Automatic Generation of Cycle-Approximate TLMs with Timed RTOS Model Support

Yonghyun Hwang[1], Gunar Schirner[2], and Samar Abdi[3]

[1] University of California, Irvine, USA, yonghyuh@cecs.uci.edu
[2] Northeastern University, Boston, USA, schirner@ece.neu.edu
[3] Concordia University, Montreal, Canada, samar@ece.concordia.ca

Abstract. This paper presents a technique for automatically generating cycle-approximate transaction level models (TLMs) for multi-process applications mapped to embedded platforms. It incorporates three key features: (a) basic block level timing annotation, (b) RTOS model integration, and (c) RTOS overhead delay modeling. The inputs to TLM generation are application C processes and their mapping to processors in the platform. A processor data model, including pipelined datapath, memory hierarchy and branch delay model is used to estimate basic block execution delays. The delays are annotated to the C code, which is then integrated with a generated SystemC RTOS model. Our abstract RTOS provides dynamic scheduling and inter-process communication (IPC) with processor- and RTOS-specific pre-characterized timing. Our experiments using a MP3 decoder and a JPEG encoder show that timed TLMs, with integrated RTOS models, can be automatically generated in less than a minute. Our generated TLMs simulated three times faster than real-time and showed less than 10% timing error compared to board measurements.

Key words: Transaction Level Modeling, Timed RTOS Modeling

1 Introduction

The importance of embedded software (SW) in heterogeneous multi-processor systems is increasing with increasing complexity of the systems. Choosing an efficient platform and a suitable SW mapping is essential to meet performance and power constrains. Estimating software performance early in the design flow, e.g. during design space exploration, is essential for achieving an efficient implementation. Traditional ISS-based approaches are performance limited, especially in a multi-processor context. Abstract simulation of SW execution is one key solution for rapid design space exploration and early prototyping.

Important influence factors for SW performance, are the hardware platform (e.g. processor type, memory hierarchy) and, equally important, SW architecture and configuration: task/data granularity, selection of scheduling policy, priority distribution and the selection of an appropriate RTOS. Automatic generation of timed TLMs that reflect the effects of the above design choices is needed to enable informed decision in a simulation-based approach.

In order to accurately estimate performance of a SW task three delay contributors have to be modeled: (a) D_{exec}, for execution of straight line code; (b) D_{sched}, for communication and dynamic scheduling (e.g. scheduling of a higher priority task); and (c) D_{sys}, the delay due to the system overhead as a result of dynamic scheduling. Existing dynamic approaches only address (a) D_{exec} and (b) D_{sched}. D_{exec} can be modeled through on-line code profiling, e.g. [16] or off-line profiling and timing annotation of the modeled code, e.g. [13, 12]. Modeling of D_{sched} can be addressed by using abstract RTOS models emulating dynamic scheduling, e.g. [16, 8, 10]. However, current solutions do not model system overhead, i.e. the overhead of executing an RTOS on the target processor remains unaccounted.

Modeling system overhead is essential in guiding the SW developer in parallelizing a given application. Choosing a too fine granularity of Inter-Process Communication (IPC) may unnecessarily increase the number of context switches, thus increase D_{sys} and therefore decrease system performance [4]. With current modeling techniques the negative effects of such a design choice are only discovered when executing on the final system, leading to an expensively long design cycle. To increase efficiency of the design process, system overhead modeling, D_{sys}, is required. In this paper, we present our approach for automatically generating cycle-approximate TLMs that include timed abstract RTOS models, hence reflecting all three delays D_{exec}, D_{sched} and D_{sys}.

This paper is organized as follows. Section 2 presents related work. Section 3 outlines the TLM generation framework, and describes our estimation and annotation approach covering D_{exec}. Section 4 outlines our abstract RTOS model yielding D_{sched}. Section 5 introduces our analysis and modeling of system overhead D_{sys}. Our experimental results, Section 6, show scalability and accuracy based on several design examples. Finally, we conclude the paper and touch on future work.

2 Related Work

Significant research effort has been spent for early performance estimation of multi-processor systems, which can be broadly categorized as static, semi-static and dynamic. Static approaches, such as [17], on one side use purely analytical methods to compute application delays. Dynamic approaches on the other side, use platform models to generate a timed executable model of the design, which later produces the estimation data at run time. ISS and virtual platforms are popular examples of dynamic approaches. Our solution is based on a dynamic approach.

