Combination of Instruction Set Simulation and Abstract RTOS Model Execution for Fast and Accurate Target Software Evaluation

Matthias Krause[1], Dominik Englert[1], Oliver Bringmann[1] and Wolfgang Rosenstiel[1,2]

[1] FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany

[2] Universität Tübingen
Sand 13
72076 Tübingen, Germany

{mkrause, englert, bringmann, rosenstiel}@fzi.de

ABSTRACT

Instruction set simulation and real time operating system modeling have become important issues for the design of distributed embedded systems. This paper presents a holistic approach to simulate a distributed, embedded system that includes target software, processing units, and abstract RTOS within a virtual prototype environment. The processing unit is modeled by an ISS, which is embedded in a SystemC environment to allow the integration into a platform model. In comparison to existing approaches, the RTOS is not directly running on the ISS but outsourced and replaced by an RTOS model. This step strongly reduces simulation time since the execution on the ISS is much more time consuming in contrast to the execution on the host processor. The results show the theoretical and measured performance gain depending on the RTOS scheduler and task switching.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems
I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Design, Performance

Keywords

Embedded Systems, RTOS Modeling, Instruction Set Simulation

1. INTRODUCTION

In recent years, the increasing complexity of electronic systems has become a complex challenge for electronic systems designers. This is particularly true for the development of embedded software. Since the rate of embedded software within an embedded system is steadily increasing, the role of real time operating systems (RTOS) becomes more and more important. Additionally, the role of interconnection is also increasing, and application is distributed to several processing elements (e.g. driver assistant applications in auto-

motive electronic systems). In this context, virtual prototypes, including processing elements and RTOS, have become an established design methodology to simulate and evaluate the entire system at an early design phase. However, the development of virtual prototypes is also a challenging task regarding the complexity and heterogeneity of the entire system. Important attributes of virtual prototypes are: sufficient abstraction to allow early system integration, high simulation accuracy, and a high simulation performance.

1.1 Virtual Prototype Simulation for Software Related Distributed Embedded Systems

A lot of work already exists which discusses embedded software modeling and simulation with focus on RTOS models, instruction set simulators (ISS), the common usage of ISS and RTOS models, and the speedup of simulation platforms.

RTOS modeling is typically performed at high levels of abstraction to allow design space exploration of different RTOS strategies. To evaluate the influence of these strategies onto the entire system, abstract RTOS models have been developed for system level design languages like SystemC [1] and SpecC [2]. RTOS modeling for system level design is discussed for SpecC [17] and SystemC ([22], [20], [18]). RTOS refinement strategies based on RTOS modeling is also presented for SpecC [21] and SystemC [23]. The approach in [19] shows RTOS modeling in connection with multi-processor platforms. The authors of [24] present a proprietary approach for embedded system co-simulation using an RTOS model. A SystemC simulation model of a concrete ITRON based RTOS is presented in [25]. What all these approaches have in common is that interrupts and task switches can only be performed at manually specified preemption points.

Instruction set simulation is widely used for embedded system design. It allows executing embedded software applications on a simulation model, the ISS, of the target processor. A well-known open source ISS is the interpretive simulator Simplescalar [10] which is presented in [11]. An ISS that bases on just-in-time cache compiled simulation is, for example, introduced in [9]. To allow the evaluation of the entire system, including the processing element as well as the system environment, some approaches have been introduced which discuss the integration of ISS into a SystemC simulation framework ([12], [13] and [14]). Also, building multi-processor platforms has been the topic of research. The authors of [15] and [16] introduce a multi-processor approach using Simplescalar within a SystemC framework.

The authors of [27] present a proprietary co-simulation approach that combines an ISS with an RTOS model. This approach is based

on virtual synchronization that is introduced in [28]. They replace the RTOS by an RTOS model within the co-simulation backplane to reduce the co-simulation overhead. In further work, the approach has been improved to overcome some limitations by introducing trace-driven virtual synchronization in [29]. In [30], a framework has been presented which is based on the approach in [27] and which uses the new simulation technique. The authors of [26] discuss a combined RTOS model/ISS approach. However, thread switches are still executed on the ISS and consume several instructions on the ISS. Other RTOS functions are executed outside the ISS, but each instruction has to be decoded additionally within the program memory to determine RTOS-API calls. This additional decoding effort slows down the simulation performance.

