pyCPA – A pragmatic implementation of Compositional Performance Analysis

Jonas Diemer, Philip Axer, Daniel Thiele, Johannes Schlatow et al.
Introduction

What is pyCPA?

- python implementation Compositional Performance Analysis
- targets heterogeneous distributed systems
- calculates bounds on
  - worst-case/best-case response times (WCRT/BCRT)
  - end-to-end latency,
  - load,
  - buffer sizes
- framework for building tailored research tools
Outline

- Brief history of (py)CPA
- Foundations – informal introduction of CPA
  - System model
  - Iterative analysis flow
- pyCPA overview
  - Core
  - Analysis extensions
- Hand-on session overview
Brief history of (py)CPA

- **2005** – the SymTA/S approach [1]
  - commercialised by Symtavision (since 2016: Luxoft)
- **2010-2012** – pragmatic implementation in python by J. Diemer and P. Axer
  - free, open-source, extensible, for academic use
- **2012** – pyCPA published at WATERS [2]
- **since then**
  - use of pyCPA for prototyping new analyses (extensions), e.g.
    - Ethernet (AVB), CAN
    - NoCs, gateways
    - runnables, task chains
    - ...
  - maintenance and minor improvements of pyCPA core

Informal introduction to CPA

System model
- Tasks and resources
- Event model interfaces
  - eta/delta curves
  - standard event models

CPA flow
- local scheduling analysis
  - busy-window approach
- event model propagation
  - jitter propagation
  - busy-times
CPA system model

- **Resources** → *provide service (CPU time)*
  - scheduled according to policy (e.g. round-robin)

- **Tasks** → *consume service*
  - worst-case and best-case execution times (WCET/BCET)
  - **phases:**
    - activation
    - **execution**
    - completion

- **Task links** → *activation dependencies*
  - flow of activation events (one task activates the other)
  - abstracted by **event models**
Event model abstraction

Idea: An event model is a worst-case abstraction from the actual trace.

\( \eta^{+/-}(\Delta t) \): minimum/maximum number of activations within any time window \( \Delta t \)

\( \delta^{+/-}(n) \): maximum/minimum time interval between first and last activation of any sequence of \( n \) activations (pseudo-inverse to \( \eta^{+/-}(\Delta t) \))
**From traces to event models**

\( \eta^{+/\cdot}(\Delta t) \): Minimum/Maximum number of activations within any time window \( \Delta t \)

\( \delta^{+/\cdot}(n) \): Maximum/minimum time interval between first and last activation of any sequence of \( n \) activations (pseudo-inverse to \( \eta^{+/\cdot}(\Delta t) \))

Example: \( \eta^{+/\cdot}(5) \):

\( \Delta t = 5 \)

We see 1 events

**Idea:** Shift a window of length \( \Delta t = 5 \) over the trace and count events
From traces to event models

$\eta^{+/-}(\Delta t)$: Minimum/Maximum number of activations within any time window $\Delta t$

$\delta^{+/-}(n)$: Maximum/minimum time interval between first and last activation of any sequence of $n$ activations (pseudo-inverse to $\eta^{+/-}(\Delta t)$)

Example: $\eta^{+/-}(5)$:

Idea: Shift a window of length $\Delta t = 5$ over the trace and count events
From traces to event models

\( \eta^{+/-(\Delta t)} \): Minimum/Maximum number of activations within any time window \( \Delta t \)

\( \delta^{+/-(n)} \): Maximum/minimum time interval between first and last activation of any sequence of \( n \) activations (pseudo-inverse to \( \eta^{+/-(\Delta t)} \))

Example: \( \eta^{+/-(5)} \):

Idea: Shift a window of length \( \Delta t = 5 \) over the trace and count events
From traces to event models

\( \eta^{+/-(\Delta t)} \): Minimum/Maximum number of activations within any time window \( \Delta t \)

\( \delta^{+/-(n)} \): Maximum/minimum time interval between first and last activation of any sequence of \( n \) activations (pseudo-inverse to \( \eta^{+/-(\Delta t)} \))

Example: \( \eta^{+(5)} \):

\[
\eta^+(\Delta t=5): 4 \\
\eta^-(\Delta t=5): 1
\]
From traces to event models

\( \eta^{+/-}(\Delta t) \): Minimum/Maximum number of activations within any time window \( \Delta t \)

\( \delta^{+/-}(n) \): Maximum/minimum time interval between first and last activation of any sequence of \( n \) activations (pseudo-inverse to \( \eta^{+/-}(\Delta t) \))

Example: \( \eta^{+/-}(5) \):

Similar approach to retrieve \( \delta^{+/-}(n) \):
Measure distance between any sequence of \( n \) events

In fact: \( \delta \) and \( \eta \) can be converted into each other!
All traces which stay between $\eta^+(\Delta t)/\delta^-(n)$ and $\eta^-(\Delta t)/\delta^+(n)$ satisfy the event model.
What if we don’t have a trace

**Standard event models:**

- periodic (P)
  \[ \delta^-(n) = (n-1) \times P \]
- periodic with jitter (PJ)
  \[ \delta^-(n) = \min(0, (n-1) \times P - J) \]
- periodic with jitter min. distance (PJd)
  \[ \delta^-(n) = \min((n-1) \times d, (n-1) \times P - J) \]
- sporadic (using min. interarrival time)
- define your own:
  *e.g. burst of c events every T time.*
CPA overview

Event model abstraction renders analysis compositional

- **output event model** can be computed from:
  - a) known **input event model** and
  - b) result of **resource analysis**

→ iterative analysis flow
CPA – iterative analysis flow

Step 1: Local analysis
- Compute each task’s worst-case behavior based on critical instant scenario.
- Derive task output event models.

