeoutException**(*value*)
Thrown if the analysis timed out

pycpa.analysis.**analyze_system**(*system*, *task_results=None*, *only_dependent_tasks=False*, *progress_hook=None*, ***kwargs*)
Analyze all tasks until we find a fixed point

system – the system to analyze task_results – if not None, all intermediate analysis results from a previous run are reused

Returns a dictionary with results for each task.

This based on the procedure described in Section 7.2 in [Richter2005].

pycpa.analysis.**analyze_task**(*task*, *task_results*)
Analyze Task BUT DONT propagate event model. This is the "local analysis step", see Section 7.1.4 in [Richter2005].

pycpa.analysis.**check_violations**(*constraints*, *task_results*, *wcrt=True*, *path=True*, *backlog=True*, *load=True*)
Check all if all constraints are satisfied. Returns True if there are constraint violations. :param task_results: dictionary which stores analysis results :type task_results: dict (analysis.TaskResult) :param wcrt: if True, check wcrt :param path: if True, check path latencies :param backlog: if True, check buffersized :param load: if True, check loads :rtype: boolean

pycpa.analysis.**out_event_model**(*task*, *task_results*, *dst_task=None*)
Wrapper to call the out_event_model() method of the actual propagation strategy in order to compute the output event model of a task. See Chapter 4 in [Richter2005] for an overview.

Parameters

- **task** (model.Task) – the source task
- **task_results** (*dict (analysis.TaskResult)*) – dictionary which stores analysis results
- **dst_task** (model.Task) – the destination task

4.1.4 Propagation Module

Event model propagation algorithms.

Authors

- Jonas Diemer
- Philip Axer
- Johannes Schlatow

Description

class pycpa.propagation.**BusyWindowPropagationEventModel**(*task*, *task_results*, *nonrecursive=True*)
Derive an output event model from busy window and in_event_model (used as reference). Typically provides better results than JitterPropagationEventModel.

This results from Theorems 1, 2 and 3 from [Schliecker2008].

class pycpa.propagation.**JitterBminPropagationEventModel**(*task*, *task_results*, *nonrecursive=True*)

Derive an output event model from response time jitter, the b_min as well as the in_event_model (used as reference).

Uses a reference to task.deltamin_func

bmin(*n*)

minimum production time for n events at the output

class pycpa.propagation.**JitterOffsetPropagationEventModel**(*task*, *task_results*, *nonrecursive=True*)

Derive an output event model from response time jitter and in_event_model (used as reference). Also calculates the offset attribute.

This corresponds to Equations 1 (non-recursive) and 2 (recursive from [Schliecker2009] This is equivalent to Equation 5 in [Henia2005] or Equation 4.6 in [Richter2005].

Uses a reference to task.deltamin_func

class pycpa.propagation.**JitterPropagationEventModel**(*task*, *task_results*, *nonrecursive=True*)

Derive an output event model from response time jitter and in_event_model (used as reference).

This corresponds to Equations 1 (non-recursive) and 2 (recursive from [Schliecker2009] This is equivalent to Equation 5 in [Henia2005] or Equation 4.6 in [Richter2005].

Uses a reference to task.deltamin_func

class pycpa.propagation.**OptimalPropagationEventModel**(*task*, *task_results*, *nonrecursive=True*)

Optimal event model based on jitter and busy_window propagation. For some schedulers, such as FIFO and EDF neither busy_window nor jitter propagation is optimal. This will try both and chooses the best result.

class pycpa.propagation.**SPNPBusyWindowPropagationEventModel**(*task*, *task_results*, *nonrecursive=True*)

Performs standard busy window propagation but additionally calculates the minimum distance to any preceding event of a given task.

This corresponds to Def. 2 from [Rox2010].

