UC Irvine
UC Irvine Previously Published Works

Title
A program state machine based virtual processing model in SystemC

Permalink
https://escholarship.org/uc/item/4gm549d6

Journal
ACM SIGBED Review, 11(4)

Authors
Schmidt, Tim
Grüttner, Kim
Dömer, Rainer
et al.

Publication Date
2015-01-22

DOI
10.1145/2724942.2724943

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

A Program State Machine Based Virtual Processing Model in SystemC

Tim Schmidt[1], Kim Grüttner[2], Rainer Dömer[1], Achim Rettberg[3]
[1]University of California, Irvine, USA
[2]OFFIS – Institute for Information Technology, Oldenburg, Germany
[3]Carl von Ossietzky University Oldenburg, Germany

ABSTRACT

The Program State Machine (PSM) Model of Computation offers a rich set of modeling elements to describe behavioral and structural hierarchy, concurrency, synchronization, state transitions and timing. With the rising software complexity of today's embedded systems, the use of Real-Time Operating Systems (RTOS) has become state-of-the-art for nearly all System-on-Chip designs. Regrettably, the PSM model itself has insufficient support for the specification of the preemptive dynamic scheduling behavior of an RTOS. In this paper, we propose a model for dynamically dispatching PSM models on a virtual processing element. Our model aims to abstract from the targeted RTOS and the processor core through execution time annotations and a flexible preemptive scheduler model. Mapping a PSM model to a set of scheduled virtual processing elements only requires minor model transformation and enables early exploration of different processing element mappings and scheduling policies. Our virtual processing model for PSMs is realized on top of the SystemC library. We evaluate the proposed virtual processing model using a Canny edge detection filter.

1. INTRODUCTION

The development process of state-of-the-art embedded systems is complex and affects many different disciplines. Among others, the design process requires *hardware* and *software* design decisions. Today's system complexity of embedded Multi-Processor System-on-Chip (MPSoC) designs is continuing at an almost exponential rate [2]. The strongly growing complexity includes the *integration* of the functionality as well as the associated *software complexity*. The development of hardware is associated with immense costs. Consequently, whenever possible, designers prefer software solutions and realize algorithms on software processors.

To cope with the increasing complexity and the time-to-market pressure, new *design methodologies* are required. One design challenge for embedded system designers is mapping the functionality on the individual processing elements while meeting the required extra-functional properties (e.g. timing and power consumption) at minimal cost. To support this challenge, System-Level Design Languages (SLDLs) enable to raise the level of abstraction and support *early* design decisions. In [3], different abstraction levels have been proposed. The *specification level* enables untimed modeling of functionality and causality between behaviors in an executable model. Behaviors can be statically composed in a sequential order, as finite-state machine or parallel. Communication between functions is described using double handshake channels with message passing and shared variables. The *architecture level* introduces processing elements (PE) that execute behaviors in sequential order. Behaviors are annotated with delays to specify the estimated execution times on the PEs. Communication between PEs is described through message passing communication with annotated delays. The *implementation level* adds instruction and cycle accurate timing for PEs and signal level protocols with cycle accurate communication times.

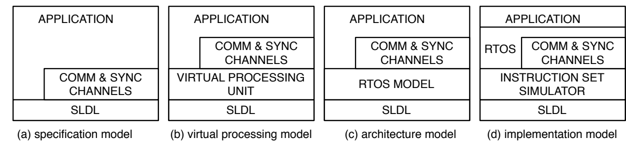

Figure 1: Extension of a virtual processing model (see [4])

In this work we focus on systems that implement their behavior in software, which can be mapped and executed on different PEs of an MPSoC. Fig. 1 (a), (c) and (d) show the layers for executable MPSoC models proposed in [4]. All models are executed on top of a *SLDL*, e.g. SpecC or SystemC. The *Application* is user-defined behavior to be executed on the MPSoC. The layers between Application and SLDL introduce communication, scheduling and timing techniques to enable a stepwise refinement from the specification level down to the implementation.

