Distributed Resource-Aware Scheduling for Multi-core Architectures with SystemC

Philipp A. Hartmann, Kim Grüttner, Achim Rettberg, Ina Podolski

Distributed Resource-Aware Scheduling for Multi-Core Architectures with SystemC

Philipp A. Hartmann[1], Kim Grüttner[1], Achim Rettberg[2], and Ina Podolski[2]

[1] OFFIS – Institute for Information Technology, Oldenburg, Germany
{hartmann,gruettner}@offis.de,
[2] University of Oldenburg
Faculty II, Department for Computer Science, Oldenburg, Germany
{achim.rettberg,ina.podolski}@iess.org

Abstract. With the rise of multi-core platforms even more complex software systems can be implemented. Designers are facing various new challenges during the development of efficient, predictable, and correct applications for such platforms. To efficiently map software applications to these architectures, the impact of platform decisions with respect to the hardware *and* the software infrastructure (OS, scheduling policies, priorities, mapping) has to be explored in early design phases.
Especially shared resource accesses are critical in that regard. The efficient mapping of tasks to processor cores and their local scheduling are increasingly difficult on multi-core architectures. In this work we present an integration of shared resources into a SystemC-based simulation framework, which enables early functional simulation and provides a refinement flow towards an implementation, covering an increasing level of platform details. We propose shared resource extensions towards multi-core platform models and discuss which aspects of the system behaviour can be captured.

Keywords: Multi-core, Resource Sharing, Platform Exploration, SystemC, Real-time, Simulation.

1 Introduction

In high-performance, desktop, and graphics processing multi- and many-core platforms are already state-of-the-art. A rise of multi-core platforms for embedded systems is not only conceivable, but is actually happening. On one hand, it enables the implementation of more functionality in software, thus exploiting the advantages of software flexibility and higher productivity. But on the other hand, this can turn into a nightmare when the new flexibility and multi-core design space needs to be limited to meet functional and non-functional system properties like real-time constraints, power consumption and cost.

To help developers during this phase of the design space exploration, efficient modelling of different architecture alternatives has to be supported by the chosen design flow. Apart from considering the underlying hardware platform, this

includes the early analysis of software and (real-time) operating system (RTOS) effects on the system's overall performance. This is important especially if multiple tasks are sharing a single processor core. Real-time properties have to be analysed and explored by choosing e.g. the scheduling policies and protocols as well as task priorities to fulfil the given set of requirements like deadlines or other application specific constraints. Furthermore, the partitioning of the application in tasks and the mapping of these tasks to the cores is complex and influences the scheduling and therefore the system performance.

An important aspect from the application's point of view is the mechanism used for inter-task communication and synchronisation. Communication via global shared memory with explicit locks for mutual exclusion impairs locality and increases coupling between tasks. Especially in the real-time domain, shared resource accesses are critical, even more so on multi-core architectures. To efficiently cope with such shared (e.g. communication) resources, task dependencies have to be considered during the mapping and scheduling phase. Depending on the target platform, dedicated hardware support for such communication primitives could be beneficial.

The contribution of this paper is the extension of OSSS for modelling software on multi-core platforms. We discuss the influence of shared resources on the execution behaviour of task sets. Since we provide a framework for early design space exploration, we do not yet capture all properties of the final platform. Instead we present a basic set of representable properties like the scheduling of parallel tasks on a multi-core execution unit, task switching, and blocking on shared resources.

In Section 3, we introduce the SystemC-based OSSS Design Methodology, with a specially focus on the modelling of embedded software for multi-cores. Based on these abstract RTOS modelling capabilities of the OSSS methodology, Section 4 covers the extension of OSSS by additional features required for supporting the distributed, scheduling approach with shared resources. In Section 5 we present our first simulation results of the extended OSSS framework for multi-core scheduling with shared resources.

Before Section 6 concludes the paper with a summary and an outlook for future research directions, we discuss the capabilities of the presented approach with respect to modelling, real-time analysis, early simulation and the further refinement flow towards an implementation.