4.1.5 Path Analysis Module

Compositional Performance Analysis Algorithms for Path Latencies

Authors

- Jonas Diemer
- Philip Axer
- Johannes Schlatow

Description

This module contains methods for the ananlysis of path latencies. It should be imported in scripts that do the analysis.

pycpa.path_analysis.**cause_effect_chain**(*chain*, *task_results*, *details=None*, *semantics='data-age'*)
computes the data age of the given cause effect chain :param chain: model.EffectChain :param task_results: dict of analysis.TaskResult

pycpa.path_analysis.**cause_effect_chain_data_age**(*chain*, *task_results*, *details=None*)
computes the data age of the given cause effect chain :param chain: model.EffectChain :param task_results: dict of analysis.TaskResult

pycpa.path_analysis.**cause_effect_chain_reaction_time**(*chain*, *task_results*, *details=None*)
computes the data age of the given cause effect chain :param chain: model.EffectChain :param task_results: dict of analysis.TaskResult

pycpa.path_analysis.**end_to_end_latency**(*path*, *task_results*, *n=1*, *task_overhead=0*, *path_overhead=0*, ***kwargs*)
Computes the worst-/best-case e2e latency for n tokens to pass the path. The constant path.overhead is added to the best- and worst-case latencies.

Parameters

- **path** (model.Path) – the path
- **n** (*integer*) – amount of events
- **task_overhead** (*integer*) – A constant task_overhead is added once per task to both min and max latency
- **path_overhead** (*integer*) – A constant path_overhead is added once per path to both min and max latency

Return type tuple (best-case latency, worst-case latency)

pycpa.path_analysis.**end_to_end_latency_classic**(*path*, *task_results*, *n=1*, *injection_rate='max'*, ***kwargs*)
Computes the worst-/best-case end-to-end latency Assumes that all tasks in the system have successfully been analyzed. Assumes that events enter the path at maximum/minumum rate. The end-to-end latency is the sum of the individual task's worst-case response times.

This corresponds to Definition 7.3 in [Richter2005].

Parameters

- **path** (model.Path) – the path
- **n** (*integer*) – amount of events
- **injection_rate** (*string 'max' or 'min'*) – assumed injection rate is maximum or minimum

Return type tuple (best case latency, worst case latency)

pycpa.path_analysis.**end_to_end_latency_improved**(*path*, *task_results*, *n=1*, *e_0=0*, ***kwargs*)
Performs the path analysis presented in [Schliecker2009recursive], which improves results compared to end_to_end_latency() for n>1 and bursty event models. lat(n)

4.1.6 Options Module

Authors

- Jonas Diemer
- Philip Axer

Description

This module contains methods to initalize the pycpa environment. It will setup an argument parser and set up default parameters.

pycpa.options.**get_opt**(*option*)
Returns the option specified by the parameter. If called for the first time, the parsing is done.

pycpa.options.**init_pycpa**(*implicit=False*)
Initialize pyCPA. This function parses the options and prints them for reference. It is called once automatically from get_opt() or set_opt() during the beginning of the analysis. It can also be called directly to control when initialization happens in order to modify options afterwards.

pycpa.options.**pprintTable**(*out*, *table*, *column_sperator=''*, *header_separator=':'*)
Prints out a table of data, padded for alignment @param out: Output stream (file-like object) @param table: The table to print. A list of lists. Each row must have the same number of columns.

pycpa.options.**set_opt**(*option*, *value*)
Sets the option specified by the parameter to value. If called for the first time, the parsing is done.

4.1.7 Util Module

Authors

- Philip Axer
- Jonas Diemer

Description

Various utility functions

pycpa.util.**GCD**(*terms*)
Return gcd of a list of numbers.

pycpa.util.**LCM**(*terms*)
Return lcm of a list of numbers.

pycpa.util.**additive_extension**(*additive_func*, *q*, *q_max*, *cache=None*, *cache_offset=1*)
Additive extension for event models. Any sub- or super- additive function additive_func valid in the domain q in [0, q_max] is extended and the approximited value f(q) is returned. NOTE: this cannot be directly used with delta curves, since they are "1-off", thus if you supply a delta function to additive_func, note to add 1 and supply q-1. e.g. util.additive_extension(lambda x: self.delta_min(x + 1), n - 1, q_max)

pycpa.util.**bitrate_str_to_bits_per_second**(*bitrate_str*)
Convert bitrate strings like "100MBit/s" or "1 Gbit/s" to an integer representation in Bit/s.

pycpa.util.**breadth_first_search**(*task*, *func=None*, *get_reachable_tasks=<function get_next_tasks>*)
returns a set of nodes (tasks) which is reachable starting from the starting task. calls func on the first discover of a task.

get_reachable_tasks(task) specifies a function which returns all tasks considered immediately reachable for a given task.

