Compositional Performance Analysis in Python with pyCPA

Jonas Diemer, Philip Axer, Rolf Ernst
Institute of Computer and Network Engineering
Technische Universität Braunschweig
38106 Braunschweig, Germany
{diemer|axer|ernst}@ida.ing.tu-bs.de

***Abstract*—The timing behavior of current and future embedded and distributed systems becomes increasingly complex. At the same time, many application fields such as safety-critical systems require a verification of worst-case timing behavior. Deriving sound guarantees is a complex task, which can be solved by Compositional Performance Analysis. This approach formally computes worst-case timing scenarios on each component of the system and derives end-to-end system timing from these local analyses. In this paper, we present pyCPA, an open-source implementation of the Compositional Performance Analysis approach. Targeted towards academia, pyCPA offers features such as support for the most common real-time schedulers, path analysis for communicating tasks, import and export functionality, and different visualizations. Thus, pyCPA is a valuable contribution to the research domain.**

I. Introduction

Embedded applications such as complex, distributed control-loops and safety-critical sensor-actuator interactions are subject to hard real-time constraints where it must be guaranteed that certain functions will finish before their deadline. In most cases, it is not straightforward to show that all timing requirements are satisfied under all circumstances. Research in the field of real-time performance analysis and worst-case execution time analysis provided various formal approaches such as compositional performance analysis (CPA) [1] to solve this problem. CPA breaks down the analysis complexity of large systems into separate local component analyses and provides a way to integrate local performance analysis techniques into a system-level analysis.

This papaer presents pyCPA[1], an easy-to-understand and easy-to-extend Python implementation of CPA. To our knowledge, pyCPA is the only free (as in speech) implementation of the CPA approach. pyCPA targets academic use-cases such as lectures in real-time education, research of further extensions of CPA, or simple reference benchmarks for novel analysis methodologies. To ease interaction with other toolkits, pyCPA offers support for integration through file-based I/O with other related tools such as SMFF [2], SymTA/S [1] and MPA [3].

The philosophy of pyCPA is to include only a baseline implementation of research relevant algorithms (e.g. analysis of fixed-priority schedulers) without puzzling or distracting add-ons to keep the package as simple and handy as possible. Thus, contrary to commercial solutions such as SymTA/S, pyCPA does not include any industrial scheduling protocols such as CAN, Flexray, OSEK, and others. Nevertheless, pyCPA is built in a modular fashion and can easily be extended to support such protocols, too. pyCPA is not aimed for maximum performance, and it is not overly fine-tuned to keep the implementation simple and comprehensible. Only obvious performance tweaks are included.

The remainder of the paper is organized as follows: In Section II, we give an overview of real-time analysis approaches and corresponding analysis tools. Then, in Section III we elaborate on the system model as used in CPA and how it is implemented in pyCPA. After the formal foundation of CPA is introduced in Section IV, we sketch the workflow of pyCPA by analyzing an exemplary architecture in Section V. The integration of pyCPA with other toolkits such as SMFF is presented in Section VI . Finally, we conclude the paper in Section VII.

II. Related Work

There are different approaches for formal analysis of worst-case timing behavior on system level. Exact approaches like Uppaal [7] use model checking techniques to derive the worst-case timing of a system. This can be very expensive in terms of run-time and memory for larger (realistic) systems. Holistic approaches such as [8] have similar issues. Compositional approaches like Real-Time-Calculus [6] and Compositional Performance Analysis (CPA) [1] solve this by decomposing the analysis of the system at component level. They use abstract event models to describe the interaction of components in the worst- and best-case. Event models describe the maximum and minimum events arrivals for specific time intervals rather than exact instances in time. This can lead to pessimism in the analysis, but avoids the state space explosion from which holistic approaches suffer.