Lately, speeding up the simulation by annotating timing behavior into high level system models has become an important research topic. In contrast to the ISS approaches, the hardware is not directly modeled, but is taken into account by annotating the corresponding timing behavior. However, the key is to determine this timing as accurately as possible. This task is difficult since not all of the timing behavior is predictable (e.g. dynamic timing aspects). Hybrid approaches have partially overcome some limitations. The authors of [5] present an approach for the instrumentation of timing behavior by using the application profiling tool that is introduced in [6]. A hybrid model that is for fast simulation by switching between native code execution and ISS based simulation is introduced in [7]. The authors of [4] present an approach which back-annotates statically analyzed timing behavior into a simulation model and which considers the dynamic timing behavior by introducing correction code during runtime. A similar approach, which determines the dynamic timing behavior by statistical models, is shown in [8].

2. MOTIVATION AND BASIC CONCEPT

This work combines the approaches of abstract real time operating system modeling with instruction set simulation. The idea is to remove the RTOS from the ISS and replace it by an abstract RTOS model outside of the ISS. In contrast to the existing combined approaches, this approach focuses on event-based simulation and performs a cycle-accurate thread switch from outside the ISS, although it uses an RTOS model instead of an RTOS. If a thread switch occurs, the context is stored and switched to the scheduled thread by pointer replacement. This consumes a minimum of time in contrast to thread switching which is executed directly on the ISS and which consumes several instructions. RTOS model and ISS wrapper are implemented in SystemC at TLM level. Hence, further system components can easily be integrated in contrast to proprietary approaches. This allows evaluating RTOS strategies within the context of the entire system. The approach has the following features:

1. The concept offers a platform for design space exploration concerning the RTOS, since it neither yet requires a decision for a particular RTOS nor the porting of the application to the RTOS. In contrast to RTOS model approaches at higher levels of abstraction, this approach consideres the cycle-accurate behavior of the application during evaluation. Furthermore, it does not need the manual specification of preemption points in the application code.

2. The approach facilitates the evaluation of multi-core RTOS and the evaluation of load balancing strategies respectively. It

allows the user to build multiple ISSs, which communicate with a single RTOS model.

3. Simulation performance is increased because no ISS instructions are consumed for thread switches, and RTOS functionality is executed outside the ISS. For interpretive ISS, instructions consume more simulation time on the ISS in contrast to the simulation host.

An important feature of the RTOS is the scheduling. Scheduling is needed if running a multitasking application on a single processing element. This paper will focus on the interaction of the scheduler and the ISS and the idea for thread switching of the application threads that are still running on the ISS. Figure 1 depicts the key concept of this paper.
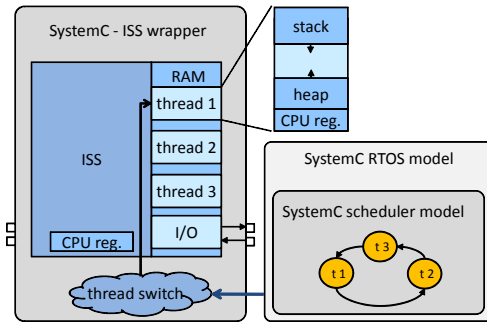

Figure 1. Combination of RTOS model and ISS.

In our approach, the scheduler is modeled in SystemC and then executed by the SystemC simulation kernel while the ISS kernel is executing the applications. Both simulation environments exchange data, such as thread switching information and synchronization. Each application thread has a separate memory and register range. If a thread switch occurs, the memory and register area is switched and the ISS kernel executes the new task.