pycpa.util.**combinations_with_replacement**(*iterable*, *r*)
combinations_with_replacement('ABC', 2) –> AA AB AC BB BC CC

pycpa.util.**cycles_to_time**(*value*, *freq*, *base_time*, *rounding='ceil'*)
Converts the cycle/bittimes to an absolute time in base_time

pycpa.util.**dijkstra**(*source*)
Calculates a distance-map from the source node based on the dijkstra algorithm The edge weight is 1 for all linked tasks

pycpa.util.**gcd**(*a*, *b*)
Return greatest common divisor using Euclid's Algorithm.

pycpa.util.**generate_distance_map**(*system*)
Precomputes a distance-map for all tasks in the system.

pycpa.util.**get_next_tasks**(*task*)
return the list of next tasks for task object. required for _breadth_first_search

pycpa.util.**get_path**(*t_src*, *t_dst*)
Find path between tasks t_src and t_dst. Returns a path as list() or None if no path was found. NOTE: There is no protection against cycles!

pycpa.util.**lcm**(*a*, *b*)
Return lowest common multiple.

pycpa.util.**recursive_max_additive**(*additive_func*, *q*, *q_max*, *cache=None*, *cache_offset=1*)
Sub-additive extension for event models. Any sub-additive function additive_func valid in the domain q in [0, q_max] is extended and the value f(q) is returned. It is optional to supply a cache dictionary for speedup.

NOTE: this cannot be directly used with delta curves, since they are "1-off", thus if you supply a delta function to additive_func, note to add 1 and supply q-1. e.g. ret = util.recursive_max_additive(lambda x: self.delta_min(x + 1), n - 1, q_max, self.delta_min_cache)

By default, the cache is filled according to the delta domain notion, so it can be used with delta-based event models. To override this behavior, change the cache_offset parameter to zero

pycpa.util.**recursive_min_additive**(*additive_func*, *q*, *q_max*, *cache=None*, *cache_offset=1*)
Super-additive extension for event models. Any additive function additive_func valid in the domain q in [0, q_max] is extended and the value f(q) is returned. It is optional to supply a cache dictionary for speedup.

NOTE: this cannot be directly used with delta curves, since they are "1-off", thus if you supply a delta function to additive_func, note to add 1 and supply q-1. e.g. ret = util.recursive_min_additive(lambda x: self.delta_plus(x + 1), n - 1, q_max, self.delta_plus_cache)

By default, the cache is filled according to the delta domain notion, so it can be used with delta-based event models. To override this behavior, change the cache_offset parameter to zero

pycpa.util.**str_to_time_base**(*unit*)
Return the time base for the string

pycpa.util.**time_base_to_str**(*t*)
Return the time base for the string

pycpa.util.**time_str_to_time**(*time_str*, *base_out*, *rounding='ceil'*)
Convert strings like "100us" or "10 ns" to an integer representation in base_out.

pycpa.util.**time_to_cycles**(*value*, *freq*, *base_time*, *rounding='ceil'*)
Converts an absolute time given in the base_time domain into cycles

pycpa.util.**time_to_time**(*value*, *base_in*, *base_out*, *rounding='ceil'*)
Convert an absolute time given in base_in to another absolute time given in base_out

pycpa.util.**uunifast**(*num_tasks*, *utilization*)
Returns a list of random utilizations, one per task [0.1, 0.23, ...] WCET and event model (i.e. PJd) must be calculated in a second step)

pycpa.util.**window**(*seq*, *n=2*)
Returns a sliding window (of width n) over data from the iterable s -> (s0,s1,...s[n-1]), (s1,s2,...,sn), ...

4.2 Schedulers (busy window functions)

4.2.1 Schedulers Module

Authors

- Jonas Diemer
- Philip Axer
- Johannes Schlatow

Description

Local analysis functions (schedulers)

class pycpa.schedulers.**CorrelatedDeltaMin**(*em*, *m*, *offset*)
Computes the correlated event model δ_j^- from Lemma 2 in [Rox2010].

class pycpa.schedulers.**RoundRobinScheduler**
Round-Robin Scheduler

task.scheduling_parameter is the respective slot size

b_plus(*task*, *q*, *details=None*, ***kwargs*)
Maximum Busy-Time for q activations of a task.

