Response-Time Analysis for Task Chains in Communicating Threads with pyCPA

Johannes Schlatow, Jonas Peeck and Rolf Ernst
Institute of Computer and Network Engineering, TU Braunschweig
{schlatow,jonasp,ernst}@ida.ing.tu-bs.de

***Abstract*—When modelling software components for timing analysis, we typically encounter functional chains of tasks that lead to precedence relations. As these task chains represent a functionally-dependent sequence of operations, in real-time systems, there is usually a requirement for their end-to-end latency. When mapped to software components, functional chains often result in communicating threads. Since threads are scheduled rather than tasks, specific task chain properties arise that can be exploited for response-time analysis by extending the busy-window analysis for such task chains in static-priority preemptive systems. We implemented this analysis by means of an analysis extension for pyCPA, a research-grade implementation of compositional performance analysis (CPA).**

The major scope of this demo is to show how CPA can be reasonably performed for realistic component-based systems. It also demonstrates how research on and with CPA is conducted using the pyCPA analysis framework. In the course of this demo, we show two approaches for the extraction of an appropriate timing model: 1) the derivation from a contract-based specification of the software components and 2) a tracing-based approach suitable for black-box components. We also demonstrate how this timing model is fed into the analysis extension in order to obtain response-time results for the task chains of interest. Finally, we present how the developed analysis extension speeds up the CPA and therefore enables an automated design-space exploration and optimisation of the threads' priority assignments in order to satisfy the pre-defined latency requirements.

I. Introduction

Larger embedded systems are often implemented as a collection of functions, each described as a task graph. To derive end-to-end response times in such task graphs, we are interested in the response times between the respective tasks. In this paper, we are interested in task chains which are derived from communicating software threads leading to specific task chain properties that can be exploited for response-time analysis covering both synchronous and asynchronous communication. Such communicating threads have become the standard implementation vehicle, e.g. in modern automotive software components [1] or in microkernel-based systems [2], [3].

When we try to perform a timing analysis for these systems, we first encounter a mismatch between the programming model and the timing analysis model: On the one hand, the programmer implements a thread that communicates at arbitrary points in its execution using the available primitives such as **synchronous IPC** or **asynchronous notifications** which are prominent in microkernel-based systems. On the other hand, the timing architect models the system by tasks that are only allowed to communicate at the end of their execution, implicitly assuming asynchronous communication semantics.

Figure 1 shows a thread (Thread 1) that (synchronously) calls another thread (Thread 2) at some point in its execution. Thread 1 can only continue after the latter completed and returned. The right side of the figure illustrates how this scenario is reflected in the timing model by a chain of tasks that represent the segments of the threads ($1a$, 2 and $1b$) along with their precedence relations.

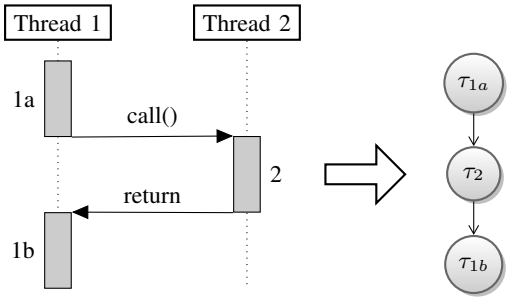

Figure 1. Communicating threads (implementation) naturally split up into a chain of tasks (timing model).

Although this is a straightforward transformation, it already obfuscates important information that should be respected by the timing analysis: The task timing model does not reflect the blocking behaviour of the synchronous call, i.e. it does not reflect that τ_{1a} cannot execute again before τ_2 returned. It neither represents the execution dependencies between the thread segments, i.e. that τ_{1a} cannot execute before τ_{1b} finished.

In [4], we presented an extension of the busy-window response-time analysis which exploits the particular semantics in task chains resulting from communicating threads in static-priority preemptive (SPP) systems. This approach is able to cope with varying priorities along the chain and even reduces the computational effort and overestimation in comparison to conventional CPA.