The design step from a non-scheduled *specification model* (a) to a complete RTOS scheduled *architecture model* (c) is a complex refinement step. It requires to transform the application or behavioral description into a process- or thread-based model and to use the configuration and scheduling primitives of the selected RTOS. At this time the choice of the task granularity and the supported scheduling primitives have usually been taken. At this level, the comparison of different task granularities and different scheduling policies, supported by different RTOSs induces major redesign effort. In this paper, we introduce a *virtual processing model* (b) to support a smoother refinement from the unscheduled *specification* to an RTOS scheduled *architecture model* for PSMs.

Our approach supports PSM modeling at the specification layer and enables early estimation of dynamic scheduling

EWiLi'14, November 2014, Lisbon, Portugal.
Copyright retained by the authors.

effects when mapping parallel behaviors on the same PE. This step can be performed without any behavior to process or RTOS specific refinements. This way, designers can profit from simple specification model modifications in combination with early estimated execution time annotations, thus enabling early decisions regarding PE allocation, behavior to process refinement granularity, process to PE binding and scheduling policy selection.

The rest of the paper is organized as follows. In Section 2 we analyze related work. Next, we discuss the design of the *virtual processing model* in Section 3. Followed by a brief description of the implementation in Section 4, we evaluate the new *virtual processing model* using a Canny filter for still image edge detection in Section 5. Finally, we conclude our work in Section 6.

2. RELATED WORK

An early proposal of a generic RTOS model based on SystemC has been published in [11]. The presented abstract RTOS model achieves time-accurate task preemption via SystemC events and models time passing via a `delay()` method. The RTOS overhead can be modeled as well. Two different task scheduling schemes are studied: the first one uses a dedicated thread for the scheduler, while the second 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 [8], an RTOS modeling 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" (interrupt, scheduling event, etc.) is predicted during run-time. This improves simulation speed, but requires deeper knowledge of the underlying system.

An RTOS based scheduling approach with focus on precise interrupt scheduling has been proposed in [17]. For this purpose, a separate scheduler is introduced to handle incoming interrupt requests. Timing annotations and synchronization within user tasks are handled by a replacement of the SystemC `wait()`. In [16] an annotation method for time estimation has been presented that supports flexible simulation and validation of real-time constraints for task migration between different target processors. The concept allows preemptive scheduling in the context of priority-based scheduling, supporting nested interrupts.

All mentioned solutions above work on architecture level models and allow to create and handle processes and interrupts on an RTOS specific abstraction. Our solution addresses scheduling at a higher abstraction level and keeps communication at specification abstraction, thus no need for interrupts.

Several approaches based on abstract task graphs [9, 10, 12, 15] 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.

The proposed RTOS model in [4] can be implemented on top of any SLDL (see Fig. 1) that supports the concepts of process handling and time modeling. An extension of this approach [13] presents a high-level, host-compiled multi-core RTOS simulator. This multi-core processor model can run more than one process simultaneously which can be organized by a separate ready queue per core (Asymmetric Multi-Processing) or one global ready queue (Symmetric Multi-Processing) used for dynamic process to core dispatching. The proposed extension supports the concept of Transaction Level Modeling (TLM) for intra-core communication. Both solutions focus on a process level RTOS abstraction at the architecture and implementation level including features like process creation and interrupt handling. In contrast, our proposed approach avoids process-level RTOS operations and operates on an estimated execution time annotated specification model. Moreover, our solution keeps communication abstract and each processing element has its own ready queue. After exploration, based on our *virtual platform model*, we can transform the scheduled specification model into an RTOS model on the architecture level.

The timing accuracy and therefore the simulation performance of [4, 13] is limited by the fixed minimal resolution of discrete time advances. An extension deploying techniques with respect to preemptive scheduling models has been presented in [14]. The *Result Oriented Modeling* collects and consumes consecutive timing annotations while still handling preemptions accurately.

A two layer for modeling approach for software task scheduling considering shared resources has been proposed in [6, 7]. The design starts with an *Application Layer* (AL) model, which describes the functionality in terms of software tasks, hardware modules and shared communication objects. These modeling elements are mapped on modeling elements of the *Virtual Target Architecture Layer* (VTAL): software processing elements with an RTOS model similar to [14], hardware processing elements with fixed static scheduling, memories and SystemC TLM for modeling shared buses and point-to-point communication channels. Communication is realized via Remote Method Invocation (RMI) via shared buses or dedicated point-to-point channels. The individually mapped software tasks can be annotated with Estimated Execution Time (EET) blocks that represent computation time. The design flow is supported with preemptive and cooperative scheduling strategies, as well as deadline driven strategies. This approach covers specification and architecture level modeling. The main difference to our approach is that the RTOS model works with explicit tasks (i.e. processes). Our model could be refined as well to [6, 7] after PSM scheduling exploration.