SW performance estimation techniques [13, 12, 2, 3] utilize the execution path of the SW. The common approach is to multiply the cost of operation with the total number of operations executed. Unlike above techniques, [14, 5] can take into account the datapath structure. However, they use bus functional models and generate ISS which provides accurate results at the expense of speed. Sim-

plescalar [1] is a well known retargetable ISS that can provide cycle accurate estimation result but is several orders slower than TLM.

Abstract RTOS models have been developed on top of System Level Design Languages (SLDLs) (e.g. SystemC [9], SpecC [7]) to expose the effect of dynamic scheduling. Examples include [8, 20], modeling typical RTOS primitives on top of an SLDL, and [16], implementing an POSIX API on top of SystemC. The latter offers an interesting combination of online estimation and RTOS-modeling, which enables an optimized modeling of periodic interrupts. However, the solutions do not include modeling of RTOS overheads, which we address in this paper. An RTOS centric cosimulator using a host compiled RTOS is described in [10], which, however, does not include target execution time simulation.

3 TLM Estimation Framework

Our support for RTOS overhead modeling and integration into TLM is built upon the TLM based design infrastructure proposed by Gajski et al. [6]. In this methodology, the user specifies the system definition consisting of application code and a graphically notated platform specification. The application is expressed as parallel executing C/C++ processes, which communicate through a set of standard communication channels. The framework generates TLMs based

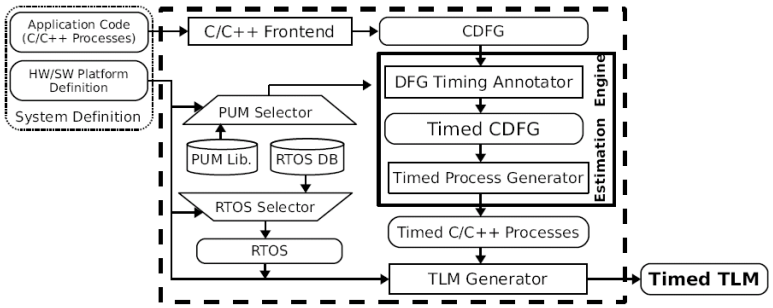


Fig. 1. Estimation Tool Architecture

on the design decisions to verify system performance and functionality early in the design cycle. The TLMs then serve as an input to cycle accurate synthesis, which produces Pin/Cycle Accurate Models (PCAM) for detailed validation and analysis. After finalizing the design, the PCAMs are used for low level synthesis generating board level prototype design.

Figure 1 shows the architecture and implementation of our timing estimation tool with RTOS support integrated into th framework. The input specification contains untimed C code for each process and and its mapping to PEs in the platform. According to the mapping decisions, the user code is analyzed based on a HW processing unit model (PUM), which characterizes datapath, memory hierarchy and instruction scheduling of the PE. We have implemented a C/C++

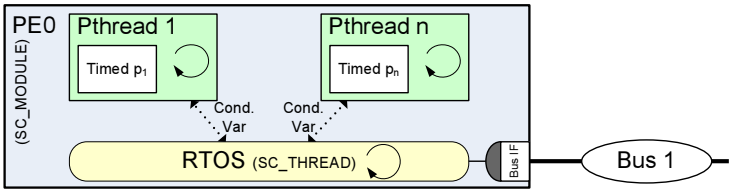


Fig. 2. Threads in Abstract RTOS

front-end using the LLVM compiler infrastructure [15] which parses the application code into CDFGs. For each basic block of the CDFG, the corresponding DFG is input to an estimation engine. The estimation engine computes the DFG delay by scheduling application instructions based on the PE's PUM. Using the LLVM source transformation API, the estimated delay is then annotated into the CDFG data structure. Subsequently, timed C code is generated for the process using the LLVM code generation API.

During TLM generation, the SW platform definition is used for configuring the abstract RTOS model with overhead delays and scheduling policy. Finally, the annotated C code and the configured RTOS model are compiled and linked with a SystemC programming model of the platform to generate the executable timed TLM. In this paper, we will focus on the timed abstract RTOS aspect. Timed C code generation for single process mapping without dynamic scheduling is described in [11].

4 Abstract RTOS Model

Multiple processes may be mapped to the same processing element, later scheduled by an RTOS on actual HW. For early exploration of dynamic scheduling effects, an abstract RTOS is integrated into the TLM. Each mapped process becomes a virtual process on the abstract RTOS. The RTOS model, as well as the virtual processes are configurable in their scheduling parameters, supporting a validation of the SW configuration (e.g. scheduling policy selection, priority distribution).

As shown in Figure 2, our abstract RTOS executes as an SC_THREAD inside the SC_MODULE of the PE. Each PE runs an own abstract RTOS. The RTOS provides services to start, stop and control virtual processes inside its context. It furthermore communicates with the outside of the processor through the bus interface (provided by an an abstract bus model) and interrupts.

Our RTOS model uses pthreads, native to the host operating system, to provide multiple flows of execution. Each virtual process (i.e. a time annotated process) is executed in an own pthread. The execution of each pthread is controlled by the RTOS model through condition variables to emulate the selected scheduling policy. At any given point in time only one pthread (one virtual process) for each PE is released through the condition variable, overwriting the host's scheduling policy.

The abstract RTOS, similarly to an actual RTOS, maintains a Task Control Block (TCB) for each virtual process. Each virtual process is scheduled according to a task state machine, with states such as RUNNING, READY, PENDING, SUSPENDED; and the abstract RTOS maintains the appropriate queues.

Each primitive in the TLM, which could potentially trigger a context switch, is executed under control of the RTOS These primitives include task control and inter-process transactions. Figure 3 shows two virtual processes executing on top of the RTOS model. Interactions with the model are indicated by arrows. The RTOS model provides an fixed API and the TLM generator produces appropriate wrappers for the generated timed process code (Section 3) to utilize this API. To give an example, when a virtual process executes an inter-process transaction (e.g. Figure 3, call *ipcRecv()* in process *Timed_P2*), this transaction is executed under RTOS control. Assuming it contains acquiring a non-available semaphore, the virtual process state is set to PENDING, the next process is selected from the ready queue according to the scheduling policy, set to RUNNING, and released through its condition variable. Finally, the old virtual process suspends by waiting its own condition variable. Note that the actual context switch is performed by pthreads of the host OS, the abstract RTOS only controls their release.

In addition to classical RTOS primitives, our abstract RTOS model provides an API to simulate progress of time due to code execution. Our time annotated code calls this interface (*execDelay()*, Figure 3), and subsequently a *sc_core::wait()* is executed under RTOS control. During this time, the virtual process remains in the RUNNING state and primarily no other process is scheduled. During the time progress, an incoming interrupt may release a higher priority task. Similarly, our abstract RTOS provides services for external bus access, which are the performed under RTOS control.

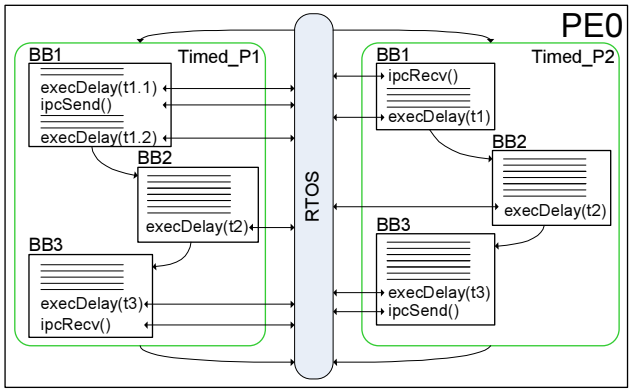


Fig. 3. Abstract RTOS interaction

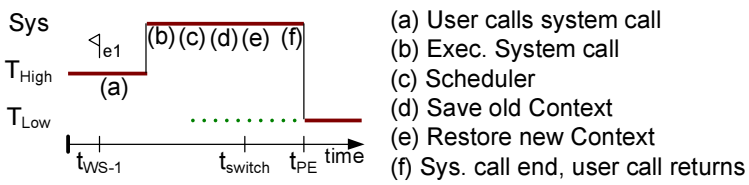


Fig. 4. sem_wait() with context switch

5 Modeling of RTOS Overheads

An abstract RTOS exposes the effects of dynamic scheduling. In addition, a system overhead delay model is needed to provide feedback about the overhead in a multi-tasking application, capturing for example the delay due to IPC, context switching and interrupt preemption. Modeling such overheads is essential to guide the developer in partitioning the code (e.g. for deciding granularity of data and communication handling). To illustrate, splitting an application into too many tasks may lead to a system overhead that dominates the application performance (too many context switches and inter-process transactions). The execution time of user code may not be a sufficient indicator for a potential performance bottleneck, which drives the need for modeling RTOS overheads.