2 Related Work

Recently published work shows the importance of new programming and abstraction paradigms for multi- and many-core systems [12]. To fully exploit the possibilities of the upcoming thousand-core chips [21], workloads of the future are already discussed [18]. To encounter these trends high-level, component-based methodology and design environment for multiprocessor SoC architectures have been proposed [15].

Many different approaches to modelling embedded software in the context of SystemC have been proposed.

Abstract RTOS models, like the one presented for SpecC in [5] are suited for early comparison of different scheduling and priority alternatives. The timing accuracy and therefore the simulation performance of this approach is limited by the fixed minimal resolution of discrete time advances. Just recently, an extension deploying techniques with respect to preemptive scheduling models very similar to the ones presented in this work has been presented in [19]. The "Result Oriented Modelling" collects and consumes consecutive timing annotations while still handling preemptions accurately similar to our "lazy synchronisation" scheme presented in [8].

Several approaches based on abstract task graphs [11,14,20] have been proposed as well. In this case, a pure functional SystemC model is mapped onto an architecture model including an abstract RTOS. The mapping requires an abstract task graph of the model, where estimated execution times can be annotated on a per-task basis only, ignoring control-flow dependent durations. This reduces the achievable accuracy.

A single-source approach for the generation of embedded SW from SystemC-based descriptions has been proposed in [3,10,17]. The performance analysis of the resulting model with respect to an underlying RTOS model can be evaluated with the PERFidiX library, that augments the generated source via operator overloading with estimated execution times. Due to the fine-grained timing annotations, the model achieves a good accuracy but relatively weak simulation performance. This interesting approach aims in the same direction as our proposed software execution time annotation.

An early proposal of a generic RTOS model based on SystemC has been published in [13]. The presented abstract RTOS model achieves time-accurate task preemption via SystemC events and models time consumption via a `delay()` method. Additionally, the RTOS overhead can be modelled as well. Two different task scheduling schemes are studied: The first one uses a dedicated thread for the scheduler, while the second one is based on cooperative procedure calls, avoiding this overhead. Although in this approach explicit inter-task communication resources are required (message queue, ...), the simulation time advances simultaneously as the tasks consume their delays.

In [9], an RTOS modelling tool is presented. Its main purpose is to accurately model an existing RTOS on top of SystemC. A system designer cannot directly use it. In this approach, the next RTOS "event" (like interrupt, scheduling event, etc.) is predicted during run-time. This improves simulation speed, but requires deeper knowledge of the underlying system.

In [23], the main focus lies on precise interrupt scheduling. For this purpose, a separate scheduler is introduced to handle incoming interrupt requests. Timing annotations and synchronisation within user tasks is handled by a replacement of the SystemC `wait()`. In [22] an annotation method for time estimation that supports flexible simulation and validation of real-time-constraints for task migration between different target processors has been presented.

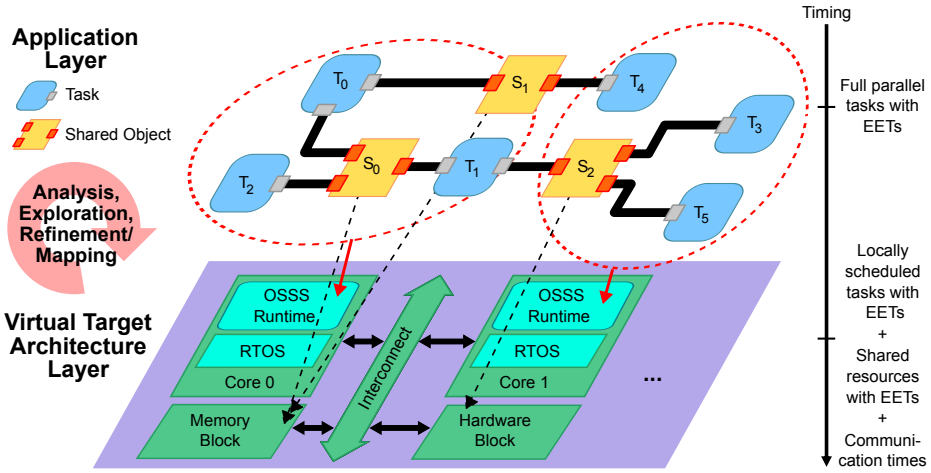


Fig. 1. Overview of the OSSS Methodology for modelling parallel Software.