3. VIRTUAL PROCESSING MODEL

3.1 Basic Modeling Elements

We use an expressive subset of the program state machine (PSM) model of computation (MoC) to describe the functionality of a system. A hierarchical PSM model with the corresponding thread graph is shown in Fig. 2(a) and (b). A *sequential* composition of n behaviors describes a total execution order, denoted as a n dimensional tuple: $(beh_1, ..., beh_n)$. The execution starts with $beh_1$and finishes with beh_n. A *parallel* composition of n behaviors describes a partial execution order, denoted as a set of n behaviors $\{beh_1, ..., beh_n\}$. The parent behavior of a parallel composition of child behaviors will not finish until all child behaviors have completed (Fork-Join semantics). The *finite-state machine* behavior composition is a special case of a sequential execution from a start state to an end state. The execution order is defined by state transitions.

The *virtual processing model* supports communication between behaviors via *double handshake channels* (synchronized) and shared variables (unsynchronized). A double handshake channel operates in *rendezvous* fashion (see Fig. 3). When the data is transferred from the *sender* to the *receiver*, both behaviors resume their execution at the *same* time. A

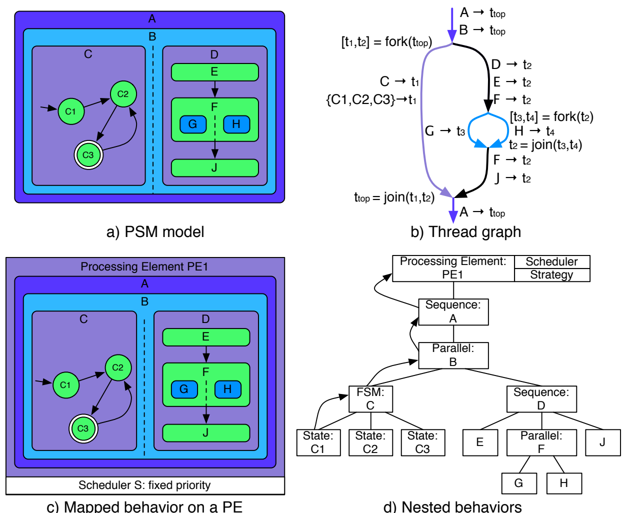

Figure 2: PSM model (a), which is mapped on a PE (c), with corresponding thread graph (b) and nesting tree (d)

communicating behavior can be *blocked* through communication for some time, until the other communication partner is ready and the handshake has been completed. To achieve a high utilization of a PE the scheduler will be able to *preempt* blocked behaviors.

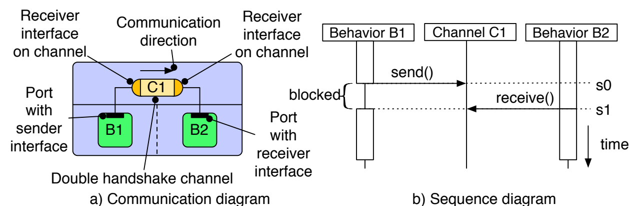

Figure 3: Double handshake protocol

3.2 Processing Elements

A PE maintains a single thread of execution (i.e. a single core processor). We associate exactly one scheduler with one PE one scheduling algorithm. We use the terms *simulated time* and *simulation time* to express the amount of task execution time currently simulated. The terms *simulation execution time* and *execution time* refer to the amount of time the simulator requires on the host computer[1]. Fig. 2(c) shows the scheduler *S* which is assigned to *PE1* and associated with the *fixed priority* scheduling strategy. In this case all behaviors mapped on *PE1* need a specific *fixed priority*, such that the scheduler can make a scheduling decision. Furthermore, we assume that *computation* is only in *leaf* behaviors (*C1*, *C2*, *C3*, *E*, *G*, *H* and *J* in Fig. 2), and *hierarchical* behaviors (*A*, *B*, *C*, *D*, and *F* in Fig. 2) describe the *causal chain* of execution in the model that must be followed by the scheduler.