3. COMBINATION CONCEPT

In this section, our approach is described in more detail. Section 3.1 introduces the ISS. Implementation and experimental results are based on the Simplescalar ISS ([10], [11]) and the ARM instruction set. Section 3.1 gives an overview to the few changes that have been made to integrate Simplescalar into the SystemC/ISS wrapper. Section 3.2 gives a brief description of the RTOS model structure. This section does not discuss RTOS modeling in detail, since our model is comparable to the models referenced in Section 1.1. Please refer to these papers to obtain further information about RTOS modeling. Section 3.3 discusses the SystemC/ISS Wrapper. It is responsible for the communication between the ISS and the RTOS model, including the context switching between two threads. However, the synchronization of the co-simulation is already the object of several papers referenced in Section 1.1 and therefore not in the scope of this paper.

3.1 Instruction Set Simulator

The ISS is directly integrated into a SystemC module. This is faster than co-simulation via IPC or GDB RDI interfacing (cp. [12]). Furthermore, multi-instantiation is possible for building multi-core/ multi-processor simulation models. Therefore, the porting of Simplescalar into C++ is necessary: memory, registers and caches are implemented by C++ classes. An interface for memory access decouples the memory from the simulator. The class implementation allows multiple instances of the memory. Each application

thread is assigned to a separate memory object. In the same way, each application thread has a separate register object. The processor kernel is implemented within a class and contains all kernel-specific data structures and functions. All classes and Simplescalar components are integrated into a common namespace. The new structure of the Simplescalar is depicted in Figure 2.
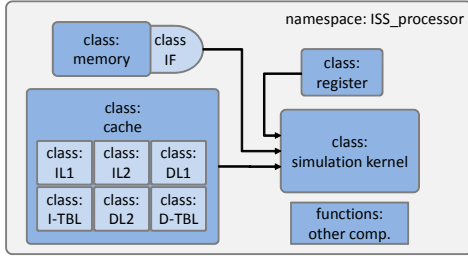

Figure 2. The new Simplescalar structure.

3.2 RTOS Model

The RTOS model is a scheduler implementation within a SystemC module that includes two SystemC threads: *schedule_thread* and *run_thread*. The *schedule_thread* determines the next schedule. It supports priority-based time slicing using round-robin and rate monotonic scheduling. The schedule for the application threads is stored within a priority list. This list is then delivered to the SystemC/ISS wrapper by the *run_thread*. Once a new schedule is available, the *run_thread* sends it and blocks until schedule execution of the ISS is finished. This information is delivered by a method of the *ISS_wrapper* class that will be introduced in Section 3.3. Figure 3 shows the SystemC RTOS model and the management structure of the internal processor kernel and thread lists.

To support multi-core systems, an internal priority-based thread list is applied for each processor kernel. If a new thread is created, it is assigned to one processor kernel. Load balancing is not supported until this point, but is planned for future applications, and can be implemented without changes to the *ISS_wrapper* class and the data structure of the thread list.
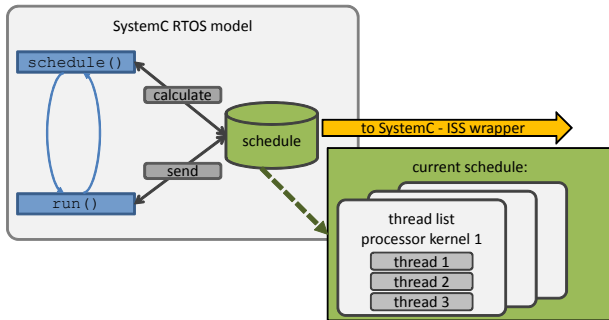

Figure 3. RTOS model with processor kernel and thread lists.

3.3 SystemC/ISS Wrapper

The SystemC/ISS wrapper is responsible for synchronization between the ISS and SystemC simulation kernel and communication between the RTOS model and ISS. It offers an interface to decouple the wrapper from the RTOS model and supports a concept for adding, removing, and switching threads at runtime.

3.3.1 Thread Abstraction

An application thread, which is executed by the ISS, is implemented as an abstract representation, the thread abstraction, in the SystemC/

ISS wrapper. This thread abstraction includes the actual information of its program or function depending on register and memory contents. A thread abstraction object is applied for each application thread. Table 1 shows the information and gives a short description. Additional information that is required by the *sim-outorder* simulator of Simplescalar is also stored in the thread class.

Table 1. Information included in the thread abstraction class.

variable	description
_name	unique thread name
*_registers	pointer to a particular register
*_memory	pointer to the memory
_argc	number of passed command line parameters
_argv	command line parameters
_filename	filename of the binary file
_finished	indicates if the thread has terminated
_core_name	name of processor this thread is running at
_state	state of thread (ready, running, blocked)
_num_of_child	number of child threads
_parent	name of the parent thread (if existing)
_first_run	indicates if the thread executes the first time
sim-outorder extensions	
_fetch_data	fetch buffer - stores fetched instructions
_pred_PC	address of next instruction
_recover_PC	instruction jump address after recover

3.3.2 ISS Wrapper - RTOS Model Interface

This interface also provides abstract functions to handle thread creation, thread deletion, thread switching and the like. It decouples the RTOS model and the ISS. The functions are activated by the RTOS model and influence the execution of the ISS. Figure 4 shows the structure of the *ISS_wrapper* with some of the interface methods. The thread abstraction class and some internal functions from which the ISS is called up are also depicted.
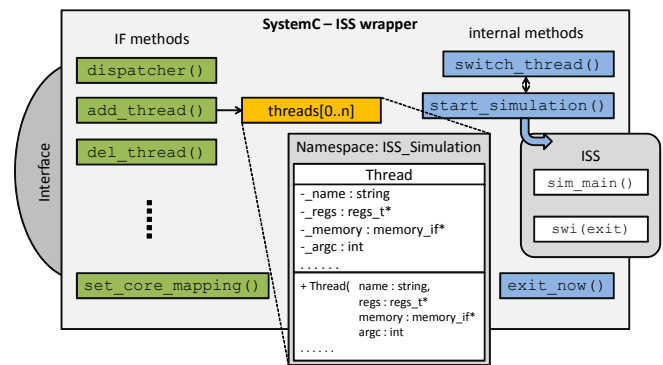

Figure 4. Wrapper structure and thread abstraction class.

Two important interface functions are dispatcher() and add_thread(). The dispatcher() function is used by the RTOS model to deliver the schedule information to the *ISS_wrapper* class. The add_thread() function adds a new thread to the wrapper. Two cases are possible: the new thread is loaded into a new address space or an existing address space. In the first possibility, a new instance of register class and memory class is created. Other-

wise, the new thread is loaded into an existing address space, which is true for a child thread. The address by itself is delivered by the function call. Additionally, the child thread needs its own stack. The free stack has to be divided for all threads within one address space. For the ARM instruction set, the new stack range is calculated as follows:

$$SB_C = SB_P - SG_P + 4 - n \cdot (SG_C + 4) \qquad (1)$$

DEFINITION 1.

SB_P: *Stack base address of parent thread.*
SB_C: *Stack base address of child thread.*
SG_P: *Maximum stack size of parent thread.*
SG_C: *Maximum stack size of child thread.*
n: Number of child threads within this address space.

3.3.3 Context Switching

The ISS itself keeps references of pointer variables to the register, memory, and data of the running thread. In case of a context switch, the pointers are replaced by the pointers to the new thread, as shown in Figure 5.
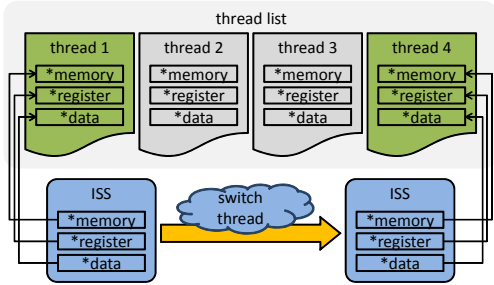

Figure 5. Example of a context switch.