Step 2: Global analysis
- Propagate event models to dependent tasks.
- Go to step 1 if any event model has changed.
- Otherwise, terminate.

(Step 3: Path analysis)
- Compute end-to-end latencies.
- E.g. sum of WCRTs.
Local resource analysis

Scheduling policy: SPP
(Static Priority Preemptive)

$T_1 > T_2$
Busy window (SPP)

**Level-i busy window**: time interval during which the resource is busy executing $T_i$ or any task with priority higher than $T_i$

- Largest (worst-case) busy window is computed from critical instant assumption!

---

![Diagram showing level-1 and level-2 busy windows with preemption and resumption points.](image-url)
Busy-window analysis based on critical instant

\[ \delta_{-1}(4) \]
\[ \delta_{-1}(3) \]
\[ \delta_{-1}(2) \]

\[ \delta_{-2}(2) \]

largest level-2 busy window \( \geq \) worst-case response time
Response times

\[ R^+ = \max_q (B(q) - \delta^{-}(q)) \] worst-case response time found among busy times \( B(q) \)
Local resource analysis

Summary

- busy-window analysis depends on scheduler (e.g. round robin, SPP, SPNP, etc.)
- busy times are used to calculate worst-case/best-case response times $R^+/-$
- output event models can be computed from busy-window analysis
  - jitter propagation:
    - $\delta_{out}(n) = \delta_{in}(n) - J_{out}$
    - with output jitter $J_{out} = R^+ - R^-$
  - better: Derive event model from output trace that results from busy times and input event model.
- see [3]

pyCPA core

What comes with pyCPA:

- CPA system model
- event models (incl. transformation between eta/delta functions)
- local resource analyses (SPP, SPNP, round robin, TDMA, etc.)
- calculation/propagation of output event models
- iterative analysis kernel
- path analyses
- visualisation
  - plotting of event models
  - system graphs
  - Gantt charts (SPP/SPNP only)

→ CPA framework for researchers, not a tool for end users
pyCPA model

CPA components

pyCPA Classes (simplified)

- System
- Resource
- Scheduler
- Task
- EventModel
- Other classes
Setting up a system

- Easiest way to model systems: **Python code**
  - Instantiate **system model**
  - Instantiate and bind
    - Resources
    - Tasks
  - Link dependent tasks
  - Instantiate **event models** for tasks with no predecessor
- **Optional:**
  - Add **paths** for latency analysis
  - Add **constraints**

```python
s = model.System()

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

t11 = r1.bind_task(model.Task("T11", wcet=5, bcet=5,
                                scheduling_parameter=1))
t12 = r1.bind_task(model.Task("T12", wcet=9, bcet=1,
                                scheduling_parameter=2))

t11.link_dependent_task(t12)

t11.in_event_model = model.EventModel(P=30, J=60)

p1 = s.add_path("P1", [t11, t12])

s.constraints.add_backlog_constraint(t11, 5)
s.constraints.add_wcrt_constraint(t12, 90)
```

System graph generated by pyCPA from system model
Analysis of a system

- Simply call `analysis.analyze_system()`
  - Results are returned in a dictionary, indexed by task names
  - More detailed analyses (e.g. path latency) are called separately

```python
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)")
```

- Results can be visualized
  - Gantt charts of critical instant scenario
  - Plots of results via matplotlib
Additional concepts and analyses

Junctions and forks
- arbitrary strategies for joining/forking event models

Limited event-model propagation
- event-model propagation only at resource boundaries

Path analyses
- event-triggered paths (event chains)
  - baseline: sum of WCRTs
  - improvement for pipelined chains (pay burst only once)
- time-triggered paths (cause-effect chains)
  - sum of WCRTs + sampling delay

→ Stable framework for (almost) arbitrary analysis extensions.
pyCPA analysis extensions

- pyCPA core
  - pyCPA extension A
    - Local Resource Analysis
    - Extended System Model
    - Improved Path Analysis
  - pyCPA extension B
    - Extended Event Models
    - Improved Propagation
    - Environment Model

- System Model
  - Environment Model
    - Input Event Models
    - Local Resource Analysis
    - Output Event Models
    - Convergence or Non-Schedulability?
      - No
      - Event Model Propagation
    - Path Analysis

5th December 2017 | J. Schlatow | An introduction to pyCPA | Slide 27
Extension overview

Quite a few model and analysis extensions have been developed over the past years, e.g.:

- CAN
- Ethernet AVB
- NoCs
- Gateways (multiplexing/demultiplexing)
- Task-chain busy-window (propagation at resource boundaries)
- Runnables
- etc.

technology-specific model layer, e.g.:
- frames = communication tasks
- output ports = communication resources
- traffic streams = paths

resource-analysis replacements
new propagation methods,
system-model refinements, etc.

Not all of them are publicly available (please ask the authors for academic use).
Example extension: task-chain analysis

**Standard CPA:**

- Input event model
- Output event model

- Resource \( \tau_a \)
- Resource \( \tau_b \)

**Drawbacks:**

- Interference accounted multiple times (in each busy window)
  - Pessimistic WCRTs
  - (Too) many iterations

- Event models get increasingly ‘bursty’ along the path
Example extension: task-chain analysis

Task-chain busy-window analysis:

Benefits:
- considers transactional chains, blocking relations
- significantly better WCRTs and end-to-end latencies (and faster analysis)

Usage:

- sequence charts manual
- thread communication (GraphML) automated (OS-specific)
- enriched task graph (GraphML) ANALYZE
What to expect in the hands-on session?

The basics:
- modelling and analyzing a system
- plotting
- path analysis

And more:
- use of junctions
- custom fork strategy
- custom scheduler
- custom propagation
Thank you for your attention.

Source code:
https://bitbucket.org/pycba/

Docs: