Hypervisor-Based Target Deployment Strategies for Time Predictability in Model-Based Development

Florian Schade, Tobias Dörr, Jürgen Becker
Karlsruhe Institute of Technology, Germany
{florian.schade, tobias.doerr, juergen.becker}@kit.edu

***Abstract*—The model-based toolchain for software systems engineering developed by the XANDAR project supports designers in the integration of software with different degrees of granularity. In its fully automated backend, it deploys this software to modern target platforms such as heterogeneous multicore processors. In this paper, we discuss target deployment strategies that guarantee a time-deterministic execution and meet relevant isolation requirements.**

***Index Terms*—X-by-Construction, model-based design, system software, embedded software, hypervisors.**

I. Introduction

The next generation of networked embedded systems is associated with major design challenges. In addition to an increasing need for processing power, support for machine learning components, and the consolidation of functionality on multiprocessor system-on-chip (MPSoC) devices, non-functional requirements such as for safety, security, or real-time behavior have to be fulfilled.

To tackle this challenge, the XANDAR project aims to provide a holistic toolchain for model-based software systems engineering that supports designers in meeting these requirements using the X-by-Construction (XbC) paradigm [1]. It automatically transforms user-provided inputs into implementation artifacts that meet functional and non-functional requirements while providing system verification artifacts.

In this work, we present strategies for the deployment of software to hypervisor-managed target platforms and discuss their compatibility with the XANDAR process.

II. Background

The XANDAR process consists of the two major phases visualized in Fig. 1. In the *model-based frontend*, the user specifies the system architecture and is then supported in integrating the functional code. This allows for a target-agnostic system simulation that accurately represents the functional and timing behavior at the system boundary, facilitating verification and validation of the system behavior. In the *XbC backend*, automated model and code transformations are applied to achieve non-functional properties, for example to implement safety mechanisms [2] or to increase performance. Finally, target-specific implementation artifacts are created for deployment on the target runtime environment (RTE).

In the architecture modeling step, software is described as a network of *software components* (SWCs) interconnected via *channels*. SWCs encapsulate user-provided application code that follows a time-triggered programming model. This model is based on the Logical Execution Time (LET) concept [3] and allows for a consistent execution of SWCs in both the simulation framework and on the target RTE. It abstracts from the actual runtime of a task by defining a logical task execution time. Tasks read inputs at the beginning of their LET interval and write output data at its end, both in conceptually zero time. In the system model, each SWC is assigned a periodic *LET window*, defined by the LET, a period, and an offset relative to the global schedule. This approach circumvents the issue of varying task runtime per iteration on the target platform and simplifies simulation.

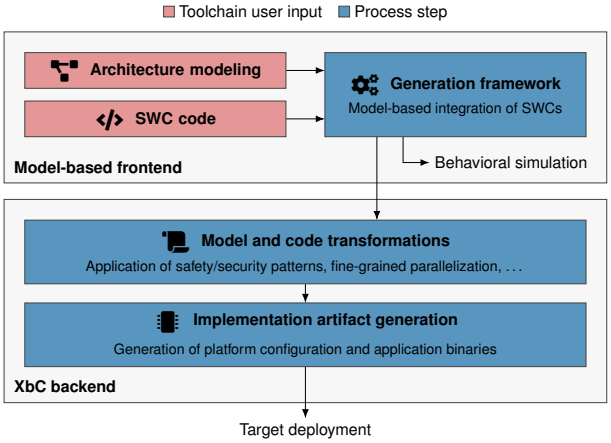

Fig. 1. High-level overview of the XANDAR development process

III. Target Deployment Strategies

To meet the requirements of modern software systems, the XANDAR toolchain supports SWCs of different degrees of granularity. While basic SWCs contain monolithic code implementing a single functionality, complex SWCs can comprise general-purpose operating systems (OS), including applications and supporting software, such as device drivers. Therefore, the RTE is required to support both OS-based and bare-metal SWCs. In addition, support for mixed-critical applications is essential, which makes it necessary to provide isolation between SWCs. The runtime is further required to implement the LET-based execution of SWCs as well as deterministic timing when accessing peripherals at the system boundary, such as sensors and actuators.

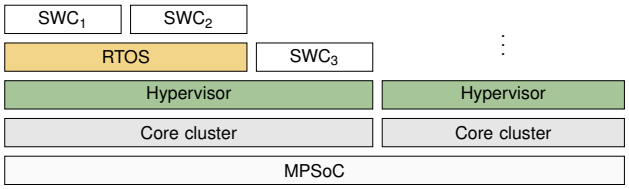

Fig. 2. Hybrid mapping of SWCs to an MPSoC

In the following, three strategies for mapping SWCs to the target platform will be discussed: mapping to hypervisor partitions (*partition mapping*), mapping to RTOS tasks (*task mapping*), and *hybrid mapping* as combination of both.