3.3 Scheduling

We describe now the concept of scheduling for PSM models for the two requested *fixed priority* and *round robin* scheduling algorithms. We decided to provide these two fundamental strategies because more complex strategies can be easily derived from them.

3.3.1 Fixed Priority

The fixed priorities are statically defined. The scheduler always executes the behavior that has the *highest* priority and is *ready* to execute. We are interested in making the process of priority assignment to the behaviors on the *virtual processing model* as simple as possible. The designer assigns fixed priorities *only* to leaf behaviors. A hierarchical higher behavior *cannot* hide the priority of a *leaf* behavior.

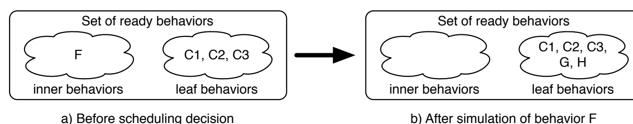

Figure 4: Differentiation between the priority of inner and leaf behaviors (example based on Fig. 2)

The scheduler distinguishes between *inner* and *leaf* behaviors. As shown in Fig. 4(a), the set of ready behaviors can be categorized into *two* subsets. *Inner* behaviors always have an infinite high priority and *leaf* behaviors have a fixed priority defined by the designer. The scheduler *always* prefers an *inner* behavior over a *leaf* behavior. Let's assume the behaviors *C1*, *C2*, *C3*, and *F* are ready to execute. In this situation, the scheduler selects behavior *F* because *F* is an *inner* behavior (see Fig. 2(c)) and has *infinite* priority. Behavior *F* has two child behaviors *G* and *H*, which are *leaf* behaviors. Behavior *F* is waiting until the child behaviors have completed. Fig. 4(b) shows both child behaviors *G* and *H* added to the set of behaviors ready to execute.

3.3.2 Round Robin

All mapped behaviors on a processing element get time slices of the *same* length. If a behavior has terminated, the scheduler selects *immediately* the next running behavior. The scheduler executes the behaviors in a *circular* order. If a behavior is *not* ready to execute, the next ready behavior with respect to the circular iteration is chosen. Fig. 5 depicts a round robin scheduling example. The parallel behaviors *G* and *H* are mapped on the same processing element. The behaviors *G* and *H* are scheduled by *round robin* and the time slice is 5 time units. In the following, we keep the focus on behavior *G* that requests once 3 and once 12 time units. The start behavior is arbitrary because the model in the figure does not define one; we assume the simulation starts with behavior *G*. Behavior *G* requests 3 time units, computes, and requests 5 more time units. The request of 3 time units can be consumed *completely* in one time slice; however, the following request is too complex. 2 more time units can be consumed after behavior *H* *preempts* behavior *G* and can start executing. At time 10, behavior *G* is active again and continues consuming the remaining 10 time units.

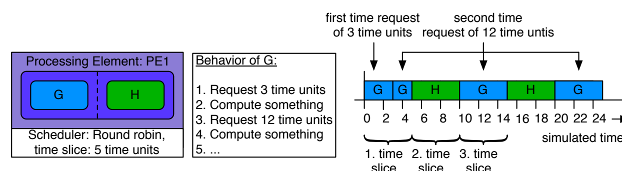

Figure 5: Round robin scheduling

Depending on the scheduling algorithm and the commu-

[1] Which is of course dependent on the host CPU, clock frequency etc. and a comparison between simulation execution times of different model is only possible on the same reference simulation host.

nication status, a behavior can have one of the following states (see Fig. 6): *ready*, *running*, *communicating* and *waiting*. A *ready* behavior can be selected by the scheduler and executed on the associated processing element. A behavior has the state *running*, if it is currently executing on the mapped processing element. A *running* behavior can be preempted in two different ways: (a) end of the time-slice, as defined by the scheduling algorithm (*waiting* state), (b) blocked communication request on double handshake channel (*communicating*). If a behavior has completed, its status is *terminated*. Fig. 6 shows all possible transitions between the described states.

