Model-Based Generation of a Fast and Accurate Virtual Execution Platform for Software-Intensive Real-Time Embedded Systems*

Jochen Zimmermann, Martin Küster, Oliver Bringmann
FZI Forschungszentrum Informatik, Germany
{zimmermann,kuester,bringmann}@fzi.de

Wolfgang Rosenstiel
Universität Tübingen, Germany
rosenstiel@informatik.uni-tuebingen.de

Abstract— The shift towards embedded functionality increasingly realized in software and the permanently growing complexity in design and verification require new methodologies in the development process of software-intensive real-time embedded systems. Major issues related to the software and hardware architecture have to be found out as early as possible to reduce subsequent costs and to allow a short time-to-market. Therefore, system analysis and verification must be possible in every stage during the design process. In this paper, we present an approach to generate a virtual execution platform in SystemC which allows to execute embedded software with strict consideration of the underlying hardware platform configuration. Starting from abstract UML/SysML models of software and hardware architecture or/and abstraction of legacy code, model transformation techniques are used during the generation process. In combination with source code timing annotations obtained from binary code analysis this approach allows a fast and accurate simulation of the embedded system model. To substantiate our allegation we present experimental results from different application domains.

I. Introduction

Across application domains, the ratio of embedded systems' functionality realized in software has significantly increased compared to pure hardware implementations. This process is due to the strong request for flexible solutions with short time-to-market and the evolution in the semiconductor industry providing fast general-purpose hardware platforms with low energy demands. Further, design complexity and the specialization efforts of most manufacturers demand for a development process which can handle the integration of software components implemented by different developing entities or re-use existing implementations.

Traditional design processes are strictly aligned to the V-Model where system analysis and verification activities start right after implementation and integration are completed. Therefore, major issues, which are often related to the system architecture and introduced early in the design process, are discovered late which usually causes high costs due to an entire system redesign. Especially during embedded software development the missing consideration of the target architecture in current design methodologies often lead to systems which are either highly oversized or don't fulfill hard real-time requirements. A solution is a more iterative and incremental approach to software development that is driven by early validation and verification activities. Therefore, it must be possible to integrate both already implemented and abstract specified software components into a single simulation framework to ensure combined validation and verification.

Besides guaranteeing the strict observance of deadlines in real-time applications, power consumption has emerged as one of the most important factors in embedded system design, especially if systems are part of ultra-portable devices or depend on long battery lifetimes in general. Even in the automotive industry, energy-efficiency is becoming a crucial factor as driver-assistance and entertainment systems are selling points more than ever before. As today's embedded functionality is growingly implemented in software due to flexibility and cost reduction reasons we included dynamic voltage and frequency scaling (DVFS) techniques in our execution platform, which are known to be one of the most efficient low-power techniques on software execution level. DVFS defines several different operating modes (power modes) to adapt system performance dynamically to the actual need and is usually controlled by a resource managing layer, mostly implemented by the operating system itself.

In modern architectures embedded processors, caches, memories, and communication networks, as well as software-related parameters (e.g. scheduling policy) have a great impact on the temporal behavior of the applications, and the power dissipated during their execution on the target platform respectively. In contrast to just dealing with computational resources, regarding a whole platform in an accurate way is generally too complicated for pure analytical approaches. Due to the huge amount of parameters they usually suffer from state space explosion which prevents early system validation and verification. Therefore, we will focus on a simulation-oriented framework in the remainder of this paper.

Section II. deals with work already done in the field of simulation and analysis frameworks for embedded software and non-functional properties like execution time and power consumption. Section III. gives a brief overview of the methodology proposed in this paper. In Section V. we depict our system models. In Section VI. we describe our tool flow and the model transformations. Section VII. shows how the embedded software is actually "executed" on the virtual execution platform. Finally, we give some experimental results in Section VIII. and conclude the paper.

II. Related Work

Analytical approaches are usually distinguished between white-box and black-box approaches. In white-box approaches [2] [6] the model is derived from functional system implementation which includes complex interactions schemes and synchronization primitives. Whereas in black-box [19] approaches, the underlying model is explicitly derived from system specifications which enables efficient performance analysis but prevents the incorporation of synchronization and interaction. These approaches all have in common that they cannot include dynamic aspects in hardware and software (e.g. data-dependent loops, multiple hardware components), and, more importantly, are not applicable to complex system models due to analysis state space explosion.