Embedded hypervisors implement a virtualization layer that allows for the integration of both bare-metal applications and full OS stacks. Strong isolation is achieved by microkernel approaches, which minimize the trusted computing base to facilitate analyzability and certifiability. Therefore, this approach allows for both basic and complex SWC deployment as well as the integration of mixed-critical SWCs on a single processor. For short-running SWCs, however, this strategy can lead to undesired overhead, since the reconfiguration of memory management units and cache flushes cause a performance penalty during context switches. Other factors such as the maximum number of partitions or insufficient capabilities of the programming interface exposed to guest applications might further limit the applicability of this approach.

Real-time operating systems (RTOS) allow developers to integrate applications in the form of tasks. They provide mechanisms for multi-threading, such as real-time scheduling, synchronization, and communication primitives, among others, which simplify the deployment of basic SWCs as tasks. Since most RTOS do not implement virtual address spaces and memory protection [4], low context switching overhead can be achieved. Compared to partition mapping, task mapping can therefore be expected to achieve significantly higher efficiency when many short-running SWCs are deployed. This, however, comes at the cost of missing isolation, rendering task mapping unsuitable for mixed-criticality SWCs. In addition, missing support for virtual memory management prevents the deployment of complex SWCs via task mapping.

To address the aforementioned requirements, XANDAR aims at implementing a *hybrid mapping* scheme to allow for the efficient deployment of both basic and complex SWCs. As visualized in Fig. 2, the approach is based on a hybrid RTE that enables both partition and task mapping of SWCs. While complex SWCs are deployed to individual hypervisor partitions, basic SWCs can be integrated more efficiently in a shared partition, managed by an RTOS instance. Where isolation between basic SWCs is required, separate partitions are used, to which SWCs are either deployed individually or in groups of compatible SWCs on an RTOS instance.

To achieve the LET-based execution of SWCs in such a hybrid architecture, a global, hierarchical schedule needs to be derived by the toolchain. During runtime, it is carried out by time-synchronized LET management components in RTOS instances and the hypervisor, managing the activation of SWCs and implementing buffered communication behavior. For task mapping, LET management can be realized by a central management task per RTOS instance. For partition mapping, time-triggered scheduling can be implemented using scheduling mechanisms provided by the hypervisor, such as time-division multiple access (TDMA) scheduling. Buffered communication can either be implemented by extending the hypervisor itself or by introducing additional management partitions.

When generating the global schedule, significant optimization potential arises when the LET window assigned to a SWC is longer than its worst-case execution time. In this case, other SWCs can be scheduled for execution on the same processor core during that LET window. Further optimization is possible if SWCs can be scheduled outside their specified LET window without affecting data flow determinism. This optimization, however, is not possible for SWCs that interact across the system boundary. In this case, it has to be ensured that the interaction with the environment takes place deterministically. To achieve this in an LET-compatible manner, such SWCs need to provide additional code to perform hardware accesses. This code will be executed precisely at the beginning and the end of the LET window.

IV. Conclusion and Outlook

Based on requirements of next-generation embedded systems in safety-critical domains and the constraints imposed by the XANDAR development process targeting such systems, we discussed hypervisor- and RTOS-based deployment options and presented a hybrid approach that aims for combining their benefits. We then introduced central considerations in mapping SWCs to the envisioned system software architecture and identified optimization potential to achieve an efficient utilization of the target platforms. In the XANDAR project, this deployment approach is currently being implemented using the XtratuM hypervisor [5] and the Real-Time Executive for Multiprocessor Systems (RTEMS) running in single-processor mode within XtratuM partitions.

Acknowledgement

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957210.

References

[1] L. Masing, T. Dörr, F. Schade, J. Becker *et al.*, "XANDAR: Exploiting the X-by-Construction Paradigm in Model-based Development of Safety-critical Systems," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022.

[2] T. Dörr, F. Schade, L. Masing, J. Becker *et al.*, "Safety by Construction: Pattern-Based Application of Safety Mechanisms in XANDAR," in *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, in press.

[3] C. M. Kirsch and A. Sokolova, *The Logical Execution Time Paradigm*. Springer, Berlin, Heidelberg, 2012, pp. 103–120.

[4] K. C. Wang, "Embedded Real-Time Operating Systems," in *Embedded and Real-Time Operating Systems*. Springer International Publishing, 2017, pp. 401–475.

[5] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "XtratuM: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, 2009, pp. 263–272.