TABLE I
Tools for Worst-Case Timing Analysis

Tool	Approach	Commercial	Free	Open-Source
MAST	[4]	no	yes	yes
Uppaal	[5]	yes	yes	no
MPA Toolbox	[6]	no	yes	yes
SymTA/S	[1]	yes	no	no
pyCPA	[1]	no	yes	yes

[1] http://code.google.com/p/pycpa

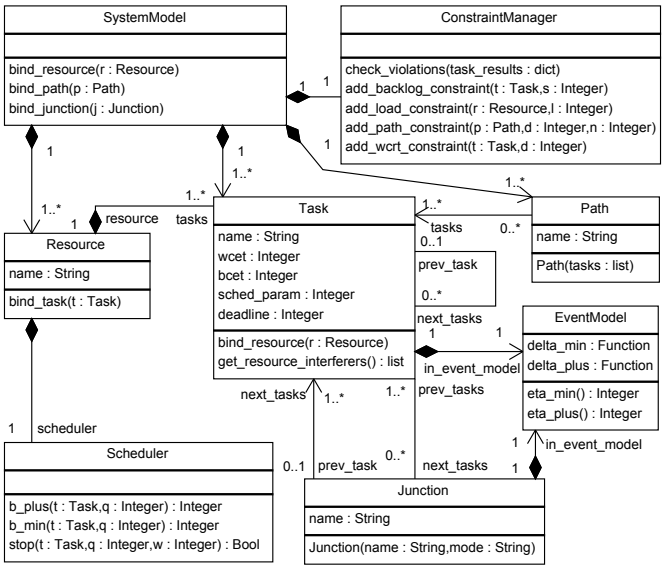

Fig. 1. System model of Compositional Performance Analysis

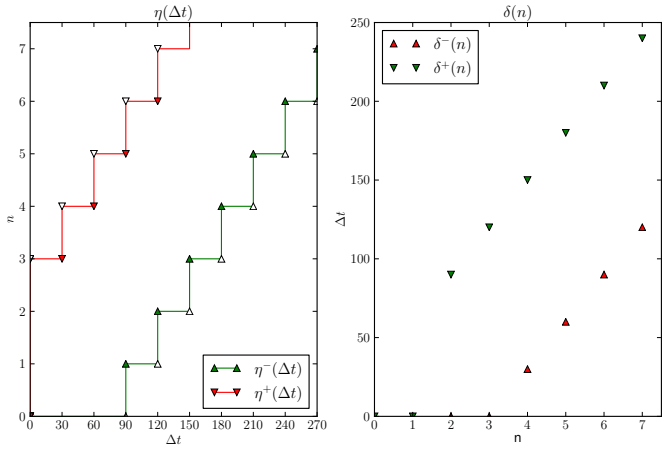

Fig. 2. Event model for a periodic activation with a period of $P = 30$ and a jitter of $J = 60$

Most of the proposed approaches have been implemented in software tools, which are summarized in Table I. With pyCPA, we present a toolkit which implements the CPA approach which is also used in the commercially available SymTA/S tool. pyCPA is publicly available in source-code, like the MPA Toolbox implementing Real-Time-Calculus.

III. The CPA System Model

In CPA, systems are modeled by sets of resources and tasks (see Figure 1). A resource provides processing time which is consumed by the tasks mapped to it. The mapping of tasks to resources is represented by references between tasks and a resource. Contention for resources with multiple tasks is resolved according to a scheduling policy (e.g. static priority preemptive), for which each task may include a scheduling parameter. The scheduling behavior is specified within a scheduler class which defines window functions (`b_min()` and `b_plus()`) used in scheduling analysis [9], see Section IV.

The execution behavior of a task τ_i is divided into the following steps: activation, core execution and finally completion/propagation. After being activated, a task (or job) is ready to execute and can be scheduled. It is assumed to require a core execution time in the interval between the best-case and worst-case execution times $[C_i^-, C_i^+]$. Between activation and completion, tasks may be interrupted by other tasks running on the same resource, which can be obtained via `get_resource_interferers()`.