In this work, we propose an extension of [7], which includes some properties of the above mentioned approaches, especially concerning a simple runtime and RTOS model. Furthermore, our model allows a separation of application, architecture and mapping. The proposed application model allows a flexible integration of shared resources for user-defined communication mechanisms via *Shared Objects* and the handling of timing (back) annotations. Our proposed extension on the architecture model includes a configurable multitasking simulation based on SystemC that allows preemptive distributed scheduling. Tasks and *Shared Objects* can be grouped together and mapped to different cores, each of them having its own local runtime. Through simulation the effects of the chosen mapping and system configuration on the functional behaviour of the task sets can be observed.

3 The OSSS Methodology for Modelling Parallel SW

OSSS defines separate layers of abstraction for improving refinement support during the design process. The design entry point in OSSS is called the *Application Layer*. By manually applying a mapping of the system's components, the design can be refined from *Application Layer* to the *Virtual Target Architecture Layer*, which can be synthesised to a specified target platform in a separate step by the synthesis tool *Fossy* [4].

The abstraction mechanisms of OSSS allow the exploration of different implementation platforms. The separation of application and platform allows different mappings and the underlying SystemC-based simulation kernel supports model execution and monitoring.

On the *Application Layer* the system is modelled as a set of parallel, communicating processes, representing software tasks (see Listing 1.1). A shared

resource in OSSS is called *Shared Object*, which equips a user-defined class with specific synchronisation facilities. *Shared Objects* are inspired by the Protected Objects known from Ada [1]. Synchronisation is performed by arbitrating concurrent accesses and a special feature called *Guarded Methods*, that can be used to block the execution of a method until an user-defined condition evaluates to true.

As a result, they are especially useful for modelling inter-process communication. User-defined Interface Method Calls (IMC), a concept well known from SystemC channels, performs communication between software tasks and *Shared Objects*. On the *Application Layer* this communication concept abstracts from the details of the underlying communication primitives, such as the actual implementation of channel across core and hardware/software boundaries. An in-depth description of the *Shared Object* concept, including several design examples, is part of the OSSS documentation [6].

```
class my_software_task : public osss_software_task {
 public:
  my_software_task() : osss_software_task() { /* ... */ }

  virtual void main() {
    while( some_condition ) // the following block has to be finished within 1ms
    OSSS_RET( sc_time( 1, SC_MS ) )
    {
      OSSS_EET( sc_time( 20, SC_US ) ) {
        // computation, that consumes 20µs
      }
      for( int i=0; i<max_i; ++i ) // estimate a data-dependent loop
        OSSS_EET( sc_time( 100, SC_US ) ) {
          // loop body
      }
      if( my_condition ) {
        // communication only outside of EET blocks
        result = my_port_to_shared->my_method();
      }
    } // end of RET block and loop
  }
};
```

Listing 1.1. Example of a software task eith estimated and required execution time annotations.

A proper modelling of software requires the consideration of its timing behaviour. In OSSS, the **E**stimated **E**xecution **T**ime (EET) of a code block can be annotated within *Software Tasks* and *Shared Objects* using the `OSSS_EET()` block annotation. In addition to the EETs, OSSS enables the designer to specify local deadlines for a specific code block. The **R**equired **E**xecution **T**ime (RET)

is specified by the `OSSS_RET()` block annotation, which observes the duration of the marked code block. If required, RETs can be nested at arbitrary depth. The consistency of nested RETs is checked during the simulation as well as a violation of the RETs. If such a timing constraint is violated during the simulation, it is reported. Unmet RETs may arise from (additional) delays caused by blocking guard conditions, or simply unexpectedly long estimated execution times (e.g. `max_i` ≥ 9 in Listing 1.1).