A context switch is handled differently depending on the kind of simulator the ISS is based on. For cycle-true simulators like the Simplescalar *sim-outorder*, a context switch is not allowed at all times due to the internal ISS pipeline model. Some conditions may delay a context switch, for instance, loading an instruction from the cache. The processor blocks until the instruction is loaded, hence, a context switch is forbidden during that time. Also, if the processor runs in the speculative mode or performs a CISC operation, which is divided into several micro operations, a task switch is forbidden. If context switching is forbidden, the state of the current running thread is returned by the switch_thread() method. Otherwise, the state of the current running thread is stored and the state of the new thread is returned. The algorithm of the switch_thread() method is shown in Figure 6 as flow diagram.
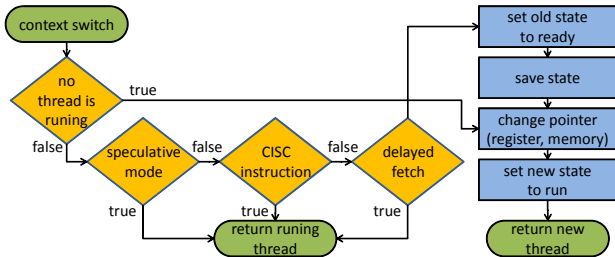

Figure 6. Flow diagram of the switch_thread() method.

3.3.4 Dynamic Thread Creation

To allow the application to dynamically create threads, the wrapper implements a mechanism to notify the RTOS model about the new, dynamically created thread. Simplescalar allows to recognize syscalls and to send them to the host operating system. In this approach, dynamic thread creation is intercepted, the thread information is sent to the RTOS model and added into the internal thread list, and the add_thread() method is invoked.

3.4 Entire Simulation Model

Figure 7 shows the architecture of the entire simulation model. The RTOS model calculates the schedule and tells the wrapper to switch the context to the specified thread. Within the wrapper, the schedule is received and the context is switched by bending the pointers to the register, memory, and data variables. Please note that despite using an abstract scheduler model running outside of the ISS, threads can be implemented without manually specified preemption points. Furthermore, thread interruption and context switching are cycle-accurate.
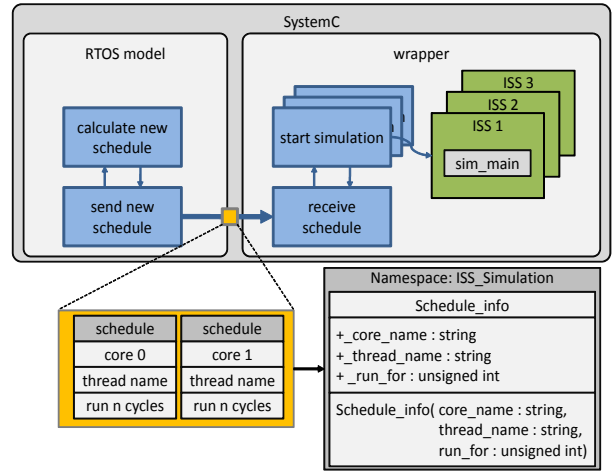

Figure 7. The entire simulation model.

Multi-processor/multi-core systems are supported by the SystemC/ISS wrapper, the new Simplescalar structure, and the RTOS model. To implement a multi-core system, multiple class-instances of the ISS are possible within the wrapper.

4. RESULTS

The new approach has been implemented using Simplescalar with ARM instruction set and has been compared with the implementation of the same simulator that executes the RTOS RTEMS [3].

4.1 Theoretical Results

4.1.1 Simulation Performance

The simulation speedup S of the combined approach is defined by the following equation:

$$S = \frac{ST_{RTOS}}{ST_{MODEL}} \qquad (2)$$

DEFINITION 2.