A distributed application consisting of multiple communicating tasks is implicitly described by a directed graph (via the `next_tasks` and `prev_task` attributes) in which nodes are tasks and edges represent functional data dependencies. After a task's execution is completed, the task activates its dependent tasks (propagation). The application graph consists of task chains, which are called paths in pyCPA. Here, forks are possible, i.e. one task can activate multiple other tasks (forming multiple paths). The opposite, i.e. a join, is represented by a junction, and requires the definition of a semantic or "mode". There are two common join-semantics as discussed in [1]: For an *OR-join* any incoming event produces one outgoing event, and for an *AND-join*, an outgoing event is produced once events are available on all incoming edges.

Also part of the model are optional constraints, which can be used to define deadlines for tasks and path response times, limitations on the load of resources, or activation backlog (which usually translates to buffer requirements). In pyCPA all elements which model the system architecture (i.e tasks, resources, event models, junctions and paths) are distinct classes, as shown in Figure 1. For easy navigation, all classes are (redundantly) cross-referenced, e.g. resources keep a list of all mapped tasks and each task keeps a reference to its resource.

As discussed above, task activation and completion denote specific events, which are chained for dependent tasks (i.e. the completion event of one task is the activation event of its dependent tasks). Events can also originate from external sources, such as a timer. The arrivals of activation events of a task τ_i are modeled by minimum / maximum arrival curves $\eta_i^-(\Delta t)$ / $\eta_i^+(\Delta t)$, which return a lower / upper bound on the number of events that can arrive within any half-open time window $[t, t + \Delta t)$ [10]. These functions have pseudo-inverse counterparts, the so-called maximum / minimum distance functions $\delta_i^+(n)$ / $\delta_i^-(n)$, which return an upper / lower bound on the time interval between the first and the last event of any sequence of n event arrivals. Such event models cover all possible event arrivals of a specific event source as opposed to a specific trace of events.

For compact representation, standard event models in [10] use three parameters, event model period P, event model jitter J and a d^{min} which specifies the minimum distance between successive events in case the jitter is larger than the period.

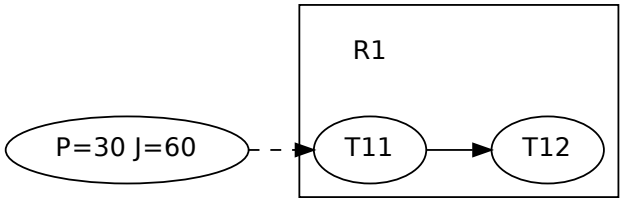

Fig. 3. A very simple pyCPA system model: two communicating tasks stimulated by one event model are mapped to one resource

The δ-functions for such an event model representation are as follows:

$$\begin{aligned} 0 \leq n < 2 \quad &: \quad \delta^+(n) = \delta^-(n) = 0 \\ n \geq 2 \quad &: \quad \delta^+(n) = (n-1)P + J \\ &\qquad \delta^-(n) = \max((n-1)d^{min}, (n-1)P - J) \end{aligned} \tag{1}$$

Figure 2 shows the arrival curves and minimum distance functions for a periodic task activation with a period of $P = 30$, and a jitter of $J = 60$. In pyCPA, event models are internally described by their δ-functions, which are represented as actual function references. There are generator functions for typical event models, such as periodic with jitter or periodic bursts. The η-functions, which are needed during some analysis steps, are derived directly from the δ-functions by using the following transformation:

$$\begin{aligned} n = 0 \quad &: \quad \eta^+(\Delta t) = 0 \\ n \geq 1 \quad &: \quad \eta^+(\Delta t) = \max_{n \geq 1, n \in \mathbb{N}} \{n \mid \delta^-(n) < \Delta t\} \\ &\qquad \eta^-(\Delta t) = \min_{n \geq 1, n \in \mathbb{N}} \{n \mid \delta^+(n+2) > \Delta t\} \end{aligned} \tag{2}$$