To overcome these limitations, there exist analysis approaches based on simulation. Obviously, simulation-based approaches lack of provability and sufficient corner case coverage. Nevertheless, they allow an early system validation and verification with respect to certain stimuli even for complex system models. Furthermore, systems can be simulated at different levels of abstraction, e.g. if some parts of the system are already implemented (bottom-up) whereas some parts are just specified by certain properties (top-down).

To allow the evaluation of an entire system, including processing elements as well as the system environment, some approaches have been introduced which discuss the integration of instruction set simulators (ISS) into a SystemC simulation framework, e.g. [7]. Usually, the simulation performance of ISS approaches is very

*This work was partially supported by the ITEA project VERDE under BMBF grant 01IS09012A.

low which prevents system evaluation of a complex system and especially comprehensive system exploration. In [8], the authors present the integration of an real-time operating system (RTOS) model into SpecC. The authors in [15] integrate an RTOS model into SystemC. These works in simulation frameworks only deal with evaluating the performance of the simulated models, e.g. target execution time, which means that they lack of considering also the power consumption of the system.

For estimating the power consumption there exist several approaches which can be separated depending on the granularity and abstraction level. Most of the previous work for power estimation has been done in low-level power simulations at gate-level, register-transfer-level, or architectural-level [21]. Due to extremely long simulation times this approach is not applicable to complex software-intensive systems. More abstract approaches are usually based on power models at instruction level [20] which give an average power dissipation of each instruction executed on a certain processor. Most of those models capture also sequences of instructions but no data-dependent power dissipation. More abstract power models rely on power state machines (PSM) [3] [9]. A state in the PSM is related to an average power dissipation of a processor whose operating mode corresponds to this power state. The overall energy consumption is calculated regarding the power dissipation over time. A similar approach is shown in [5], where system-level activity information exchanged over an on-chip bus is used for triggering the PSM. Park et.al. [14] present a methodology for power models at different levels of granularity depending on the required accuracy.

Commercial tools for virtual prototyping, e.g. [18], usually need a detailed view on the system which prevents early system verification. Mostly, ISS approaches are used for the simulation of computational tasks mapped on processing elements and early system power analysis is not supported at all. Furthermore, they don't provide abstract and timing accurate operating system funtionalities like scheduling tasks on multiple processing elements.

III. Methodology

A. Motivating Example for Simulation-based Approach

To compare pure analytical to simulation-based approaches we will give a short motivating example. The simulated multi-core platform model (2 ARM7TDMI cores, 50 MHz, shared 1kB 2-way set associative instruction cache with 16 bytes line size) is executing a controlflow-dominated circle detection algorithm in parallel to another applications. A cache miss inflicted a penalty of 5 clock cycles compared to a cache hit.

TABLE I
Circle Detection Algorithm on SystemC Execution Platform

	Exec. Cycles	Hit/Miss Ratio
Worst-case Exec. Time	12,124,151,203	–
Simulation	7,317,925,058	–
Simulation incl. I-Cache	2,368,802,550	16.57

Table I shows a 39% reduction of target execution time calculated by a WCET analysis tool [1] (without regarding any caching policy) compared to the target execution time gained by taking the timing paths in a simulation of the algorithm on an abstract platform model, stimulated by a testbench. Including an abstract cache model further reduces the execution time significantly to almost 2.4 billion cycles due to cache hits. Simulation time was 111.5 seconds on a 2.66 GHz machine. This motivating example shows that including complex dynamic behavior is essential for accurate performance and power estimations (especially for normal use cases), but this is mostly disregarded in analytical approaches.

B. SystemC Language

In recent years, SystemC [13] [10] has emerged to a de-facto standard in EDA industry and research for modeling and simulation of embedded systems, especially at high abstraction levels. In general, SystemC is a static library for discrete, event-driven simulation of a real-time environment and bridges the gap between hardware and software providing the means for component-based modeling and simulation by separating computation and communication concerns. The SystemC IEEE standard [11] defines both the language concepts as a C++ library and the simulation kernel for concurrent hardware simulation. In SystemC, a design is partitioned and encapsulated into modules. Each module can contain other modules (and act as a hierarchical element), processes (threads) that describe the functionality, and ports through which a module communicates with other modules. A process can be suspended by calling a wait statement with a certain wait condition and is simulated concurrently to other processes by the SystemC simulation kernel. Interfaces contain a set of operations which are accessed by a port and implemented within a channel. These communication operations could be primitive, e.g. signals, or realize a more complex specific communication protocol.

C. Abstraction Level for Simulation

Dealing with simulations always means dealing with the trade-off between accuracy and simulation speed. In SystemC, there exist several abstractions levels, ranging from cycle-accurate hardware modeling (CA) at register-transfer level to programmers-view level (PV), where modules are interacting through point-to-point connections, allowing timing annotations (PVT). In general, simulation speed is defined by synchronization calls to the simulation kernel (`wait()`-call either on a specified time or on an event), which causes a context switch inside the kernel. For our simulation framework we choose a loosely-timed modeling style, which means that different processes might have their own timing (temporally decoupled simulation) but are synchronized if necessary, e.g. access to shared resources, thread switch in scheduling policy, reaching a synchronization point due to interaction with another process. This allows a fast simulation due to calling `wait()` only if necessary, but also keeps the temporal behavior sufficiently accurate.

IV. Layered Approach for Integration

In this section we will present our layered approach to encapsulate software components for an integration into the simulation framework.

A. Embedded Software Development

In classical embedded software development there should be no need to care about the deployment of that software, meaning in which context the software is actually executed on the target platform. Therefore, software in object-oriented programming is implemented based either on a direct access between caller and callee of a software function, or indirectly through well-defined interfaces. This is illustrated in Figure 1 using standard UML syntax.

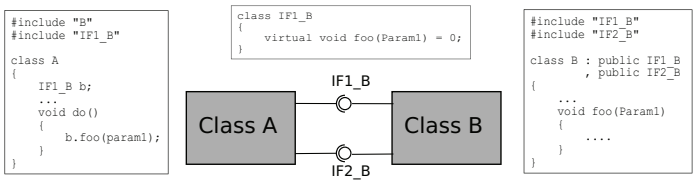

Fig. 1. Structure in Classical Software Architectures

Class B provides a function `foo()` through an interface `IF1_B` (we will refer to this as a "software interface" in the following). *Class A* calls this method by instantiating an object of class *B*. This interaction requires at least some visibility of *B*. However, the SystemC simulation kernel as a component-based framework requires software components to communicate solely

through ports. As a result, additional layers for software integration need to be generated.

B. Modular Software Components in SystemC

The procedure for generating the simulation framework integration layers is subdivided into the direction of the software component interaction. Basically, there are two cases: functions which are provided to the outside, and functions which require to call another function. Principally, Figure 2 shows the same interaction pattern as Figure 1, but now each class is encapsulated in a SystemC module (*module_A* and *module_B*), delegating their interface semantics through corresponding ports: *sc_port* for required interfaces and *sc_export* for provided interfaces.

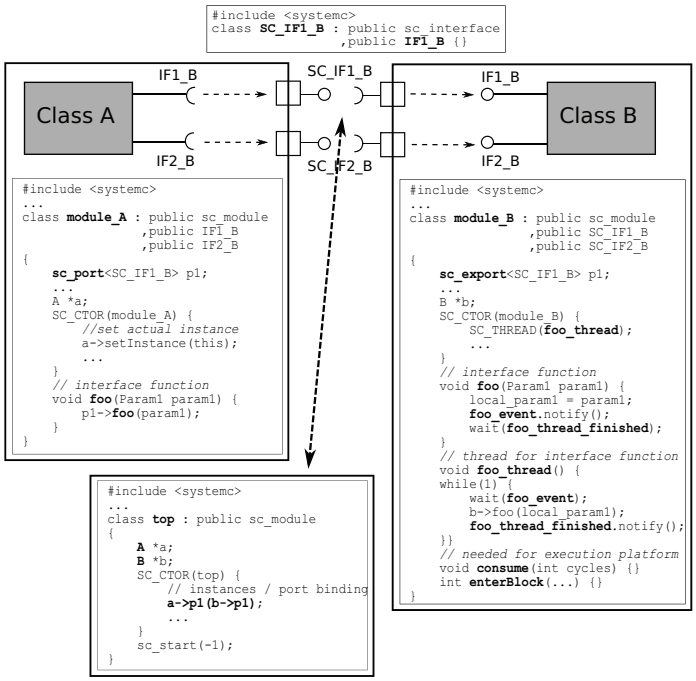

Fig. 2. Layered Approach for Software Component Integration

To define the type of these module ports, SystemC interfaces (*SC_IF1_B*, *SC_IF2_B* is not shown due to space limitations) are generated inheriting from the usual software interfaces. For each provided interface function, a thread (*foo_thread*) is generated which calls the appropriate function inside the encapsulated class (*B*). Events are used for activating the SystemC thread (*foo_event*) and notifying the interface function (*foo_thread_finished*) after encapsulated function execution has finished. Function parameters have to be saved in local module variables before thread activation. For required interface functions, the call can just be delegated to the appropriate port (*foo()*) because the containing module is implementing the software interfaces.

The generated modules are connected by means of the standard SystemC port binding (module *top* in Figure 2).

V. System Models

To allow a fast evaluation of the functional and temporal behavior of software components on the actual target architecture, the application source code can be annotated with low-level properties obtained from the target binary code. By compiling the application source code for the target architecture this software model is enriched with target architecture properties (e.g. target instruction set, pipeline stages, branch prediction) while still executing on the simulation host which enables a simulation speed close to native software execution.

A. Timing Model and Back-Annotation into Source Code

First work for the used timing model was presented in [16] and extensively improved in [17] to include complex code optimizations of modern compilers like loop unrolling, function inlining, and loop invariant code motion. For each basic block of the cross-compiled binary program a timing analysis which models *pipeline* effects and static *branch prediction* penalties of the target processor is performed. This results in a timing-annotated control flow graph (CFG) of the binary executable with labeled edges representing the execution time (or cycles) required for instruction execution (see Figure 4). Based on the binary-level CFG, the program control flow on the target architecture is analyzed to create path simulation code which models the target-specific behavior of the program. Compiling the instrumented source code and the path simulation code for the simulation host yields a model of the program which determines its execution time on the target processor. Using the markers that were added to the original source code during instrumentation, the path simulation code can approximate the path taken through the binary executable. This reconstruction of binary-level control flow is executed in parallel to the functionality of the original source code during simulation on the simulation host and allows the dynamic selection of timing annotations. This makes the timing estimation very accurate, but several factors faster compared to an instruction set simulator (ISS). Experimental results are shown in Section VIII.. Note that this timing model is used for single program/task execution.

To change the timing model, e.g. using another target processing resource, only the annotated basic block execution time has to be adapted whereas the binary control flow remains the same. Models for caches, pipelines, and branch prediction are (virtually) accessed and evaluated during simulation which makes them easily configurable (cf. Section VI. C.).

B. Power Model

There exist a lot of power models based on instructions executed on the target architecture (see Section II.). However, these low-level models are not suitable for fast simulation and disregard low-power techniques implemented in the target platform hardware, e.g. operand isolation. Also, analyzing power consumption on instruction level is a tedious and error-prone task because platform manufacturers usually provide average power values for each operation mode. Thus, we decided to use a state-based power model for fast evaluation of power dissipation during embedded software execution. The states of the corresponding power state machine (PSM) contain the power dissipation of the modeled entity (e.g. microprocessor) running in this state. Also, an appropriate execution speed (frequency) is mapped to each power state denoted in cycles. State transitions switch between power states including a hardware-dependent switching overhead in time and power dissipation. Comparable to the timing model, this power configuration can be easily changed before the simulation since it is generated according to the platform specification.

VI. Model-based Transformations

A. Model Flow

The approach fits into an extensive chain of model transformations (depicted in Figure 3). There are two ways to come to a SystemC model that may serve as an input for the execution on the virtual platform. First, existing SystemC source code may be parsed and transformed into an AST representation. Afterwards, a relational transformation serves to extract the necessary information which is then condensed in the SystemC model. All steps may be reversed, establishing a way from any SystemC model to source code.

Second, a top-down approach based entirely on the UML can be taken, too. An initial software design and hardware platform (modeled in SysML) is the starting point for the transformation. As detailed in Section IV., the different artifacts are combined

resulting in a SystemC model. The two different ways may complement each other: for an initial draft of the system architecture, the top-down flow may be taken. Based on detailed knowledge about the implementation an additional timing model (as outlined in Section V. A.) can be included. The wrapped functional implementation containing the respective delays is carried over to the SystemC model leading to a more accurate simulation. Obviously, the resulting SystemC virtual prototype can easily be modified by changing the UML model of both hardware and software.

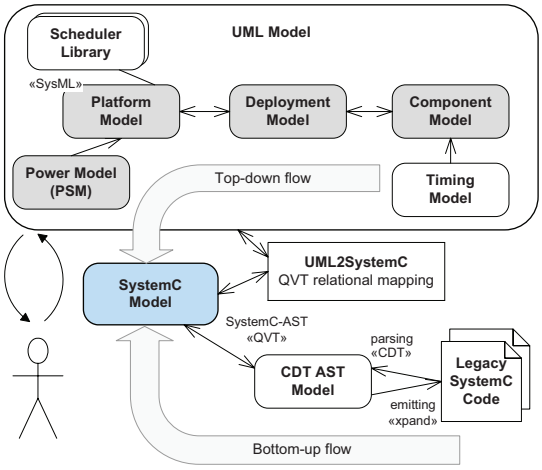

Fig. 3. From UML models to SystemC and back

Even without a timing model extracted from an existing implementation, a designer may attach coarse-grained timing annotations. This is particularly suitable for early design phases when the implementation is not yet available. These rough values are carried over to the simulation leading to an estimation of the actual execution times.

Note that the central advantage of the overall model-based approach is that no artifact created in the design phases gets outdated: bi-directionality of the involved transformations (due to [12]) guarantees consistency despite changes to either of the inter-related models.

B. Abstraction of Legacy Code

Intrusive approaches for legacy code abstraction (see Section II.) depend on a completely implemented systems which can be elaborated by the SystemC simulation kernel. As we devised a different approach to tackle the problem: pattern-based extraction based on static analysis. From the Eclipse CDT tooling we get the abstract syntax tree (AST) of the preprocessed and hence fully expanded source code. A model-to-model transformation is responsible for translating AST fragments to elements of the SystemC model. If a complete and functional SystemC implementation is at hand, techniques such as the one presented in [4] could as well build the abstract SystemC model. Such an adapter has not been implemented, however.

C. Virtual Platform Generation

The hardware platform is specified and configured in SysML and equipped with additional information from a set of library schedulers and links to a power model (see Section V. B., and also Figure 4). The combined information serves as the input to a model-to-model transformation (UML2SystemC in Figure 3). We have shown a template-based generation process of structural simulation code in a previous publication [22] (see emitting process in Figure 3). Changing the hardware platform has a direct impact on the models for timing and power estimation (cf. Section V.).

For a dynamic consideration of the hardware platform we are using a model-based approach to generate models for software scheduling and state-based dynamic power management.

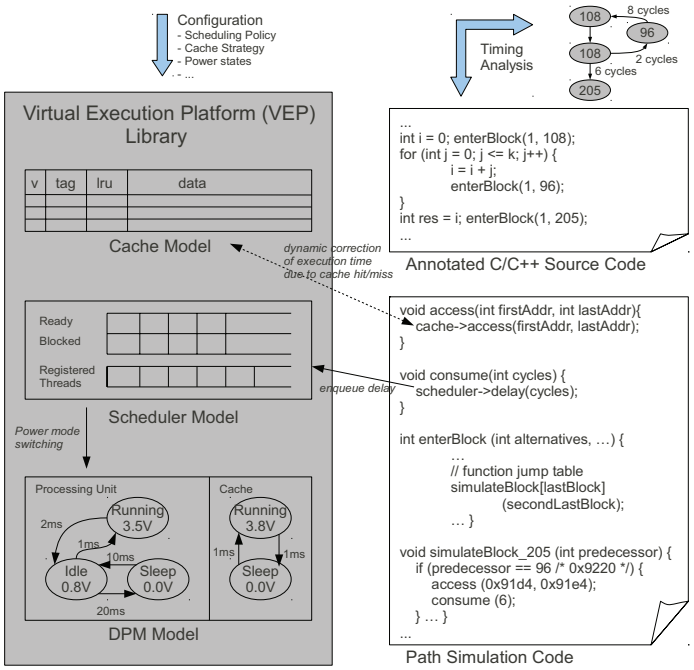

Fig. 4. Architecture of Virtual Execution Platform (VEP)

Based on the selected model scheduling events are triggered, e.g. priority-based or time-division scheduling. Software components deployed on different processing units are synchronized when they access shared resources or when a thread switch occurs. Power mode switches are triggered as event calls to the corresponding PSM model, which realizes the switching delay. For simulation speed-up, delay portions that need not be synchronized can be condensed to prevent avoidable context switches inside the simulation kernel. Also, a cache model is generated giving accurate cache hit or miss numbers.

VII. Virtual Execution Platform

In this section we will describe how the embedded software is executed on the virtual execution platform (VEP). The solution for creating source code with timing annotations representing the execution time of a single program was already described in Section V. A.. As we target a SystemC-based simulation the equivalent concept to indicate consumed execution time is a call to the SystemC `wait()` function.

A. Platform Scheduling

We developed a generic approach to schedule the access to shared resources, e.g. computational time on microprocessor cores or transfer time on buses, in a SystemC-based virtual execution platform library (see Figure 4). The behavior of the scheduler model is also illustrated in Figure 5, scheduling threads to a computational unit serves as an example.

In general, there exists a scheduler component in each simulation model which manages one or multiple dispatcher depending on the number of shared resources. In a multicore platform with 2 microprocessor cores there would exist 2 dispatchers, one for each core. Each thread has to register with the scheduler which stores a thread information for internal thread management and resource conflict decomposition.

Each thread executes its annotated source code, and also the path simulation, and enqueues the corresponding delay of each basic block in its scheduler queue. If a synchronization point is reached, e.g. thread blocks due to a communication point or the scheduler initiates a thread switch, the scheduler checks if the thread, which was supposed to be active, can really consume all the delays until the thread switch happens. Note that

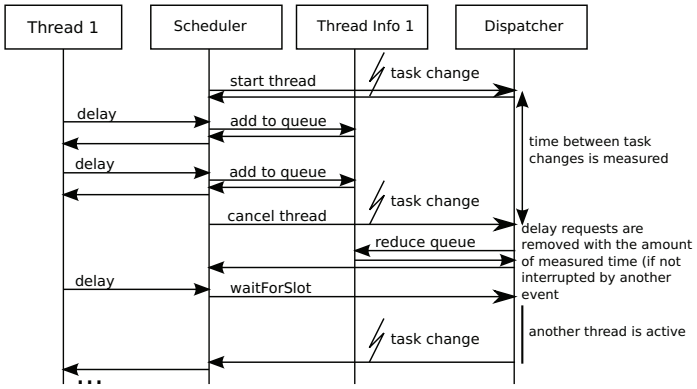

Fig. 5. Thread Scheduling due to Scheduling Policy

the SystemC simulation time has not advanced up to this point which means that the thread is temporal decoupled since there is no need to synchronize (maximum quantum going ahead can be specified).

If threads have to be synchronized the scheduler must call a SystemC `wait()`. This `wait()`-call is realized by a disjunction of all events which might happen during the synchronization period, e.g. activation event of another thread by an interrupt. SystemC semantics ensure that synchronization is done at the first synchronization point in SystemC simulation time, for example the aforementioned interrupt. So, if this interrupt happens before (in simulation time) the thread switch, the scheduler dequeues the delays of the active thread up to that point, the rest stays in the queue and the thread is notified that it was interrupted before it has actually finished the time slice it has expected to have (see Figure 6). The remaining delays can be consumed when the thread is activated again by the scheduler. In a metaphorical sense, the thread local time is corrected by the scheduler having a global view on all events that might happen in the simulation.

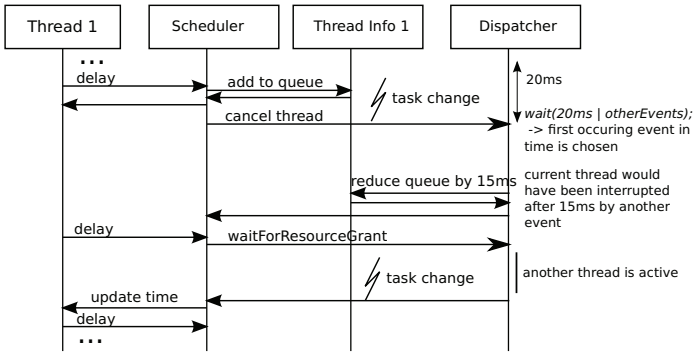

Fig. 6. Interrupted Thread Scheduling

For simulation speed-up, this delay queue is implemented in a data structure allowing to perform a stepwise search for the accumulated time which has to be consumed. First, a coarse-grain search for the appropriate queue is performed based on fixpoint evaluation and binary search is used inside the queue, reducing the search effort to O(log n) being n the queue length.

B. External Events

In general, the scheduler model handles blocking periods due to blocking calls, e.g. blocking read, comparable to thread switches by executing a synchronous delay. Before yielding the scheduler, the thread additionally calls the `sleep()` function to indicate that it is waiting for an external event (which might be the requested data or an interrupt) and calls a SystemC `wait()` on this event (see Figure 7). This leads to a status switch to *blocked* inside the scheduler model which prevents the thread from being scheduled again until it is unblocked. In the case that this external event occurs, SystemC simulation continues after the `wait()`-call. Therefore, the scheduler model uses the notification capabilities in SystemC by setting the thread status to *ready* if the thread hits the next `delay()`-call in its path simulation code (cf. Figure 4).

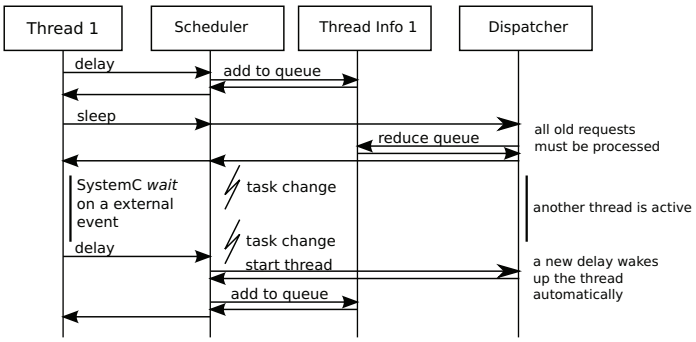

Fig. 7. Thread Blocking and Notification by External Event

C. Platform Power Management

For triggering the stated-based power model the function `triggerPowerModeSwitch()` is called which expects the desired power state as a parameter. This call can be both made manually and initiated from a power manager e.g. triggered by the scheduler in idle phases. Within this call, switching overheads apply due to the PSM model, and the execution on the appropriate power-managed entity is stalled during mode switching. As target execution time is calculated by the number of cycles and the maximum frequency is defined by the actual power state, the target execution time (not simulation time!) is also adapted to that power state.

VIII. Experimental Results

Figure 8 exemplifies the priority-based scheduling result of two software components sharing a single computation resource. Resource requests are indicated by rectangle, wherein blocking times are solid black. Assuming component B has a higher priority, component A is scheduled only if B is blocked due to a blocking call or has eventually finished. If the scheduler discovers idle phases of the computational resource, it triggers a switch to a low-power mode. During switching times, which are indicated with black flanks in the upper area of Figure 8, software execution is stalled.

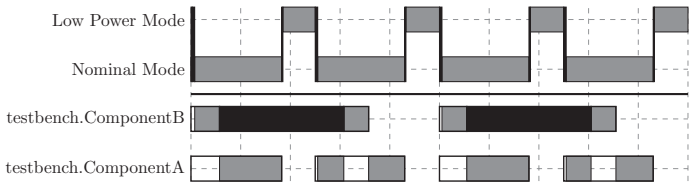

Fig. 8. Priority-based Scheduling including different Power Modes (white: inactive, grey: active, black: blocked)

We can also determine the energy consumption calculating the execution time in both power states over time and adding the switching overhead between consecutive power states for components A and B. On a ARM-based Cortex A8 virtual platform this results in about 27mJ energy consumption for this example.

For more complex experiments we implemented a circle detection algorithm (cda) as part of an automotive traffic sign recognition. A camera image is processes by a Sobel filter algorithm for edge detection whose center point is transformed into the Hough space, where reaching a certain threshold decides about being a circle or not. Detected circles are classified by a support vector machine.

TABLE II
Comparison of Simulation Speed between ISS and VEP

	Sim. Cycles	Sim. Time (s)		Speed-up
		ISS	VEP	
edn (50000x)	2,699,160,944	2,700	43	63x
matmul (50000x)	5,939,931,199	500	56	9x
cda (4x)	40,006,112,152	1,800	151	12x

Table II shows the simulation time of annotated source code of a WCET benchmark suite and the cicrle detection algorithm (cda) on our Virtual Execution Platform (VEP) compared to the execution on an instruction set simulator (ISS). To get a robust number we executed *edn* and *matmul* 50000 times, the detection algorithm 4 times with different data input. The latter needs 151 seconds to simulate over 40 billion cycles on the target architecture. During the simulation, the virtual execution platform processes more than 715 million basic blocks.

Analysis of the accuracy of timing results show an average error less than 5% compared to the execution on an ISS. However, simulation speed is faster multiple times. The actual simulation speedup depends on the control flow characteristics on the algorithm – more basic blocks mean less speedup.