4 Multi-core Scheduling with Shared Resources

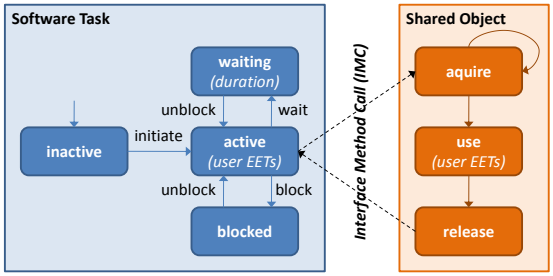
In this paper, we focus on the abstract modelling capabilities of OSSS for embedded software, especially targeting multi-core platforms. Here, the OSSS model is *not* meant to directly represent existing real-time operating system (RTOS) primitives. Instead, the *Software Tasks* in OSSS are meant to *run* on top of a rather generic (but lightweight) run-time system (see Fig. 1), where the synchronisation and inter-task communication is modelled with *Shared Objects*.

In a refinement step the *Application Layer* model is mapped to the *Virtual Target Architecture*. Each task is then mapped to a specific core, each of which provides a distinct run-time, to improve locality and reduce the coupling between different cores, as shown in Fig. 1. Tasks have may have statically or dynamically assigned priorities, according to a given scheduling policy for each core, an initial startup time, optional periods and deadlines.

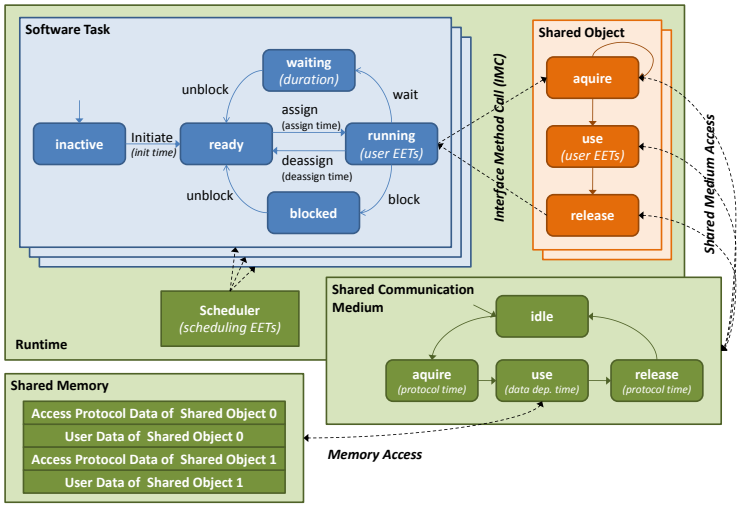
During simulation, the tasks can be in different states as shown in Fig. 2. We distinguish between the full parallel *Application Model* and the core mapped *Virtual Target Architecture Model* task state machines. In the *Application Model* a task can either be `running`, `waiting` or `blocked`. The distinction between `blocked` and `waiting` has been introduced to ease the detection of deadlocks. A task in the `waiting` state will enter the `running` state after a given amount of time (duration), whereas a `blocked` task can only be de-blocked, once the access to a shared resource is granted. In the `running` state, a task might access a *Shared Object* through IMC. This either leads to the acquisition of its critical section (`use` state) or a suspension in the `blocked` state. In this state the task tries to reacquire the shared resource until it gets access.

The execution times of certain code blocks can be annotated flexibly, to introduce control-flow dependent time consumption, as shown in Listing 1.1.

In the *Virtual Target Architecture Model Software Tasks* and *Shared Objects* are grouped and mapped onto runtimes of the cores. During the simulation, the OSSS software runtime abstraction handles the time-sharing of a single processor core by several *Software Tasks*, which are bound to this OS instance. Therefore, a `ready` state has been introduced. A scheduler for handling the time-sharing is attached to the set of mapped tasks. Several frequently used scheduling policies are already provided by the simulation library, like static priorities (preemptive and cooperative), or earliest-deadline first. Additionally, arbitrary user-defined scheduling policies can be added. The RTOS overhead of context switches (assign & deassign times) and execution times of scheduling decisions can be annotated



(a) Application Layer Model



(b) Virtual Target Architecture Model

Fig. 2. Task states and transitions (terminate edges omitted)

as well. With this set of basic elements, the behaviour of the real RTOS on the target platform can be modelled.