ST_{MODEL}: *Simulation runtime of the combined approach.*
ST_{RTOS}: *Simulation runtime if the RTOS is executed by the ISS.*
The variables are calculated by the two equations:

$$ST_{MODEL} = P + R_{MODEL} \qquad (3)$$
$$ST_{RTOS} = P + I + R_{RTOS} \qquad (4)$$

DEFINITION 3.

P: The runtime of the application program.

R_{MODEL}: *The runtime of the RTOS Model.*
R_{RTOS}: *The runtime of the RTOS on the ISS.*
I: The runtime of the idle-process of the RTOS on the ISS.
As a result, our approach has a speedup of:

$$S = \frac{P + I + R_{RTOS}}{P + R_{MODEL}} \qquad (5)$$

4.1.2 Accuracy

Saving simulation time also means a loss of accuracy. Since the RTOS is not executed by the ISS, its timing has to be annotated in the RTOS model. The accuracy loss is:

$$\frac{|CN_{RTOS} - CN_{Model}|}{CN_{RTOS}} \qquad (6)$$

DEFINITION 4.

CN_{RTOS}: *Total clock cycle numbers if RTOS is executed at ISS.*
CN_{Model}: *Total clock cycle numbers of the combined approach.*
The total clock cycle number is the sum of application clock cycles and RTOS clock cycles. To allow a higher accuracy of CN_{Model} the measured number of RTOS clock cycles is annotated in the model.

4.2 Experimental Results

We tested the approach for two scheduling algorithms: round robin scheduling (RR) and priority-based rate monotonic scheduling (RMS). The focus was to evaluate the experimental results with the theoretical results. The application consists of two tasks for both approaches. Figure 8 shows a comparison of the measured simulation time of the combined RTOS model/ISS approach and the RTEMS running on the ISS. The speedup is shown in relation to the thread switch rate that is defined by:

$$w = \frac{TSN}{CN} \cdot 10000 \qquad (7)$$

DEFINITION 5.

w: Thread switch rate per 10000 simulation cycles.
TSN: The total number of thread switches.
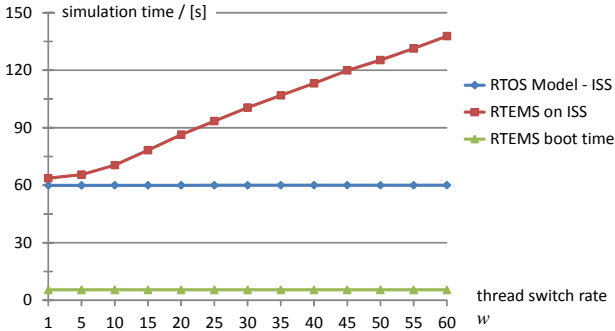*CN: The total cycle number.*

Figure 8. Comparison of simulation time with a varying thread switch rate for two threads and round robin scheduling.

The figure shows that the simulation time is steadily increasing for a growing number of thread switches for the RTEMS-on-ISS approach. Contrarily, the increasing number of thread switches has a minimum effect for the RTOS model/ISS approach, even for very big thread switch rates. This is because thread switches consume much more time with the ISS by executing instruction than outside the ISS by pointer replacement. The figure also shows the additional boot time for the RTEMS, which is about 5.2 seconds and marginal for long simulation times. The measured speedup of four different

numbers of w is shown in Figure 9 for RR and for RMS. For round robin, a high speedup is only achieved for high thread switching rates. In comparison, RMS has much higher costs and hence a better speedup.

Figure 9. Speedup for different thread switch rates and for different scheduling algorithms.

Table 2 compares the measured speedup with the speedup calculated by equation (5). For this, we determined the RTOS clock cycle, application clock cycle, and total clock cycle numbers. The values were inserted in equation (5) assuming that a clock cycle has a constant timing value. Furthermore, $R_{MODEL} \ll R_{RTOS}$ is assumed.