TABLE III
Scheduling Performance on Virtual Execution Platform

	Sim. Time VEP	# switches	average error
600ms slot	134s	698	596.48ns
30ms slot	152s	13,781	670.45ns

In Table III, we show two TDMA scheduler configurations. Note that simulation time is just increasing by 13,4%, although the number of thread switches has increased by a factor of almost 20. This shows that avoiding unnecessary context switches in the SystemC kernel (by calling `wait()` function) works very well inside the scheduler model. External events are not handled at instruction level, but at basic block level. So the interrupt will become effective at the end of the last simulated basic block. Therefore, thread preemption (caused by e.g. interrupts, external events) might happen slightly too early. The resulting average error in ns is shown in column four. However, the accumulated error is only about 0.005% compared to the overall execution time on the target platform.

In an additional experiment we focused on determining the power consumption on a single-core ARM-based Cortex A8 virtual platform with several operation/power modes by dynamic supply voltage and frequency scaling (DVFS) and guiding an optimization process (hard requirement: classification of 6 cycles at 720MHz). The execution platform was stimulated by multiple camera images, so different timing paths depend on the number of detected circles in the camera image. Available data sheets of the SoC map supply voltages (0.975-1.35V) onto appropriate maximum frequencies (125-720MHz) in each mode. The result showed that for example a frequency of 250MHz is enough to classify 2 circles, which results in 55% reduction in power dissipation by using lower power modes.

IX. Conclusion

In this paper we have shown our model-based virtual platform generation methodology to enable fast and accurate system validation and verification by simulation in SystemC. Therefore, we devised a model-based generation technique of a virtual execution platform taking into account the execution time on the target platform obtained from timing analysis on binary level, different scheduling policies for shared resources, and several power modes for platform components. To integrate existing C/C++ software components, we developed a layered approach wrapping interactions between software components into SystemC-based communication patterns. Giving experimental results we showed the general applicability in terms of simulation speed and degree of automation, and sufficient accuracy for early system verification.

References

[1] AbsInt. *aiT WCET Analyzer*.

[2] K. Albers, F. Bodmann, and F. Slomka. Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*. IEEE, 2006.

[3] R. A. Bergamaschi and Y. W. Jiang. State-based power analysis for systems-on-chip. In *DAC '03: Proceedings of the 40th annual Design Automation Conference*, pages 638–641. ACM, 2003.

[4] H. Broeders and R. van Leuken. Extracting behavior and dynamically-generated hierarchy from systemc models. In *DAC '11: Proceedings of the 48th Annual Design Automation Conference*. ACM, 2011.

[5] Y. Cho, Y. Kim, S. Park, and N. Chang. System-level power estimation using an on-chip bus performance monitoring unit. In *ICCAD '08: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. ACM, 2008.

[6] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya. A generalized static data flow clustering algorithm for mpsoc scheduling of multimedia applications. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008.

[7] F. Fummi, S. Martini, G. Perbellini, and M. Poncino. Native iss-systemc integration for the co-simulation of multi-processor soc. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, volume 1, pages 564 – 569. ACM, 2004.

[8] A. Gerstlauer, H. Yu, and D. Gajski. Rtos modeling for system level design. In *DATE '03: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 130 – 135. ACM, 2003.

[9] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*. ACM, 2008.

[10] T. Groetker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[11] IEEE Computer Society. *IEEE Standard SystemC Language Reference Manual*, 2006.

[12] OMG. *MOF2 QVT Specification*. Object Management Group, 2011.

[13] Open SystemC Initiative (OSCI). SystemC. `http://www.systemc.org`.

[14] Y.-H. Park, S. Pasricha, F. J. Kurdahi, and N. Dutt. Methodology for multi-granularity embedded processor power model generation for an esl design flow. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2008.

[15] H. Posadas, J. Adamez, E. Villar, F. Blasco, and F. Escuder. Rtos modeling in systemc for real-time embedded sw simulation: A posix model. *Design Automation for Embedded Systems*, 10:209–227, 2005.

[16] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-performance timing simulation of embedded software. In *Proceedings of 45th annual Design Automation Conference*. ACM, 2009.

[17] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate source-level estimation of software timing considering complex code optimizations. In *DAC '11: Proceedings of the 48th annual Design Automation Conference*. ACM, 2011.

[18] Synopsys. Virtualizer. `http://www.synopsys.com/systems/virtualprototyping/pages/virtualizer.aspx`.

[19] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 4, 2000.

[20] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. 2(4), 1994.

[21] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of simplepower: A cycle-accurate energy estimation tool. In *Proceedings of the 37th Design Automation Conference*. ACM, 2000.

[22] J. Zimmermann, M. Pressler, A. Viehl, O. Bringmann, and W. Rosenstiel. Model-based virtual prototyping for early automotive software systems evaluation. In *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design at DATE*, 2010.