This default implementation assumes that all other tasks disturb the task under consideration, which is the behavior of a "random priority preemptive" scheduler or a "least-remaining-load-last" scheduler. This is a conservative bound for all work-conserving schedulers.

> **Warning:** This default implementation should be overridden for any scheduler.

Parameters

- **task** (model.Task) – the analyzed task
- **q** (*boolean*) – the number of activations
- **details** – reference to a dict of details on the busy window (instead of busy time)

Return type integer (max. busy-time for q activations)

class pycpa.schedulers.**SPNPScheduler**(*priority_cmp=<function <lambda>>*, *ctx_switch_overhead=0*, *cycle_time=1e-09*)

Static-Priority-Non-Preemptive Scheduler

Priority is stored in task.scheduling_parameter, by default numerically lower numbers have a higher priority

Policy for equal priority is FCFS (i.e. max. interference).

b_plus(*task*, *q*, *details=None*, ***kwargs*)

Return the maximum time required to process q activations

spnp_busy_period(*task*, *w*)

Calculated the busy period of the current task

stopping_condition(*task*, *q*, *w*)

Check if we have looked far enough compute the time the resource is busy processing q activations of task and activations of all higher priority tasks during that time Returns True if stopping-condition is satisfied, False otherwise

class pycpa.schedulers.**SPPScheduler**(*priority_cmp=<function <lambda>>*)

Static-Priority-Preemptive Scheduler

Priority is stored in task.scheduling_parameter, by default numerically lower numbers have a higher priority

Policy for equal priority is FCFS (i.e. max. interference).

b_plus(*task*, *q*, *details=None*, ***kwargs*)

This corresponds to Theorem 1 in [Lehoczky1990] or Equation 2.3 in [Richter2005].

class pycpa.schedulers.**SPPSchedulerActivationOffsets**(*priority_cmp=<function <lambda>>*)

Static-Priority-Preemptive Scheduler which considers activation offsets assuming all tasks are activated synchronously with the given offsets/phases (phi).

Assumptions:

- implicit or constrained deadlines

We exclude/shift interferers whose phase is larger than the task under analysis iff the interferers period is equal or smaller.

b_plus(*task*, *q*, *details=None*, ***kwargs*)

This corresponds to Theorem 1 in [Lehoczky1990] or Equation 2.3 in [Richter2005].

class pycpa.schedulers.**SPPSchedulerCorrelatedRox**(*priority_cmp=<function <lambda>>*)

SPP scheduler with dmin correlation. Computes the approximate response time bound as presented in [Rox2010].

b_plus(*task*, *q*, *details=None*, *task_results=None*)

This corresponds to Theorem 1 in [Lehoczky1990] or Equation 2.3 in [Richter2005].

b_plus_busy(*task*, *q*, *details=None*, *task_results=None*)

Implements Case 1 in [Rox2010].

b_plus_idle(*task*, *q*, *details=None*, *task_results=None*)

Implements Case 2 in [Rox2010].

class pycpa.schedulers.**SPPSchedulerCorrelatedRoxExact**(*priority_cmp=<function <lambda>>*)

SPP scheduler with dmin correlation based on [Rox2010]. This is the exact version which performs an extensive search of busy window candidates.

b_plus(*task*, *q*, *details=None*, *task_results=None*)

This corresponds to Theorem 1 in [Lehoczky1990] or Equation 2.3 in [Richter2005].

class pycpa.schedulers.**SPPSchedulerRoundRobin**(*priority_cmp=<function <lambda>>*)

SPP scheduler with non-preemptive round-robin policy for equal priorities

b_plus(*task*, *q*, *details=None*, ***kwargs*)

This corresponds to Theorem 1 in [Lehoczky1990] or Equation 2.3 in [Richter2005].

class pycpa.schedulers.**TDMAScheduler**

TDMA scheduler task.scheduling_parameter is the slot size of the respective task

b_plus(*task*, *q*, *details=None*, ***kwargs*)

Maximum Busy-Time for q activations of a task.

This default implementation assumes that all other tasks disturb the task under consideration, which is the behavior of a "random priority preemptive" scheduler or a "least-remaining-load-last" scheduler. This is a conservative bound for all work-conserving schedulers.

> **Warning:** This default implementation should be overridden for any scheduler.