Table 2. Comparison of experimental and theoretical results.

total cycles @ ISS	RTOS cycles @ ISS	idle cycles	S (by equ. (5))	S (measured)
9129936	4283013	0	1.8837	1.8836
9152701	4303653	0	1.8876	1.8875
9390271	4537350	0	1.9350	1.9364
11778182	6872625	0	2.4010	2.4010
22395341	17236116	0	4.3408	4.3403

To take the timing of RTEMS into account, we annotated the runtime of the RTOS into the RTOS model using the SystemC *wait* statement. The cycle numbers for different RTEMS syscalls are measured to determine the total number of RTOS cycles; their mean value is presented in Table 3. Since the variance of the measured values is very small, the annotation of the mean value is almost accurate.

Table 3. Mean value of measured cycles for RTEMS syscalls.

syscalls (RTEMS)	# of cycles
boot sequence	4269493
rtems_task_create	1671
rtems_task_start	347
rtems_task_mode	213
rtems_task_set_priority	244
rtems_task_wake_after(YIELD)	129
rtems_task_delete	2305

Table 4 shows the accuracy loss in equation (6) for different numbers of simulation cycles. Although the accuracy loss is marginal for our small application, it might be bigger for more complex applications. Nonetheless, this loss is still small with reference to the entire application.

In summary, the experimental results show a speedup from removing and replacing the RTOS scheduler by a RTOS scheduler model. Further speedup is expected depending on idle task and more expensive scheduling algorithms. Outsourcing further RTOS functionality (e.g. interprocess communication, interrupt handling as

well as load balancing algorithms for multi-core systems) will also lead to an increase in performance.

Table 4. Accuracy loss of the RTOS model.

CN_{Model}	CN_{RTOS}	accuracy loss in percent
9130679	9129936	0.0081%
9153444	9152704	0.0081%
9389517	9390271	0.0080%
11764153	11778182	0.1191%
22337821	22395341	0.2568%

5. CONCLUSION AND SUMMARY

This paper has presented an approach for combining abstract RTOS models and instruction set simulation for distributed embedded software. The simulation framework allows evaluating decisions with reference to RTOS and scheduling strategies by taking the cycle-accurate application behavior into account. Furthermore, multi-processor/multi-core systems and the integration of the entire system into the simulation framework are supported. The approach has been implemented using Simplescalar and a scheduler model in SystemC. The speedup has been shown in comparison to an ISS that executes the RTOS scheduler.

Future work includes the outsourcing of further RTOS functionality into the RTOS model, the implementation of load balancing strategies, and their evaluation with reference to the target system.

