Modelling Software Multitasking and Multi-Core Architectures in SystemC/ OSSS

Philipp Reinkemeier¹, Philipp A. Hartmann², Achim Retberg¹

¹Carl von Ossietzky University Oldenburg, Germany philipp.reinkemeier@uni-oldenburg.de achim.rettberg@uni-oldenburg.de
²OFFIS Institute for Information Technology Oldenburg, Germany philipp.hartmann@offis.de

Abstract

Today’s embedded systems contain an increasing portion of software. Therefore the impact of platform decisions regarding hardware and the software infrastructure (OS, scheduler priorities, mapping) has to be taken into consideration already in early design phases. This requires methodologies and tools supporting the designer.

In this work we present our existing SystemC-based OSSS design flow, which enables the modelling of software multitasking at system level. We also discuss the modelling capabilities with regards to multi-core platforms and identify which aspects of the system behaviour can be captured. This paper concludes with challenges which are not solved until now and are subject to on-going research.

1 The OSSS Methodology

The OSSS methodology is based on SystemC and is characterized by an object-oriented approach for designing Hardware/Software systems. During this work we mainly focus on the software related parts. The design flow is defined by separate layers of abstraction. On entry the designer models his application as a set of tasks communicating via Shared Objects (see Figure 1).

The software tasks are equipped with properties, like a priority, an arrival time, a period and a deadline, which are definable by the user. A flexible block-based mechanism enables the user to annotate Estimated Execution Times (EETs) to code blocks and Required Execution Times (RETs), which are checked during simulation. RETs can also cover the overhead of inter-task communication, which is modelled through calls to Shared Objects. These Shared Objects allow the designer to equip a class with synchronization facilities, which is performed by arbitrating concurrent accesses and providing Guarded Methods. These methods are protected by a user-defined condition (the guard), which must hold in order to execute the method. Otherwise the execution of the process calling the method is blocked.

A more detailed description of Shared Objects is part of the OSSS documentation [1].

By manually mapping the processes and Shared Objects the Application Layer model can be refined to a Virtual Target Architecture. Tasks to be executed in software can be mapped to a run-time system, which abstracts from a real-time operating system. This run-time system is not meant to directly model RTOS primitives, but instead models the time-sharing of a processor by the mapped software tasks. For that purpose the run-time system makes use of optionally user-defined scheduling policies. As the run-time system controls the simulation of the estimated execution times annotated to code blocks, it is able to model task preemption by delaying the EETs by the time a task is interrupted by higher priority tasks, according to the scheduling policy [3].
2 Modelling Multi-Core Platforms in OSSS

While multi-core processing is now mainstream in desktop computing the shift towards multiple cores also takes place in the domain of embedded systems. Despite our framework performs well in modelling single- and multi-tasking scenarios, like depicted in Figure 2(a) and 2(b), we identified issues with regards to modelling multi-core platforms.

![Figure 2. Evolution from single- to multi-processing](image)

Application Layer: With the separation of the Application Layer from the Architecture Layer model, OSSS already provides a good starting point for targeting future many-core platforms. Especially inter-task communication via *Shared Objects* provides architecture-independent modelling of synchronization and data-dependencies.

Architecture Layer: The main mechanism for modelling software performance in OSSS software models is the EET annotation. While this already abstracts from lower level details of the CPU/core – like pipelines, branch prediction or the concrete ISA – some aspects are increasingly important on future platforms.

For example additional complex CPU features, like simultaneous multi-threading or power-saving mechanisms, like frequency scaling or sleep states can not be modelled based on static EETs, as well. The same holds for caches and memory hierarchies, which can heavily influence the execution time of basic blocks. While the impact of cache coherency and related protocols on the system’s performance depends on properties of the final implementation, this effect might not be negligible. Therefore we are working on improving our current annotations towards supporting such dynamic run-time artifacts.

Additional delay on communication will occur due to conversion of different data representations, especially in case of heterogeneous core sets. This is not yet covered in OSSS explicitly.

With regards to the platform interconnect the OSSS RMI mechanism and the OSSS Channel approach (see [2]) allow the abstract modelling of communication through method calls, which can later be mapped to physical channels imposing communication delay. Therefore these concepts are flexible enough to cope with communication topologies like for example NoCs.

Architecture Mapping In single- or multi-tasking scenarios using OSSS the mapping is performed statically during system elaboration. Considering multi-core platforms like depicted in Figure 2(c) future OS abstractions will support multiple concurrent processing units and the target RTOS might transparently handle the task↔core assignment. This will require additional information from the designer such as core affinity in order to efficiently exploit the cores.

3 Conclusion

The presented OSSS framework allows the modelling of embedded software and the exploration of different architectures at early design phases. As briefly described in Section 1 single- and multi-tasking scenarios can be handled well with OSSS.

Thanks to the separation of application and architecture models, the application layer modelling elements can also be used in multi-core scenarios. The most important extension of OSSS towards the exploration of multi-core platforms, is the separation of instrumentation hooks and the actual cost model. This would allow for annotations, which costs can be additionally determined dynamically during simulation based on the underlying architecture and the system’s state.

References