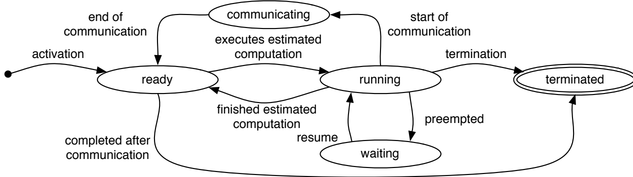

Figure 6: State automaton of a behavior

4. IMPLEMENTATION

4.1 Processing Element

The class `osss_processing_element` represents a PE and inherits from the `osss_behaviour`. Fig. 7 shows the extension of the *OSSS-Behaviour* class diagram [5]. The designer derives a class from `osss_processing_element` class and defines in the constructor the execution order of the mapped behaviors on the highest hierarchical level.

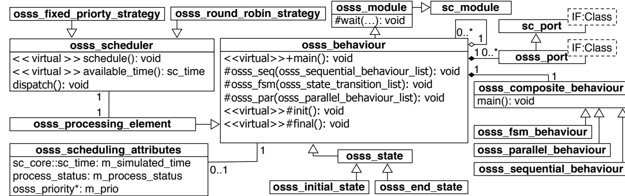

Figure 7: *OSSS-Behaviour* class extensions (bold boxes)

In the following, we describe how the basic scheduling algorithms are designed to support *preemption* and *communication*.

4.2 Simulation of Time

Fig. 8 shows an example where the behaviors $B1$ and $B2$ are mapped on the same PE under *fixed priority* scheduling strategy. Thread $t1$ is associated with behavior $B1$ and thread $t2$ with behavior $B2$. We assume behavior $B2$ is *running* and $B1$ *ready* (see Fig. 6) at the beginning and neither $B1$ nor $B2$ have consumed any time. Thread $t2$ starts executing the `main()` function of $B2$ and enters the 3 milliseconds estimated `waste_time()` function of the scheduler, see Alg. 1. The **while** loop runs until the entire requested time of a timing annotation has been consumed. At the beginning of the loop, thread $t2$ calls the function `available_time(requested_time)`, which asks the *scheduling strategy* how much of the totally requested time can be consumed. In our example, the scheduling strategy is *fixed priority* and the current behavior has the highest priority. In this case, the fixed priority scheduler accepts the complete time (i.e. 3 milliseconds).

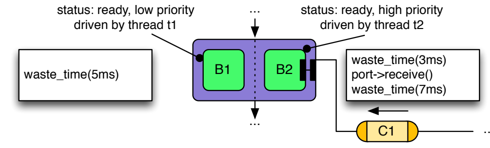

Figure 8: Scheduling with communication

We allow preemption of *running* behaviors. The function then of preemption is to *suspend* the current running process and to *resume* a process the scheduling algorithm has selected to be executed next. If we *suspend* a process waiting on a timed event,we have to interrupt the waiting process, store how much time the process has already consumed, and consume the remaining time later. For this reason, the SystemC `wait()` statement in Line 6 is sensitive to two different *or-composed* sets of events. The first event `max_time` notifies the thread after the provided time slice of the scheduler is over. The second parameter is an *or-composed* event list. All behaviors that start communication *add* the corresponding synchronization event of the channel to that list. This mechanism allows *preempting* a current running behavior by a behavior with higher priority that has completed communication.

Algorithm 1 function *waste_time(requested_time)*

```
1:  req_time ← requested_time
2:  while req_time > 0 do
3:     max_time ← available_time(requested_time)
4:     start_time ← current_time
5:     behavior_status ← running
6:     wait(max_time or registered_communication_events)
7:     req_time ← req_time - (current_time - start_time)
8:     if req_time = 0 then
9:        return
10:    else
11:       behavior_status ← waiting
12:       dispatch()
13:    end if
14: end while
```

Each PE represents a single core processor. For this reason, only one behavior can be *ready* for the SLDL scheduler. Otherwise, the scheduler would execute multiple behaviors in parallel on the same PE. Communicating behaviors are an exception because they are *waiting* for their synchronization event. If the process of a communicating behavior would be *suspended*, the behavior would ignore the synchronization event. The function `dispatch()` (see Alg. 2) guarantees this requirement. The set of behaviors is stored in two lists, namely a list for *inner* behaviors and a list for *leaf* behaviors. The first behavior in the list of *ready* behaviors defines the next *running* behavior. In this situation behavior $B1$ needs to be *suspended* and behavior $B2$ should be *ready*.