6. REFERENCES

[1] OSCI. http://www.systemc.org.
[2] SpecC. http://www.specc.org.
[3] RTEMS. http://www.rtems.com.
[4] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-Performance Timing Simulation of Embedded Software. *Proc. of the 45th Design Automation Conference (DAC)*, Anaheim, CA, USA, 2008.
[5] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW Performance Estimation Framework for Early System-Level-Design Using Fine-Grained Instrumentation. *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, Munich, Germany, 2006.
[6] K. Karuri, M.A. al Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-grained Application Source Code Profiling for ASIP Design. *Proc. of 42nd Design Automation Conference (DAC)*, Anaheim, CA, USA, 2005.
[7] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr. HySim: a Fast Simulation Framework for Embedded Software Development. *Proc. of the 5th international conference on hardware/software codesign and system synthesis (CODES+ISSS)*, Salzburg, Austria, 2007.
[8] Y. Hwang, S. Abdi, and D. Gajski. Cycle Approximate Retargettable Performance Estimation at the Transaction Level. *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, Munich, Germany, 2008.
[9] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A Universal Techinque for Fast and Flexible Instruction-Set Architecture Simulation. *Proc. of the Design Automation Conference (DAC)*, New Orleans, LA, USA, 2002.
[10] Simplescalar LLC. http://www.simplescalar.com.
[11] T. Austin, E. Larson, and D. Ernst. SimpleScalar: an Infrastructure for Computer System Modeling. *Computer*, 35(2):59-67, 2002.
[12] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. SystemC Co-simulation and Emulation of Multi-Processor SoC Designs. *Computer*, 36(4):53-59, 2003.
[13] F. Fummi, S. Martini, G. Perbellini, and M. Poncino. Native ISS-SystemC Integration for the Co-Simulation of Multiprocessor SoC. *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, Paris, France, 2004.
[14] L. Formaggio, F. Fummi, and G. Pravadelli. A Timing-Accurate HW/SW Cosimulation of an ISS with SystemC. *Proc. of the 2nd international conference on hardware/software codesign and system synthesis (CODES+ISSS)*, Stockholm, Sweden, 2004.
[15] F. R. Boyer, Liping Yang, E. M. Aboulhamid, L. Charest, and G. Nicolescu. Multiple SimpleScalar Processors, with Introspection, under SystemC. *Proc. of the 46th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS '03)*, Cairo, Egypt, 2003.
[16] R. Zhong, Y. Zhu, W. Chen, M. Lin, and W.-F. Wong. An Inter-Core Communication Enabled Multi-Core Simulator Based on SimpleScalar. *Proc. of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINA)*, Niagara Falls, Canada, 2007.
[17] A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modeling for System Level Design. *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, Munich, Germany, 2003.
[18] P. Hastono, S. Klaus, and S.A. Huss. Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded Systems. *Forum on Specification & Design Languages (FDL)*, MACC-University of Lille, France, 2004.
[19] J. Madsen, K. Virk, and M. J. Gonzalez. A SystemC-Based Abstract Real-Time Operating System Model for Multiprocessor System-on-Chip. *Multiprocessor System-on-Chip*, Morgan-Kaufmann Publishers, 2004.
[20] H. Posadas, J. A. 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*, Vol. 10, No. 4, Springer, 2005.
[21] H. Yu, A. Gerstlauer, and D. Gajski. RTOS Scheduling in Transaction Level Models. *Proc. of the 1st international conference on hardware/software codesign and system synthesis (CODES+ISSS)*, Newport Beach, CA, USA, 2003.
[22] R. Le Moigne, O. Pasquier, and J. P. Calvez. A Generic RTOS Model for Realtime Systems Simulation with SystemC. *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, Paris, France, 2004.
[23] F. Hessel, V. da Rosa, C. Reif, C. Marcon, and T. Dos Santos. Scheduling Refinement in Abstract RTOS Models. *ACM Transactions on Embedded Computing Systems*, Vol. 5, No. 2, 2006.
[24] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada. RTOS-Centric Hardware/Software Cosimulator for Embedded System Design. *Proc. of the 2nd international conference on hardware/software codesign and system synthesis (CODES+ISSS)*, Stockholm, Sweden, 2004.
[25] M. Hassan, K. Sakanushi, Y. Takeuchi, and M. Imai. RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC. *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, Munich, Germany, 2005.
[26] W.-T. Sun and Z. Salcic. Modeling RTOS for Reactive Embedded Systems. *Proc. of the 20th International Conference on VLSI Design*, Bangalore, India, 2007.
[27] Y. Yi, D. Kim, and S. Ha. Virtual Synchronization Technique with OS Modeling for Fast and Time-Accurate Cosimulation. *Proc. of the 1st international conference on hardware/software codesign and system synthesis (CODES+ISSS)*, Newport Beach, CA, USA, 2003.
[28] D. Kim, C.-E. Rhee, Y. Yi, S. Kim, H. Jung, and S. Ha. Virtual Synchronization for Fast Distributed Cosimulation of Dataflow Task Graphs. *Proc. of the 15th International Symposium on System Synthesis*, Kyoto, Japan, 2002.
[29] D. Kim, Y. Yi, and S. Ha. Trace-driven HW/SW Cosimulation Using Virtual Synchronization technique. *Proc. of the 42nd Design Automation Conference (DAC)*, Anaheim, CA, USA, 2005.
[30] Y. Yi, D. Kim, and S. Ha. Fast and Accurate Cosimulation of MPSoC Using Trace-Driven Virtual Synchronization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 12, 2007.