Parameters

- **task** (model.Task) – the analyzed task
- **q** (*boolean*) – the number of activations
- **details** – reference to a dict of details on the busy window (instead of busy time)

Return type integer (max. busy-time for q activations)

4.3 Plotting related Modules

4.3.1 Plot Module

Authors

- Jonas Diemer
- Philip Axer

Description

General purpose plotting functions: * event model plotting * gantt plotting (requires the simulation engine)

pycpa.plot.**augment_range**(*plot_range*)
Adds points around every point in plot_range for accurately plotting integer-based curves

pycpa.plot.**plot_eta**(*eta*, *plot_range*, *label=None*, *color=None*, *show=False*, *filename=None*)
Plot an eta function

pycpa.plot.**plot_event_model**(*model*, *num_events*, *file_format=None*, *separate_plots=True*, *file_prefix=''*, *ticks_at_steps=False*)
Plot the Task's eta and delta_min functions. Intervals in eta are shown half-open, as defined in [Richter2005].

Parameters

- **model** (model.EventModel) – the event model
- **num_events** – Number of events to plot
- **file_format** (*string*) – the format of the file to be plotted
- **separate_plots** (*bool*) – whether eta and delta plots should be combined
- **file_prefix** (*string*) – prefix of file name of plots
- **ticks_at_steps** (*bool*) – If True, draw the x-axis ticks at steps of the functions. Otherwise, let matplotlib decide where to draw ticks.

Return type None

pycpa.plot.**plot_gantt**(*tasks*, *task_results*, *file_name=None*, *show=True*, *xlim=None*, *preemtion_bar_height=0.2*, *height=1*, *hdist=1*, *bar_linewidth=1*, *min_dist_arrows=0.2*, *plot_event_arrival=True*, *plot_activation_finishing=False*, *annotate_tasks=True*, *task=None*, *wcrt_voffset=0.5*, *annotation_offset=0.2*, *arrow_width=0.05*, *arrow_head_width=0.4*, *arrow_head_length=0.2*, *arrow_xscale=1*, *arrow_yoffset=0.1*, *xticks_only_on_changes=False*, *color_preemtion_bar='0.30'*, *color_execution_bar='lightblue'*, *title='Gantt'*, *number_xticks=20*)
Plot a gantt chart of a given task list. Execution time information is taken from the task attribute q_exec_windows which is written by the simulation framework

4.3.2 Task Graph Module

Authors

- Jonas Diemer

- Philip Axer

Description

This module contains methods to plot task/architecture graphs of your system

class pycpa.graph.**dotgraph**(***kwargs*)
Minimalistic implementation of the pygraphviz API. With this, you can write graphs to a file.

pycpa.graph.**graph_system**(*s*, *filename=None*, *layout='dot'*, *empty_resources=False*, *short_tasks=False*, *exec_times=False*, *sched_param=False*, *rankdir='LR'*, *show=False*, *dotout=None*, *use_pygraphviz=False*, *chains=[]*)
Return a graph of the system

Parameters

- **s** (model.System) – the system
- **filename** – if not None, the graph is plotted to this file
- **layout** – graphviz layout algorithm (default l'dot' works best with hierarchical graphs)
- **empty_resources** – Plot resources that have no tasks assigned
- **short_tasks** – Label tasks using "T_nn" instead of their potentially long name
- **exec_times** – Show execution times for each tasks
- **sched_param** – Show scheduling parameter for each task
- **rankdir** – Layout option for graphviz
- **show** (*boolean*) – Show plot
- **dotout** – If set, write a dot file to this filename

Return type None

4.3.3 Simulation Module

4.4 Server and Import/Export filters

4.4.1 XML RPC Module

Authors

- Philip Axer
- Jonas Diemer

Description

XML-RPC server for pyCPA. It can be used to interface pycpa with non-python (e.g. close-source) applications.

class pycpa.cparpc.**CPARPC**
Basic XML RPC Server for pyCPA.

Methods prefixed with "xmlrpc_" are actually callable from the client.

Please see *pycpa.model* for more details about the pyCPA model and *pycpa.analysis* for information about the analysis.

debug_prefix = None
Prefix for function calls in debug output

id_type = None
Specifies how unique IDs are generated

scheduling_policies = None
Dictionary of scheduler classes.

xmlrpc_analyze_system(*system_id*)
Analyze system and return a result id.