4.3 Scheduling Strategy

We decided to separate the scheduler and the scheduling strategies (see Fig. 7). The designer derives a class from `osss_scheduler`. The function `schedule()` takes a list of all leaf behaviors as argument. The function moves the next executing behavior to the beginning of the list. The function `available_time` defines the size of a consumable time quantum.

5. EXPERIMENTS AND EVALUATION

In order to evaluate our proposed *virtual processing model*, we focus on scheduling different partitions of a specification

Algorithm 2 function *dispatch()*

```
1:  Unsorted List: inner_behaviors, leaf_behaviors
2:  Process successor_process ← null
3:  Process current_process ← get_current_process()
4:  if inner_behaviors = ∅ then
5:      schedule(leaf_behaviors)
6:      successor_process ← first_element(leaf_behaviors)
7:  else
8:      successor_process ← first_element(inner_behavior)
9:  end if
10: for all Behavior b in inner_behavior do
11:     if process(b) ≠ current_process and
        status(b) ≠ communicating then
12:         suspend(process(b))
13:     end if
14: end for
15: if current_process ≠ successor_process then
16:     resume(successor_process)
17:     suspend(current_process)
18: end if
```

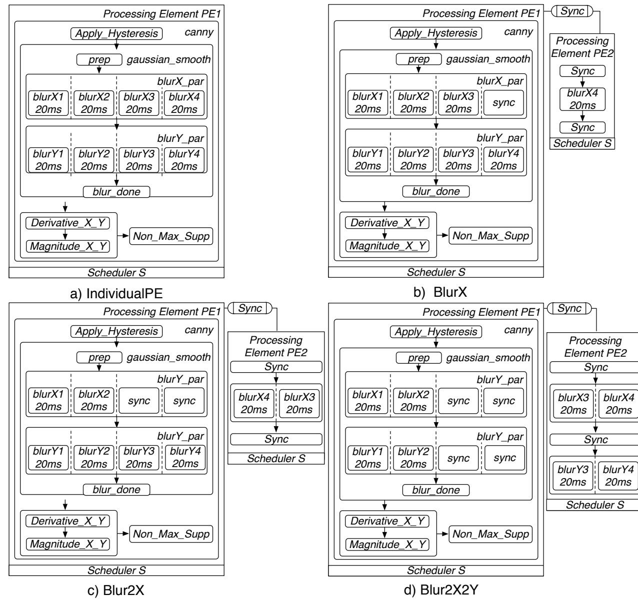

Figure 9: Partitioning of the Canny edge detector

model and measure the simulated time, count the scheduler calls and measure the execution time of the SystemC program. The count of scheduler calls is compared to the expected number of scheduler calls depending on the scheduling strategy. Based on the results, we can show that the communication via shared variables and double handshake channels, as well as the individual scheduler on the PEs, do not violate the causal execution order defined in the specification model.

The design we use for evaluation is a *Canny edge detector* [1], a graphical filter to detect edges in a *gray-scale* image. As starting point, we used an existing *SpecC* PSM model of the filter, transformed it into an *OSSS-Behaviour* [5] model, and implemented different virtual processing models, as shown in Fig. 9. Communication is performed via *shared variables* and *double handshake* channels. An array containing the complete image is shared among the *blur* behaviors. Each of these behaviors manipulates pixels on non-overlapping tiles of the image. In the Canny algorithm blurring is the most computationally intensive block and therefore we map different combinations of *blur* leaf behaviors to a second processing element. For our evaluation, we only require timing annotations for parallel and mapped leaf behaviors. We annotated each *blur* behavior with 20ms.

	Simulated time [ms]	Speedup	LoC	Δ [LoC]	Δ [%]
Specification	40	-	1497	-	-
IndividualPE	160	1	1541	44	2.9
BlurX	140	1.14	1760	219	14.2
Blur2X	120	1.33	1840	80	4.5
Blur2X2Y	80	2.00	1897	57	3.1

Table 1: Comparison of selected design metrics

The following models, as shown in Fig. 9, are evaluated: **Specification** (untimed, without virtual PE), **Individual PE** (timed *blur* leaf behaviors, all behaviors mapped on a single PE), **BlurX** (timed *blur* leaf behaviors, `blurX4` mapped to $PE2$, synchronization between $PE1$ and $PE2$ via double handshake channel (behavior `blurX4` on $PE2$ can only start if behavior `blurX_par` on $PE1$ has been entered)), **Blur2X** (timed *blur* leaf behaviors, parallel composition of `blurX3` and `blurX4` mapped to $PE2$, synchronization of parallel composition like in BlurX) and **Blur2X2Y** (timed *blur* leaf behaviors, parallel composition of `blurX3` and `blurX4` and parallel composition of `blurY3` and `blurY4` mapped to $PE2$, synchronization like in Blur2X with additional synchronization barrier between sequential composition of parallel *blur* behaviors).