To speed up the event-model transformation, pyCPA leverages the fact that δ-functions are monotonous and implements a binary search. For further efficiency, δ-functions are cached, as they are referenced often with the same values (e.g. during event propagation, see next section). Note that although the CPA system model was conceived to analyze tasks executing on processor resources, it can also be used to for different systems such as Ethernet networks as presented in [11], [12] as well as CAN-buses as shown in [13]. Due to the CPA approach, pyCPA performs very fast, with the results being available within seconds. Even for a large system with over 1700 tasks on over 500 resources and an average load of over 90%, the analysis required only a couple of minutes.

A. Design Entry

Although pyCPA does not provide a GUI, there are rich ways to enter a system description. The easiest way is to instantiate the corresponding CPA objects directly in Python. As an example, consider the system model shown in Figure 3, which represents a small system with one resource R1 and two dependent tasks T11 and T12. Listing 1 demonstrates how this system can be represented in pyCPA. At first, a system object is instantiated which stores further objects. A resource

```
1  s = model.System()
2
3  r1 = s.bind_resource(model.Resource("R1",
4    schedulers.SPPScheduler()))
5
6  t11 = r1.bind_task(model.Task("T11", wcet=5, bcet=5,
7    scheduling_parameter=1))
8  t12 = r1.bind_task(model.Task("T12", wcet=9, bcet=1,
9    scheduling_parameter=2))
10
11 t11.link_dependent_task(t12)
12
13 t11.in_event_model = model.EventModel(P=30, J=60)
14
15 p1 = s.add_path("P1", [t11, t12])
16
17 s.constraints.add_backlog_constraint(t11, 5)
18 s.constraints.add_wcrt_constraint(t12, 90)
```

Listing 1. CPA system model directly instantiated in Python

named R1 is added to the system, for which a scheduling policy is defined by instantiating an SPP scheduler object. The scheduler object encapsulates scheduling specific functions, as discussed in Section IV. Both tasks are created and mapped to resource R1, worst- and best-case timing as well as the scheduling parameter (in this case a priority) are defined. Then, both tasks are linked according to the application graph and an input event model with a period of $P = 30$ and a jitter of $J = 60$ is specified for the first task. Since we are interested in the end-to-end latency from T11 to T12, we also define a path which includes the corresponding tasks. During runtime, pyCPA checks if the entered system description is well-formed e.g. there are no dangling tasks and no functional cycles without further external stimuli exist.

Some of the created tasks may exhibit constraints which either emerge from the underlying physical architecture (e.g. buffer size constraints) or non-functional timing constaints of the modeled application such as deadlines. In pyCPA, constraints are handled by a constraint manager (cf. Figure 1) which is attached to the system object. During analysis, the pyCPA kernel checks if any constraints are violated and eventually stops the analysis with an error message. As discussed, the constraint manager supports a set of constraints, but additional constraint semantics (e.g. reliability, mode-change latencies, slack) can be added by deriving from the pyCPA constraint manager class. In our simple example we constrain the available buffer size for the input queue of task T11 to 5 and add a deadline for task T12 of 90 time units.

Since one major focus of pyCPA is the interoperability with other timing toolkits, it offers a set of import and export libraries, which either convert a system description of another tool to the pyCPA model or vice versa. Specifically, pyCPA provides importers for SMFF [2] and SymTA/S 1.4 [1] as well as an exporter for MPA [3]. Depending on the complexity of the model transformation, additional import and export filters are quite simple to implement. Listing 2 illustrates the import of a SymTA/S 1.4 model.

```
loader = symload.SymtaLoader14()
s = loader.parse("symta14_test.xml")
```

Listing 2. Importing a SymTA/S 1.4 system model to pyCPA

IV. Compositional Performance Analysis

Once the system model is formulated, we are interested in its timing properties. A common metric is the worst-case response time, which is the largest time from activation of a task to its completion. Obviously, it is not straight forward to analyze timing for communicating tasks, since some event models are not know a priori (e.g. the input event model of $T12$ in Figure 3). Therefore, CPA uses an hierarchical iterative approach which is illustrated in Figure 4 to analyze the timing of such systems.