Parameters **system_id** (*string*) – ID of the system to analyze

Returns ID of a results object

Return type string

xmlrpc_assign_ct_event_model(*task_id*, *c*, *T*, *min_dist*)
Create an eventmodel and assign it to task. The event model will represent a periodic burst with c activations every T time units, with the activations in each burst being min_dist time units apart from each other.

Parameters

- **task_id** (*string*) – ID of the task
- **c** (*integer*) – Number of activations per burst
- **T** (*integer*) – Period of the bursts
- **min_dist** (*integer*) – Minimum distance between events (in unit time)

Returns 0

xmlrpc_assign_pjd_event_model(*task_id*, *period*, *jitter*, *min_dist*)
Create an eventmodel and assign it to task.

Parameters

- **task_id** (*string*) – ID of the task
- **period** (*integer*) – Period (in unit time)
- **jitter** (*integer*) – Jitter (in unit time)
- **min_dist** (*integer*) – Minimum distance between events (in unit time)

Returns 0

xmlrpc_assign_scheduler(*resource_id*, *scheduler_string*)
Assign a scheduler to a resource. See *xmlrpc_get_valid_schedulers()* for a list of valid schedulers.

Parameters

- **resource_id** (*integer*) – ID of the resource to which to assign the scheduler.
- **scheduler_string** (*string*) – Identifies the type of scheduler to set.

Returns 0 for success

xmlrpc_clear_models()
Delete all models, i.e. all systems, resources, tasks, results etc.

Returns 0

xmlrpc_end_to_end_latency(*path_id*, *results_id*, *n*)
Perform a path analysis to obtain the end-to-end latency. Requires that the system has been analyzed before to obtain the results_id.

Parameters

- **path_id** (*string*) – ID of the path
- **results_id** (*string*) – ID of the results
- **n** (*integer*) – Number of activations to obtain the latency for

Returns best- and worst-case latency for n events along path.

Return type tuple of integers

xmlrpc_get_attribute(*obj_id*, *attribute*)
Return the attribute of a task.

Parameters

- **obj_id** (*string*) – ID of the task to get the parameter from.
- **attribute** (*string.*) – Attribute to get.

Returns Value of the attribute

Return type Depends on attribute.

xmlrpc_get_task_result(*results_id*, *task_id*)
Obtain the analysis results for a task.

Parameters

- **results_id** – ID of the results object
- **task_id** (*string*) – ID of the task

Returns a dictionary of results for task_id.

Return type *pycpa.analysis.TaskResult*

xmlrpc_get_valid_schedulers()
Find out which schedulers are supported.

Returns List of valid schedulers

Return type list of strings

xmlrpc_graph_system(*system_id*, *filename*)
Generate a graph of the system (in server directory). It uses graphviz for plotting, so the 'dot' command must be in the PATH of the server environment.

Parameters

- **system_id** (*string*) – ID of the system to analyze

- **filename** (*string*) – File name (relative to server working directory) to which to store the graph.

Returns 0

xmlrpc_graph_system_dot(*system_id*, *filename*)
Generate a graph of the system in dot file format (in server directory). The resulting file can be converted using graphviz. E.g. to create a PDF, run:

dot -Tpdf <filename> -o out.pdf

Parameters

- **system_id** (*string*) – ID of the system to analyze
- **filename** (*string*) – File name (relative to server working directory) to which to write to. If empty, return dot file as string only.

Returns string representation of graph in dot format

Return type string

xmlrpc_link_task(*task_id*, *target_id*)
Make task with target_id dependent of the task with task_id.

Parameters

- **task_id** (*string*) – ID of the task that activates the target task
- **target_id** (*string*) – ID of the task that is activate by the task.

Returns 0

xmlrpc_new_path(*system_id*, *name*, *task_ids*, *attributes={}*)
Adds a path consisting of a list of tasks to the system.

Parameters

- **system_id** (*string*) – ID of the system
- **name** (*string*) – Name of the path
- **task_ids** (*list of strings*) – List of task ids corresponding to the tasks in the path.

Returns ID of the created path

Return type string

xmlrpc_new_resource(*system_id*, *name*, *attributes={}*)
Create a new resource with name and bind it to a system.

Parameters

- **system_id** (*string*) – ID of the system
- **name** (*string*) – Name of the resurce.