When neglecting communication (shared array access), synchronization and scheduling (including context switching) times, our model's total simulated times, as expected by Amdahl's law, are shown in Tab. 1. We compare the complexity of the different models using a simple Lines of Code (LoC) metric. The major effort was to allocate new channels and behaviors for synchronization. For instance, for the model $BlurX$ a new PE and three *sync* behaviors have been instantiated. Furthermore, the double handshake channel was hierarchically bound from the ports of the PE's to the ports of the *blur* leaf behaviors.

In the following, we discuss the simulation of the four virtual processing models using a *round robin* scheduler with different time slice granularities from 1ns up to $100,000$ns on each individual PE. We measured the number of *context switches* and associated them with a constant cost of 1 ms. Fig. 10 shows the simulated time of model Blur2X2Y with *context switching* costs for a round robin scheduling of $PE1$ and $PE2$. As expected, we can observe that fine grained time slices < 100 ns have a huge impact on the overall simulated time. On the other hand, the responsiveness (although not necessary for the image filter design) rises.

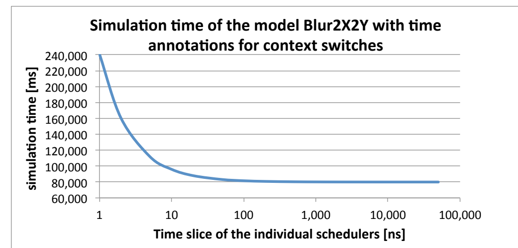

Figure 10: Simulation with costs for context switches

Fig. 11 visualizes the ratio between the simulated time for *context switches* and *computation* for a 20 ms leaf behavior timing annotation. When the time slice is very short, almost 70 % of the *simulated time* is spent on *context switches*. Fig. 12 shows the measured execution time of the various models on an Intel(R) Core(TM)2 Quad CPU Q9650 @ 3.00 GHz with 4 GB RAM running Fedora 12 Linux using the

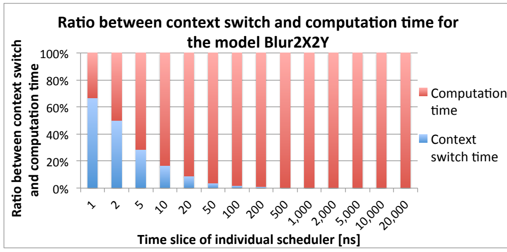

Figure 11: Ratio between context switches and computation time

time command. From this measurement, we can observe that the execution time of the individual models is proportional to the number of context switches, as shown in Fig. 10. We traced the individual scheduler calls and compared the

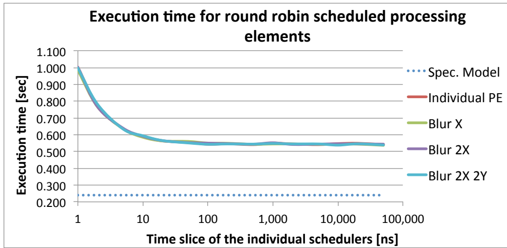

Figure 12: Simulation execution times of canny edge detector partitionings

execution order of leaf behaviors (i.e. the causal chain) between the specification and the Virtual Processing Models. In the specification model, the *blur* behavior's execution order was $blurX1$, ... , $blurX4$, while in the different VPM models the execution order changed to $blurX4$, ... , $blurX1$. Even though the ordering is different, validity of causality for parallel compositions (partial order) only requires to be order isomorph, which is the case.