The general idea of the algorithm is as follows: At first, input event models for all tasks are initialized to one optimistic starting point. Naturally, the event model at the start of a path is an optimistic event model for all tasks on the path. Note, that later during the analysis, this optimism is resolved. Then, a local (component-level) analysis is performed for each task. After all local analyses have been performed, it is possible to derive output event models for all previously analyzed tasks. In a second step, newly derived output event models are propagated to all dependent tasks. If the output event model of a task has changed compared to the previous iteration step all tasks which are functionally or non-functionally (i.e. through resource sharing) influenced by this event model are re-analyzed.

This way, the two steps (local analysis and propagation) are alternated until either all event models remain stable or any constraint that might be specified (e.g. task deadline or path latency) is violated.

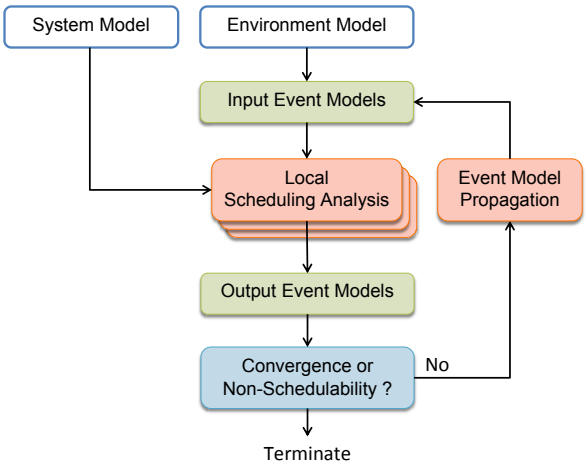

Fig. 4. The system analysis loop

```
class SPPScheduler(Scheduler):
    def b_plus(self, task, q):
        w = q * task.wcet
        while True:
            s = 0
            for ti in task.get_resource_interferers():
                if ti.scheduling_parameter <= \
                   task.scheduling_parameter:
                    s += ti.wcet * \
                    ti.in_event_model.eta_plus(w)
            w_new = q * task.wcet + s
            if w == w_new:
                return w
            w = w_new

    def stop(self, task, q, w):
        if task.in_event_model.delta_min(q + 1) >= w:
            return True
        return False
```

Listing 3. Simplified SPP scheduler class implementation

A. Local Analysis

The local analysis is based on a busy window approach as presented by Lehoczky in [14]. For this, we compute a so-called maximum q-event busy-time $B_i^+(q)$ which describes an upper bound of the amount of time a resource requires to service q activations of task τ_i, assuming that all q activations arrive "sufficiently early" (see [9]). A sufficient condition for the "sufficiently early" arrival of the q-th event is the arrival prior to the completion of its preceding event (the $(q-1)$-event busy-time). For this computation, a worst-case arrival of all interfering tasks is assumed. For a static-priority-preemptive (SPP) scheduler, the maximum busy-time can be computed as follows [14]:

$$B_i^+(q) = q \cdot C_i^+ + \sum_{j \in hp(i)} \eta_j^+(B_i^+(q)) \cdot C_j^+ \qquad (3)$$

where C_i^+ is the worst-case execution time of task τ_i, $hp(i)$ is the set of tasks with a higher-priority than task τ_i. Note that in this equation, $B_i^+(q)$ appears on both sides, resulting in an integer fixed-point problem. It can be solved by iteration, starting from $B_i^+(q) = q \cdot C_i^+$.