We implemented this by means of an extension of the pyCPA analysis framework [5], [6], which is specifically tailored for easy and modular extensibility. This implementation augments the task graph model provided by the pyCPA analysis kernel with additional information about the task chains and their activation semantics: In this extended model,

we differentiate between the activation semantics, i.e. we distinguish synchronous (i.e. blocking) from asynchronous (i.e. non-blocking) edges in the task graph. As our response-time analysis approach [4] considers entire task chains as opposed to single tasks, we introduce a preprocessing step in which the superordinate task chains are defined. Note that this preprocessing can either be done manually by the designer or timing architect, or automatically by tool support.

II. Demo

In the scope of this demo, we will present the design flow that can be applied for the timing verification during the integration of component-based systems. We assume that the implementation of the software components is already completed and that some of the software components may be provided by a third party, i.e. their source code is either not available or not fully understood. Furthermore, we already obtained a valid composition of the components either by a manual designer-driven process or an autonomous approach [7]. The remaining task is thus to extract a timing model for the system in order to evaluate and explore possible scheduling parameters. As we use SPP scheduling, the only scheduling parameter is the priority of a thread. Our demo splits up into a model-extraction and analysis part that we illustrate on an exemplary but realistic application. The software components and run-time environment used for our demonstrator are based on the Genode OS Framework [8].

A. Model extraction

We demonstrate two methods for extracting the timing model for a particular application in a system. On the one hand, we apply a contract-based approach that requires a formal description of a component, its threads and their interaction with other components via the specified interfaces. On the other hand, we show a tracing-based approach in which the components' behaviour is extracted by acquiring tracing information of one or multiple test runs.

1) Contract-based: The contract-based model extraction is based on an abstract formal specification of the components implementation. In order to extract a timing model, a component's contract must specify an abstract task graph, which dissects the threads into sequential parts (i.e. tasks) and communication directives (i.e. synchronous or asynchronous communication to another thread or another component). Each task is annotated with an upper and lower bound on its execution time. As the communication partner is not necessarily known during component development, the communication directives must be (automatically) resolved after a valid composition is chosen in order to link the abstract task graphs of all components. In this part, we demonstrate how our tools automatically derive this composite timing model from the contracts of our example application.

2) Tracing-based: For the tracing-based model extraction, we execute the software components (i.e. their composition) on the target platform and record their execution times and communication pattern. For this purpose, we leverage the existing tracing infrastructure of the Genode OS Framework in order to record the corresponding timestamps. We further developed a set of tools that process these traces as well as extract and visualise Gantt charts, similar to the sequence diagram in Figure 1, that assist the manual derivation of a timing model. Note that these tools can also be used for a validation of an existing timing model. It should furthermore be mentioned, that this approach is not solely suitable for the timing verification of critical application as, in general, the traces do not capture the entire range of timing behaviour.

B. Analysis, evaluation and exploration

Once we acquired a timing model of our application, we can run the response-time analysis for all paths of interest using pyCPA and the analysis extension presented in [4]. By this, we can explore the performance of different scheduling parameters and investigate their feasibility w.r.t. the applications' latency constraints specified by contracts. We demonstrate this by conducting an automated design-space exploration and by visually inspecting the analysis results in relation to the solution space defined by the contracts. This eventually allows as to optimise the scheduling parameters and validate the expected results by executing our example application on our demonstrator in order to record, extract and visualise the corresponding traces.

Acknowledgements

This work was supported by the DFG Research Unit Controlling Concurrent Change (CCC), funding number FOR 1800. We thank the members of CCC for their support.

References

[1] AUTOSAR website. [Online]. Available: http://www.autosar.org/

[2] PikeOS Hypervisor. [Online]. Available: https://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/

[3] seL4 Microkernel. [Online]. Available: https://sel4.systems/

[4] J. Schlatow and R. Ernst, "Response-time analysis for task chains in communicating threads," in *22nd Real-Time Embedded Technology and Applications Symposium (RTAS 2016)*, Vienna, Austria, April 2016.

[5] pyCPA website. [Online]. Available: https://bitbucket.org/pycpa/pycpa

[6] J. Diemer, P. Axer, and R. Ernst, "Compositional Performance Analysis in Python with pyCPA," in *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Jul. 2012.

[7] J. Schlatow, M. Moestl, and R. Ernst, "An extensible autonomous reconfiguration framework for complex component-based embedded systems," in *12th International Conference on Autonomic Computing (ICAC 2015)*, Grenoble, France, July 2015, pp. 239–242.

[8] Genode OS Framework. [Online]. Available: http://genode.org/