Returns ID of the created resource

Return type string

xmlrpc_new_system(*name*)
create new pycpa system and return it's id

Parameters **name** (*string*) – Name of the system.

Returns ID of the created system

Return type string

xmlrpc_new_task(*resource_id*, *name*, *attributes={}*)
Create a new task and bind it to a ressource.

Parameters

- **resource_id** (*string*) – ID of the resource
- **name** (*string*) – Name of the task.

Returns ID of the created task

Return type string

xmlrpc_pickle_system(*system_id*)
Pickle the pycpa system on the server-side

Parameters **system_id** (*string*) – ID of the system to analyze

Returns 0 on sucess

xmlrpc_protocol()

Returns protocol version

Return type integer

xmlrpc_set_attribute(*obj_id*, *attribute*, *value*)
Set the attribute of the object to value.

This method can be used to set any attribute of any previously created object., However, each scheduler or analysis expects certain attributes that must be set and ignores all others. See scheduler documentation for details (e.g. *pycpa.schedulers*).

Parameters

- **obj_id** (*string*) – ID of the task to set the parameter for.
- **attribute** (*string.*) – Attribute to set.
- **value** (*Depends on attribute.*) – Value to set the attribute to

Returns 0

xmlrpc_set_id_type(*id_type*)
Select the type for returned IDs.

- 'numeric' generates numeric IDs (strings of long int)
- 'id_numeric' like 'numeric', but prefixes `'id_'` (makes debug output executable)
- 'name' generates the ID from the objects' name
- 'full' is like 'name', but prefixes name by parent's name (TODO)

In case of 'name' or 'full', the ID is suffixed in case of duplicates.

Parameters **id_type** (*string*) – 'numeric', 'id_numeric', 'name', or 'full'

Returns 0

xmlrpc_tasks_by_name(*system_id*, *name*)

Returns a list of tasks of system_id matching name

Return type list of strings

4.4.2 SMFF Loader Module

Authors

- Philip Axer

Description

SMFF import/annotate

exception pycpa.smff_loader.**InvalidSMFFXMLException**(*value*, *dom_node*)

class pycpa.smff_loader.**SMFFLoader**
a simple SMFF xml loader reverse engineered sources, implements only a functional subset

CHAPTER 5

Bibliography

CHAPTER 6

What does pyCPA do?

Given, you have a (distributed) real-time system and you want to know about worst-case (end-to-end) timing behavior, then you can use pyCPA to obtain these bounds. You provide your architecture in the form of resources such as busses and CPUs and the corresponding scheduling policies. In a second step, you define your task-graph which is a specification of task-communication (precedence relations) and tasks' properties (best/worst-case execution times, priorities, activation patterns). pyCPA will then calculate the following metrics:

- worst-case response times (WCRT) for tasks
- end-to-end timing for chains of tasks
- maximum backlog of task activations (maximum buffer sizes)
- output event models for tasks

An introduction to the approach is provided in [Henia2005]. If you want to understand the internals of pyCPA we advice to read the paper [Diemer2012b].

6.1 Features:

- schedulers: (non-)preemtive fixed priority , Round Robin, TDMA, FIFO
- event model with periodic, jitter, minimum distance support
- system analysis: event model propagation
- end to end analysis (event- and time-triggered chains)
- gantt-charts (spnp, spp only)
- graphviz plots of your taskgraph
- SMFF support (through xml interface)

CHAPTER 7

Why pyCPA

Why not?

- pyCPA is a reference implementation and ideal for students who want to learn about real-time performance analysis research as well as researchers who want to extend existing algorithms.
- pyCPA is -as the name suggests- written in Python and extremely easy to use and extend. If you want, you can easily plugin new schedulers or your own analyses.
- pyCPA is -as the name also suggests- a framework for Compositional Performance Analysis that particularly addresses complex heterogeneous systems. You can easily use distinct analyses for different processing resources, which makes testing a new analysis in a more complex and realistic environment extremely easy.

However, pyCPA *should not* be used in any commercial-grade, safety-critical designs. It does not provide analysis methods for commercial scheduling protocols like OSEK.