To find the worst-case response-time, only the first q_i^+ activations need to be considered, where q_i^+ is defined by a scheduler-dependent stopping condition. For SPP, the stopping condition is that all own and higher-priority load must be serviced. Hence:

$$q_i^+ = \min\{q \in \mathbb{N}^+ \mid \delta_i^-(q+1) \leq B_i^+(q)\} \qquad (4)$$

Note that in pyCPA, the satisfaction of the stopping condition is evaluated during the search for the worst-case response-time for every q without the explicit computation of q_i^+.

In pyCPA, this analysis is implemented in a modular way. As shown in the example from Listing 1, a scheduler-specific class object must be given for each resource. This class contains functions such as $B_i^+(q)$ and a stopping condition which evaluates whether the next activation has to be considered ($q+1 \leq q^+$) or whether the local analysis should terminate ($q+1 > q^+$). For an SPP scheduler a simplified

```
results = analysis.analyze_system(s)

for t in [t11, t12]:
  print("%s: wcrt=%d" % (t.name, results[t].wcrt))

bcl, wcl = path_analysis.end_to_end_latency(p1, 5)
print("Path latency: [%d,%d]"%(bcl,wcl)")
```

Listing 4. Analysis of a CPA system model

implementation is shown in Listing 3. Here, the busy-time function and the stopping condition are straightforward implementations of Equation 3 and Equation 4. pyCPA comes with scheduler implementations for the most common scheduling policies used in the embedded domain (e.g. static priority preemptive and non-preemptive, round-robin, TDMA, and earliest-deadline-first).

B. Global Analysis

The global analysis iteration is performed on task-level, i.e. the event-model propagation is done after the analysis of each task. For this, pyCPA maintains a set of dirty tasks to which all dependent tasks are added after the output event model of a task changes. In order to avoid unnecessary re-analysis of tasks, pyCPA analyzes tasks with the most dependent tasks first.

To compute the output event model of a task τ_i, we first need to determine its best- and worst-case response times R_i^- and R_i^+. These can be obtained from the busy windows which were gathered from the local analysis step. The worst-case response time can be found among the first q_i^+ busy-windows, whereas it is a safe assumption, that the best-case response-time equals the best-case execution time:

$$R_i^+ = \max_{q \in \mathbb{N}^+ \mid q \leq q_i^+} \left(B_i^+(q) - \delta_i(q) \right) \quad (5)$$

$$R_i^- = C_i^- \quad (6)$$

The worst-case scheduling jitter J_i^s for a task can be bounded to $J_i^s = R_i^+ - R_i^-$. From this, we can compute the output event model $\delta_{out,i}$ which adds the scheduling jitter to the input event model $\delta_{in,i}$ according to Equation 1.

$$\begin{aligned} \delta_{out,i}^-(n) &= \max\left(\delta_{in,i}^-(n) - J_i^s \ , \ (n-1)C_i^-\right) \\ \delta_{out,i}^+(n) &= \delta_{in,i}^+(n) + J_i^s \end{aligned} \quad (7)$$

This way of obtaining an output model is called *jitter propagation* in pyCPA. In [15], Schliecker et al. provide a more sophisticated event-model propagation which constructs the output event-model by considering the cases of all q^+ busy windows. This *busy-window propagation* yields tighter results, and therefore is the default in pyCPA.

V. Running an Analysis in pyCPA

Once a pyCPA system model is available, several different real-time metrics can be derived easily. Listing 4 shows the necessary steps to analyze the system model which was given in Listing 1 in Section III. The actual CPA iteration (local analysis and propagation) as depicted in Figure 4 is performed in `analyze_system()`. The analysis results are returned inside a result object and are available directly after the execution of `analyze_system()`. This includes the activation backlog, which is the largest amount of unprocessed task activation events, as well the best- and worst-case response-times. To derive more sophisticated properties of the system, such as the best- and worst-case end-to-end latency of a path, is is necessary to call dedicated analysis methods after the system has been analyzed. In Listing 4, we additionally derive the best- and worst-case path latency for 5 consecutive events for path P1.