To improve the real-time capabilities, *Guarded Methods* that can lead to arbitrary blocking times due to data-dependent conditions, are ignored. Instead, only the guaranteed mutual exclusive access to Shared Objects is used for synchronisation and communication between the tasks. Each method of such a Shared Object can then be considered as a critical section, which is executed atomically. Intra-core communication, i.e. communication between tasks mapped to the same core, can be handled as usual. Here, the accesses are ordered according to the local scheduling policy.

Moreover, the *Virtual Target Architecture Model* allows incorporating the effects of a shared memory that is connected to the cores via a shared communication medium. In an implementation on a target architecture the access

protocol data, as well as the user data of a *Shared Object* are mapped to a specific location in a shared memory. Therefore, all states of the *Shared Object* include a certain overhead of shared medium acquisition, usage and release. These times could also be annotated to the proposed simulation model, but are not in the focus of this paper. We also do not cover effects of instruction and data fetches over the shared communication medium, assuming that each core has its local data and instruction memory.

5 Experiments

The main purpose of the modelling of abstract software multitasking in early design phases is the exploration of the impact of platform choices on the system's correctness and performance. In the context of multi-core architectures, accesses to shared resources (modelled as Shared Objects) have to be considered carefully. Distributed access from different cores and runtimes to the same resource has to be orchestrated. Different strategies are possible and lead to quite different behaviours during run-time. An early simulation of these cross-dependencies helps during the development of the application.

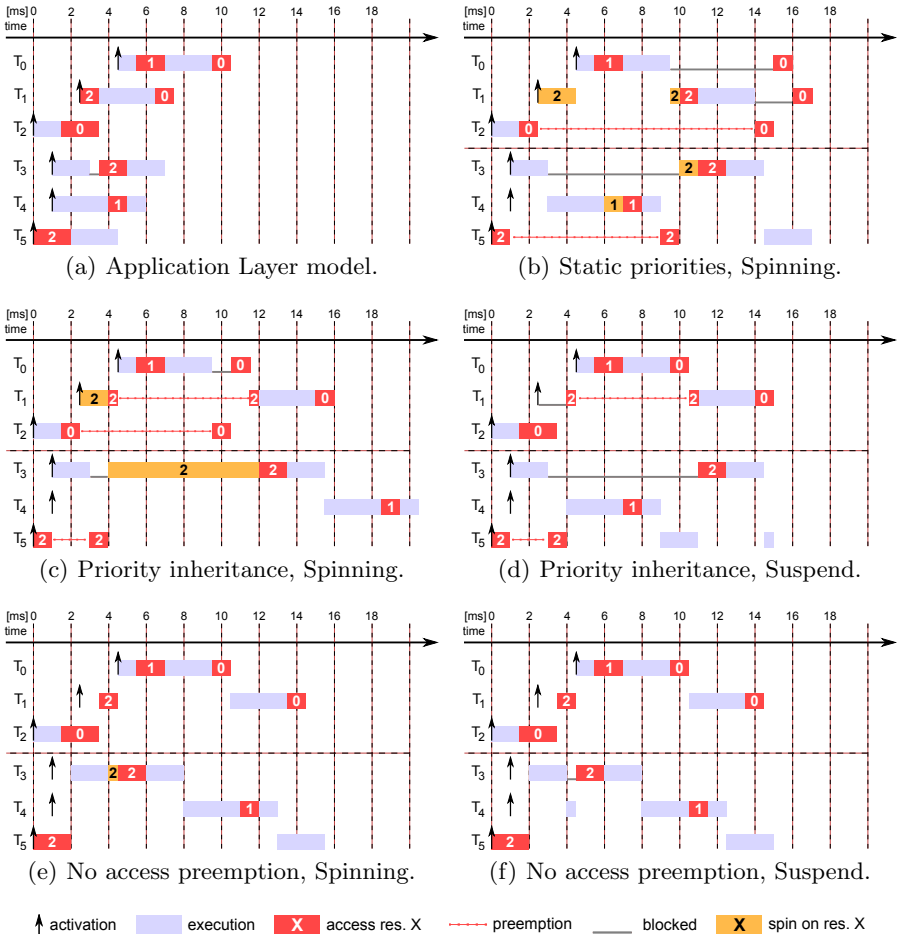
In Fig. 3, several combinations of local and distributed access policies are compared for the application and mapping example shown in Fig. 1: Six tasks are mapped on two cores, accessing three Shared Objects.