6. CONCLUSION AND OUTLOOK

In this paper, we extended the proposed methodology in [4] and introduced a novel *virtual processing model* for PSM based models. The existing methodology allowed scheduling of processes using generic RTOS primitives. The design step from a non-scheduled *specification model* to a process-based RTOS scheduled *architecture model* is a major refinement step. For this reason, we proposed to introduce an intermediate model, called *virtual processing model*. This model enables to add a scheduler with user defined scheduling algorithm to a behavior, called *virtual processing unit*. This flexible scheduling annotation enables fast and easy exploration, regarding scheduling granularities of behaviors and assignments of scheduling policies. After successful exploration, the behavior to process transformation and RTOS configuration for architecture refinement can be performed. We have sketched how to use SystemC to implement our *virtual processing model*. Furthermore, we have integrated the virtual processing model into the *OSSS-Behaviour* library, supporting program state machine (PSM) modeling in SystemC. Our *implementation concept* allows designers to implement new scheduling strategies. For the *evaluation*, we used a *Canny filter* design and created different behavior partitions and scheduler configurations. The evaluation showed that our extension retains the functional causalities of the original PSM model when using a round robin scheduling with different time slice granularities. So far we did not evaluate our simulation results against measurements on a real platform. To do a trade-off between simulation speed and accuracy, a comparison with measurement results is necessary and part of future work. Currently, the context switch penalty and communication delay is handled in a very simple way and after the conduction of measurement trails these timing models will be refined.

Acknowledgement

This work has been partially supported by the *ARAMiS* project (01IS11035M) and the *EMC2* collaborative ARTEMIS project (01IS14002R), both funded by the German Federal Ministry of Research and Education (BMBF).

References

[1] J. Canny. A Computational Approach to Edge Detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, nov. 1986.

[2] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer, 1st edition, 2009.

[3] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Springer, 1 edition, 2000.

[4] A. Gerstlauer, H. Yu, and D. D. Gajski. RTOS Modeling for System Level Design. In *Proceedings of DATE*. IEEE Computer Society, 2003.

[5] K. Grüttner and W. Nebel. Modelling Program–State Machines in SystemC. In *Forum on Specification and Design Languages 2008*, 09 2008.

[6] P. Hartmann, H. Kleen, P. Reinkemeier, and W. Nebel. Efficient modelling and simulation of embedded software multi-tasking using SystemC and OSSS. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 19–24, Sept 2008.

[7] P. A. Hartmann, K. Grüttner, A. Rettberg, and I. Podolski. Distributed Resource-Aware Scheduling for Multi-Core Architectures with SystemC. In *Distributed, Parallel and Biologically Inspired Systems*, volume 329, pages 181–192. Springer, 2010.

[8] Z. He, A. Mok, and C. Peng. Timed RTOS modeling for Embedded System Design. In *Real Time and Embedded Technology and Applications Symposium (RTAS'05)*, 2005.

[9] S. Huss and S. Klaus. 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)*, 2007.

[10] T. Kempf, M. Dörper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-Processor SoC Platforms. In *Proceedings of DATE*, 2005.

[11] R. Le Moigne, O. Pasquier, and J. Calvez. A Generic RTOS Model for Real-Time Systems Simulation with SystemC. In *Proceedings of DATE*, 2004.

[12] S. Mahadevan, M. Storgaard, J. Madsen, and K. Virk. Arts: A System-Level Framework for Modeling MPSoC Components and Analysis of their Causality. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005.

[13] P. Razaghi and A. Gerstlauer. Host-Compiled Multicore RTOS Simulator for Embedded Real-Time Software Development. In *Proceedings of DATE*. IEEE, 2011.

[14] G. Schirner and R. Dömer. Introducing preemptive scheduling in abstract RTOS models using result oriented modeling. In *Proceedings of DATE*, New York, NY, USA, 2008. ACM.

[15] M. Streubühr, J. Falk, C. Haubelt, J. Teich, R. Dorsch, and T. Schlipf. Task Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures. In *Proceedings of DATE*, 2006.

[16] H. Zabel and W. Müller. An Efficient Time Annotation Technique in Abstract RTOS Simulations for Multiprocessor Task Migration. In *Distributed, Parallel and Biologically Inspired Systems*, volume 271 of *IFIP Advances in Information and Communication Technology*. Springer, 2008.

[17] H. Zabel, W. Müller, and A. Gerstlauer. Accurate RTOS Modeling and Analysis with SystemC. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware-dependent Software*, pages 233–260. Springer Netherlands, 2009.