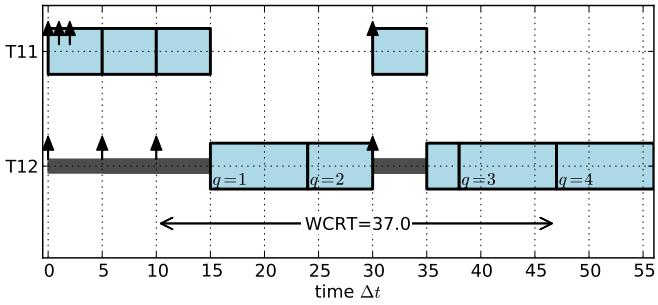

Fig. 5. Gantt-chart of the worst-case scheduling scenario for task T11

A. Visualization

Once analysis data is available, the results can be post-processed and visualized using one of the many existing Python packages such as matplotlib. Based on this, pyCPA provides several functions for visualization. The complete system model can be displayed using PyGraphviz. The system graph shown in Figure 3 was generated this way. The system plot is especially useful to visually inspect the entered system model for larger systems. Also, event models can be plotted using pyCPA (using matplotlib internally) as the one shown in Figure 2.

For illustrative purposes, it might be of further interest to generate one execution trace (Gantt-chart) which leads to the worst-case response time. For instance, the chart in Figure 5 shows the worst-case response time for task T12 from the initial example. pyCPA comes with a discrete event simulator which is built on top of SimPy. To generate the Gantt-chart, pyCPA simulates the critical instant behavior according to the specified scheduling policy and stores all preemption- and run-times of a task. The trace data can then be used to plot the Gantt-chart as shown in Figure 5 which also highlights the activation at which the worst-case response time is observed.

VI. Interoperability

pyCPA uses a generic system model that is compatible with that of many other tools. For instance, pyCPA provides an XML importer for SymTA/S system models. Furthermore, it can directly read and write system models generated by the system-models-for-free (SMFF) generator [2]. pyCPA uses

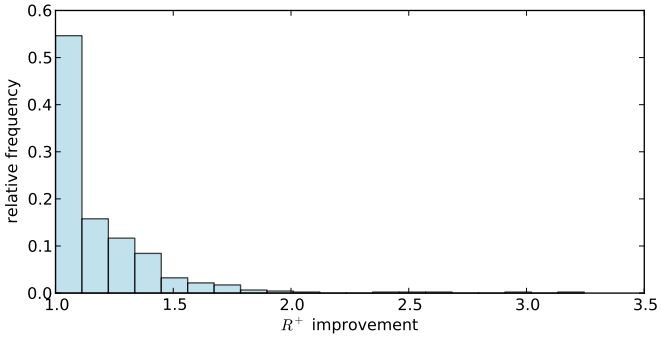

Fig. 6. Relative improvement of the busy-window propagation vs. jitter propagation

Python's `minidom` to parse the XML-based SMFF file-format and converts the SMFF model to the internal pyCPA representation. After analysis, the SMFF model can be written including annotated analysis results such as the worst-case response time as well as output event models.

To show the convenience and actual applicability of such an interface, we conduct an experiment in which the busy-window propagation from [15] is compared with the previously presented jitter propagation technique. Obviously, the results highly depend on the actual system model. Therefore, we use SMFF to generate a large set of representative random systems consisting of up to 4 resources and 3 paths with 4 tasks which lead to a resource load of up to 0.8. Actual mapping and scheduling parameters are randomized according to a heuristic implemented in SMFF. We have generated 500 systems with SMFF which were analyzed with pyCPA to derive the relative improvement of the busy-window propagation over jitter propagation. Systems, which were not feasible (i.e. WCRT larger than ten periods), were discarded. Figure 6 shows the distribution of this improvement. As expected, busy-window propgation yields a up to 3 times better result compared to jitter propagation. The analysis runtime for jitter and busy-window where approximately the same with about 10 ms per analyzed system. All experiments can be directly carried out in Python which has the immediate advantage that results can be post-processed and directly plotted. Thus experiments are self-contained and can be easily reproduced later.