In Fig. 3(a), the Application Layer model of the system is depicted. In this initial model, no local scheduling policy is enforced, which leads to independently running tasks. The only blocking times occur in case of conflicting accesses to Shared Objects. This model already exhibits the execution times (and periods within critical regions inside the Shared Objects), according to the task arrival times, the EETs and the access patterns of tasks to resources.

Next, static priorities are assigned to the tasks ($T_0 > T_1 > T_2, T_3 > T_4 > T_5$) and the tasks are mapped to different cores, following Fig. 1. The scheduling policy is always assumed to be priority-based, either with or without support for local priority inheritance. Inter-core accesses to shared resources are resolved based on the set of pending requests (see Section 3). In the example, it is assumed that tasks on Core 0 have a higher priority, than those running on Core 1.

In the various scenarios, different access strategies with respect to the shared resources are compared, according to their impact on the overall system scheduling. For local resource accesses, i.e. resources that are accessed from tasks within the same core, task preemption is allowed in Fig. 3(b)–(d), and suppressed in Fig. 3(e)–(f). In case, a shared resource is currently locked by another core, the calling task can either try to do busy-waiting until the resource is available again (Spinning, in (b), (c), (e)), or stop its execution to let other tasks execute on the current core (Suspend, in (d), (f)). It is then assumed, that the runtime is able to resume the task, as soon as the blocked resource is available again.

For the given task set and mapping, the different execution traces that can be obtained by the OSSS Multi-Core Software simulation in the different scenarios are shown. Tasks are assigned to their cores according to the local priority based



(a) Application Layer model.
(b) Static priorities, Spinning.
(c) Priority inheritance, Spinning.
(d) Priority inheritance, Suspend.
(e) No access preemption, Spinning.
(f) No access preemption, Suspend.

Fig. 3. Different scheduling scenarios with shared resources.

scheduling policy. Different run-time artefacts can be observed (in addition to potential RET violations, which are not shown here).

If a task that is currently accessing a shared resource can be preempted by the runtime system due to the arrival of a higher-priority task (or its availability due to resource grants), so called "priority inversion" can occur. In Fig. 3(b), this can be observed on both cores, when tasks T_2, and T_4 get access to their cores, although higher-priority tasks T_0, and T_3 are waiting for the Shared Objects S_0, and S_2, respectively. This leads to longer blocking times for these high-priority tasks.

In the context of a single core, priority inheritance [2] is known to be a solution for such scenarios. With priority inheritance, the low-priority tasks holding resources required by high-priority tasks get an elevated priority, which reduces

their lock times. In the context of multi-cores, a local priority inheritance implementation may lead to even worse scenarios, as shown in Fig. 3(c). First of all, the response time of T_0 is reduced, since T_2 can continue until the release of S_0, once T_0 requests S_0. But the resource S_2 has been locked by the arrival of T_1 on Core 0, before T_3 could obtain S_2 on Core 1. Core 1 is subsequently busy waiting on S_2, which is held by the preempted task T_1 on Core 0. Overall, the response time of the latest task is now significantly worse.

An approach towards better CPU utilisation in the context of shared resources might be the suspension of tasks, blocked by conflicting inter-core accesses, as shown in Fig. 3(d). The overly long spinning time of T_3 and even the preemption of the access to S_0 by T_2 is avoided in this example.