CHAPTER 8

What pyCPA is not

pyCPA cannot and won't obtain the worst-case execution time of a task. Also, there is and will be no support for any specific protocols (e.g. OSEK, Ethernet, CAN, ARINC, AUTOSAR, etc.). Contact Luxoft if you need commercial support for any protocols or anything else that is beyond academic use-cases.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[Bate1998] Iain John Bate, “Scheduling and Timing Analysis for Safety Critical Real-Time Systems”, Phd Thesis, 1998

[Davis2007] Robert I. Davis and Alan Burns and Reinder J. Bril and Johan J. Lukkien, “Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised”, Real-Time Systems, Springer, 2007, 35, 239-272

[Diemer2010] Jonas Diemer and Rolf Ernst, “Back Suction: Service Guarantees for Latency-Sensitive On-Chip Networks”, The 4th ACM/IEEE International Symposium on Networks-on-Chip, 2010

[Diemer2012] Jonas Diemer and Daniel Thiele and Rolf Ernst, “Formal Worst-Case Timing Analysis of Ethernet Topologies with Strict-Priority and AVB Switching”, 7th IEEE International Symposium on Industrial Embedded Systems (SIES’12), 2012

[Diemer2012b] Jonas Diemer, Philip Axer and Rolf Ernst, “Compositional Performance Analysis in Python with pyCPA”, 3rd International Workshop on Analysis Tools and Methodologies for Embedded Real-TIme Systems (WATERS), 2012

[Gemlau2017] Kai-Björn Gemlau, Johannes Schlatow, Mischa Möstl und Rolf Ernst, “Compositional Analysis for the WATERS Industrial Challenge 2017”, International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), (Dubrovnik, Croatia), 2017

[Henia2005] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst, “System Level Performance Analysis - the SymTA/S Approach”, IEE Proceedings Computers and Digital Techniques, 2005

[Jersak2005] Marek Jersák, “Compositional Performance Analysis for Complex Embedded Applications”, Dissertation, Technische Universität Braunschweig, 2005

[Lehoczky1990] John P. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines”, Proceedings of the 11th Real-Time Systems Symposium, 1990

[Paliencia1998] Palencia and Harbour, “Schedulability analysis for tasks with static and dynamic offsets”, Proc. of RTSS, 1998

[Palencia2003] Palencia and Harbour, “Offset-based response time analysis of distributed systems scheduled under EDF”, Proc. of 15th Euromicro Conference on Real-Time Systems, 2003

[Richter2005] Kai Richter, “Compositional Scheduling Analysis Using Standard Event Models”, Dissertation, Technische Universität Braunschweig, 2005

[Rox2008] Jonas Rox and Rolf Ernst, “Modeling event stream hierarchies with hierarchical event models”, Proc. Design, Automation and Test in Europe (DATE 2008), 2008.

[Rox2010] Jonas Rox and Rolf Ernst, “Exploiting Inter-Event Stream Correlations Between Output Event Streams of non-Preemptively Scheduled Tasks”, Proc. Design, Automation and Test in Europe (DATE 2010), 2010.

[Schlatow2016] Johannes Schlatow and Rolf Ernst, “Response-Time Analysis for Task Chains in Communicating Threads”, Real-Time and Embedded Technology and Applications Symposium, 2016

[Schlatow2017] Johannes Schlatow and Rolf Ernst, “Response-Time Analysis for Task Chains with Complex Precedence and Blocking Relations” in International Conference on Embedded Software, 2017

[Schliecker2008] Simon Schliecker, Jonas Rox, Matthias Ivers, Rolf Ernst, “Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems, Proc. 6th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS), 2008

[Schliecker2009recursive] Simon Schliecker and Rolf Ernst, “A Recursive Approach to End-To-End Path Latency Computation in Heterogeneous Multiprocessor Systems”, Proc. 7th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS), 2009

[Schliecker2009] Simon Schliecker and Mircea Negrean and Rolf Ernst, “Response Time Analysis in Multicore ECUs with Shared Resources”, IEEE Transactions on Industrial Informatics, 2009

[Schliecker2011] Simon Schliecker, “Performance Analysis of Multiprocessor Real-Time Systems with Shared Resources”, Dissertation, Technische Universität Braunschweig, 2011

[Thiele2015] Daniel Thiele, Johannes Schlatow, Philip Axer und Rolf Ernst, “Formal timing analysis of CAN-to-Ethernet gateway strategies in automotive networks”, Real-Time Systems Journal, 2015

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

W

X