It is also possible to export the pyCPA system model to other file-formats. This is useful to compare analysis results with other frameworks. For this purpose, pyCPA includes a simple exporter which directly outputs a Matlab description of the system to be used with the Modular Performance Analysis (MPA) framework. In our experiments, the analysis results for static-priority systems were identical with pyCPA and MPA, which matches our expectations.

VII. Conclusion

In this paper, we have presented pyCPA, a Python-based framework for Compositional Performance Analysis. It can be used to derive worst-case timing of complex embedded and distributed real-time systems. We have presented the basic architecture of pyCPA which is very easy-to-use as we have demonstrated in this paper by a small example. Furthermore, pyCPA is open-source and has a modular architecture, so it can be extended easily to cover different scheduling policies or implement advanced analysis algorithms. pyCPA also provides interfaces to existing tool suites such as the system-model generator SMFF or the Modular Performance Analysis framework. For these reasons, pyCPA is a valuable contribution to the research community in the field of timing analysis of embedded real-time systems.

Acknowledgment

This work has been funded by the "Bundesministerium für Bildung und Forschung" (BMBF), the "Deutsche Forschungsgemeinschaft" (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - spp1500.itec.kit.edu), the Advanced Research & Technology for Embedded Intelligence and Systems (ARTEMIS) within the project 'RECOMP', support code 01IS10001A, agreement no. 100202 as well as Intel Corporation.

References

[1] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis–the SymTA/S Approach," *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 2, 2005.

[2] M. Neukirchner, S. Stein, and R. Ernst, "SMFF: System Models for Free," in *2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Porto, Portugal, July 2011.

[3] E. Wandeler, "Modular performance analysis and interface-based design for embedded real-time systems," Ph.D. dissertation, Swiss Federal Institute of Technology Zurich, 2006.

[4] M. Gonzalez Harbour, J. Gutierrez Garcia, J. Palencia Gutierrez, and J. Drake Moyano, "MAST: Modeling and analysis suite for real time applications," in *Real-Time Systems, 13th Euromicro Conference on, 2001.*, 2001, pp. 125 –134.

[5] R. Alur and D. Dill, "Automata for modeling real-time systems," *Automata, languages and programming*, pp. 322–335, 1990.

[6] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *ISCAS*, vol. 4, 2000.

[7] K. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[8] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and microprogramming*, vol. 40, no. 2-3, 1994.

[9] S. Schliecker, "Performance analysis of multiprocessor real-time systems with shared resources," Dissertation, Technische Universität Braunschweig, 2011, submitted 2010.

[10] K. Richter, "Compositional scheduling analysis using standard event models," Ph.D. dissertation, TU Braunschweig, 2005.

[11] J. Rox and R. Ernst, "Formal Timing Analysis of Full Duplex Switched Based Ethernet Network Architectures," in *SAE World Congress*, vol. System Level Architecture Design Tools and Methods (AE318). SAE International, Apr 2010.

[12] J. Diemer, J. Rox, and R. Ernst, "Modeling of Ethernet AVB Networks for Worst-Case Timing Analysis," in *MATHMOD*, Austria, 2012.

[13] P. Axer, M. Sebastian, and R. Ernst, "Probabilistic response time bound for can messages with arbitrary deadlines," in *Proc. of Design, Automation and Test in Europe*, Dresden, Germany, 2012.

[14] J. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proceedings of the 11th Real-Time Systems Symposium*, 1990.

[15] S. Schliecker, J. Rox, M. Ivers, and R. Ernst, "Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems," in *CODES-ISSS*, oct 2008.