Instead of suspending inter-core blocked tasks, an orthogonal approach to reduce blocking times between cores is to suppress the preemption of local, lower-priority tasks, that are currently accessing a shared resource. The results of this access strategy are shown in Fig. 3(e),(f). Both traces lead to very good overall response times with nearly no blocking times. The high-priority tasks are of course started with an additional delay, depending on the currently ongoing resource accesses. But since resource occupation should be kept short anyhow, this might be a feasible strategy. The spinning time on S_2, that can be observed in Fig. 3(e) is quite short. Since in case of the suspension strategy, runtime overhead costs are excluded for simplicity, the slightly better result in Fig. 3(f) might be misleading. A refined model should consider these overheads as well.

6 Conclusion and Future Work

In this paper, we have presented the current modelling capabilities for embedded software of the OSSS hardware/software design methodology, especially focussing on multi-core platforms.

OSSS features a layered approach with a separation between an abstract Application Layer which can later be mapped to a Virtual Architecture Layer. This separation enables flexible exploration of different (software) architecture variants already at early phases in the design process, e.g considering scheduling policies, priorities, resource access strategies, etc.

After a general overview of the current OSSS Software Modelling approach in Section 3, some of the required extensions to the existing methodology towards abstract, but more accurate multi-core system models have been discussed in Section 4. For a set of distributed multi-tasking systems, the OSSS approach is an expressive and suitable modelling approach for applications running on top of such platforms. Due to the explicitly visible resource sharing, expressed by using Shared Objects, the resulting synchronisation and communication overheads and conflicts can be observed already in early simulation models.

In Section 5, a simple Application Layer model has been mapped to a multi-core platform. Since distributed resource accesses are critical for the overall system behaviour, several different access strategies, both regarding local scheduling decisions (priority inheritance, atomic/nonpreemptable resource accesses) as well

as the handling of remotely blocked resources (Spinning, Suspension) have been compared. It can be seen, that even for small and allegedly simple cases, the resulting system behaviour is hard to predict. Therefore, early simulation of the different alternatives is a valuable analysis tool for a designer.

Regarding an implementation on a real multi-core platform, the proposed access strategies require different platform primitives, depending on the intended implementation approach. As proposed in Section 4, an implementation purely in terms of a shared memory region with a software implementation of the access protocol is possible. For the support of an suspend-based access strategy, platform support for the reactivation of suspended tasks on a certain core is required, e.g. via sending an interrupt from the core, that releases a given resource to all cores, waiting for said resource. An initial implementation based on a Linux implementation of the OSSS runtime will be published separately.

In the context of real-time applications, it is even more difficult to give guarantees, when considering shared resources as well. We intend to further extend the presented resource access protocols to improve the static analysability of OSSS system models. This includes a restricted task/object model, e.g. by omitting user-defined guard conditions, which can lead to arbitrary, data-dependent blocking times. Future work is to study real-time scheduling approaches for multi-cores as discussed in [16]. These scheduling aspects can be integrated into the OSSS methodology.

Summarising can be said, that OSSS already provides a good starting point for modelling, exploring, refining, and implementing applications on emerging multi-core platforms. Further extensions are possible and promising to improve these capabilities even more.

References

1. Burns, A., Wellings, A.: Concurrency in Ada. Cambridge University Press (1997)
2. Buttazzo, G.C.: Hard Real-time Computing Systems. Kluwer Academic Publishers (2002)
3. Fernandez, V., Herrera, F., Sanchez, P., Villar, E.: Embedded Software Generation from SystemC, chap. 9, pp. 247–272. Kluwer (Mar 2003)
4. Fossy – Functional Oldenburg System Synthesiser, `http://fossy.offis.de`
5. Gerstlauer, A., Yu, H., Gajski, D.: RTOS Modeling for System Level Design. In: Proceedings of Design, Automation and Test in Europe. pp. 47–58 (2003)
6. Grüttner, K., Andreas, H., Hartmann, P.A., Schallenberg, A., Brunzema, C.: OSSS - A Library for Synthesisable System Level Models in SystemC™ (2008), `http://www.system-synthesis.org`
7. Hartmann, P.A., Reinkemeier, P., Kleen, H., Nebel, W.: Modeling of Embedded Software Multitasking in SystemC/OSSS, LNEE, vol. 36, chap. 14, pp. 213–226. Springer (2009)
8. Hartmann, P.A., Reinkemeier, P., Kleen, H., Nebel, W.: Efficient modelling and simulation of embedded software multi-tasking using SystemC and OSSS. In: Forum on Specification, Verification and Design Languages, 2008. FDL 2008. pp. 19–24 (Sep 2008)

9. He, Z., Mok, A., Peng, C.: Timed RTOS modeling for Embedded System Design. Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE pp. 448–457 (Mar 2005)
10. Herrera, F., Villar, E.: A Framework for Embedded System Specification under Different Models of Computation in SystemC. In: Proceedings of the Design Automation Conference (2006)
11. Huss, S.A., Klaus, S.: Assessment of Real-Time Operating Systems Characteristics in Embedded Systems Design by SystemC models of RTOS Services. In: Proceedings of Design & Verification Conference and Exibition (DVCon'07). San Jose, USA (2007)
12. Hwu, W.m., Keutzer, K., Mattson, T.G.: The concurrency challenge. IEEE Design and Test of Computers 25(4), 312–320 (2008)
13. Le Moigne, R., Pasquier, O., Calvez, J.P.: A Generic RTOS Model for Real-time Systems Simulation with SystemC. Design, Automation and Test in Europe Conference, 2004. Proceedings 3, 82–87 Vol.3 (16-20 Feb 2004)
14. Mahadevan, S., Storgaard, M., Madsen, J., Virk, K.: ARTS: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality. 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems pp. 480–483 (Sep 2005)
15. O.Cesário, W., Lyonnard, D., Nicolescu, G., Paviot, Y., Yoo, S., A.Jerraya, A., Gauthier, L., Diaz-Nava, M.: Multiprocessor soc platforms: A component-based design approach. IEEE Design and Test of Computers 19(6), 52–63 (2002)
16. Podolski, I., Rettberg, A.: Overview of multicore requirements towards real-time communication. In: Lee, S., Narasimhan, P. (eds.) Software Technologies for Embedded and Ubiquitous Systems, 7th IFIP WG 10.2 International Workshop (SEUS). LNCS, vol. 5860, pp. 354–364. Springer (2009)
17. Posadas, H., Herrera, F., Fernandez, V., Sanchez, P., Villar, E.: Single Source Design Environment for Embedded Systems based on SystemC. Transactions on Design Automation of Electronic Embedded Systems 9(4), 293–312 (Dec 2004)
18. Rabaey, J.M., Burke, D., Lutz, K., Wawrzynek, J.: Workloads of the future. IEEE Design and Test of Computers 25(4), 358–365 (2008)
19. Schirner, G., Dömer, R.: Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling. In: Proceedings of Design, Automation and Test in Europe (DATE 2008). pp. 122–127. Munich, Germany (March 2008)
20. Streubühr, M., Falk, J., Haubelt, C., Teich, J., Dorsch, R., Schlipf, T.: Task-Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures. In: Proceedings of the Design, Automation and Test in Europe Conference. pp. 480–481. European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2006)
21. Yeh, D., Peh, L.S., Borkar, S., Darringer, J., Agarwal, A., mei Hwu, W.: Thousand-core chips. IEEE Design and Test of Computers 25(3), 272–278 (2008)
22. Zabel, H., Müller, W.: An Efficient Time Annotation Technique in Abstract RTOS Simulations for Multiprocessor Task Migration. In: Kleinjohann, B., Kleinjohann, L., Wolf, W. (eds.) DIPES. IFIP, vol. 271, pp. 181–190. Springer (2008)
23. Zabel, H., Müller, W., Gerstlauer, A.: Accurate RTOS Modelling and Analysis with SystemC. In: W. Ecker, W. Mueller, R. Doemer (eds.) "Hardware Dependent Software – Principles and Practice". Springer-Verlag (2009)