Modeling RTOS overhead is challenging as it depends on RTOS, CPU, and CPU configuration (e.g. caching). The analysis is further complicated by a limited source code availability especially for a commercial RTOS , as well as API and structural/organizational differences between implementations. These factors inhibit a static source code analysis or make it prohibitly expensive.

We have developed a time stamping approach to analyze RTOS overheads on RTOS API level without source code analysis. We characterize a RTOS on the actual processor in supported configuration(s). The determined overhead characteristics are stored in our database. As they are independent of processor external HW and independent of the user application, they can be applied in to many designs. Our TLM generator reads the DB to instantiate an abstract RTOS inside a PE model with specific delay parameters.

We developed a special test application that captures time stamps and invokes RTOS primitives in a controlled environment in which we know a priori the scheduling outcome. We use a processor external timer to measure time. The time stamp code and its data are exclusively placed in a non-cached fast local memory (BRAM) to minimize impact on caching and execution time. We disable timer interrupts while analyzing timing unrelated RTOS primitives to eliminate the impact of unexpected interrupts.

Figure 5 and Figure 4 show an example of our analysis based on acquiring a semaphore without and with a context switch. As system calls basically follow the same sequence the example is representative.

In Figure 4, task T_{High} calls *sem_wait()* to acquire non-available sema-phore (a), which results in a context switch to T_{Low}. We record time stamp t_{WS-1} in user mode before the call. Next, the processor mode is switched to system

mode, and the system call (the actual semaphore code) is executed (b). Then, the scheduler (c) determines the next task to execute, the current task's context is saved (d), the new task's context is restored (e). Finally, a system call returns (f) in the new task's context. In our application, T_{low} had earlier relinquished the CPU by posting a semaphore to T_{high}. Therefore, the returned system call is a *sem_post()*, and we record time stamp t_{PE}. Please note, that the code for returning a system call is independent of the call type (e.g. the code is identical for *sem_post()* and *sem_wait()* starting at the scheduler call (c)). We can therefore use t_{PE} for analyzing the duration of a *sem_wait()*.

Figure 5 illustrates the case of an available semaphore where no context switch occurs. We record t_{WS-2} before the call. The sequence during the system call is shorter. The scheduler (c) determined no context switch and the system call returns (f) to the same task where we record t_{WE}. Based on these time stamps, we determine the duration for *sem_wait()*, and for a context switch. We separate these two, in order to simplify abstract modeling and database. Then, only a single delay characterizes each RTOS primitive. Our analysis with these time stamps, after eliminating the overhead of time stamping itself, is as follows:

$$Dur(sem\_wait) = t_{WE} - t_{WS-2}$$
$$Dur(switch) = t_{PE} - t_{WS-1} - Dur(sem\_wait)$$

The duration estimation of *sem_wait* is the difference between start and end time stamp. We compute the context switch duration based on the duration of the *sem_wait* with a context switch ($t_{PE} - t_{WS-1}$) and subtract the time for *sem_wait()* without context switch. We chose *sem_wait()*, since we expect portion (b), execution of the system call itself, to be minimally dependent on the scheduling outcome as *sem_wait()* only manipulates the own task state. Conversely, *sem_post()* shows a variance beyond the estimated context switch duration as it manipulates other task's states.

We analyze other communication and synchronization primitives in a similar manner. For each primitive we measure the time without and with a context switch. After normalizing for the already determined context switch duration, we calculate the average between the two cases to determine the primitive's duration.

In addition to the basic RTOS primitives, we also characterized standard functions that are dependent on the data length. In particular, we have characterized *memcpy()* as it is frequently used and its code is often heavily optimized.

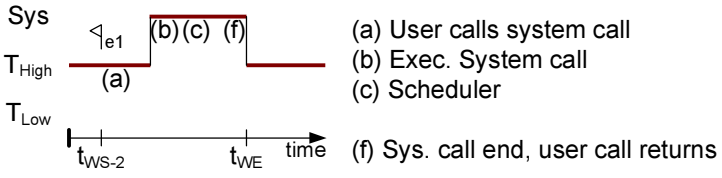
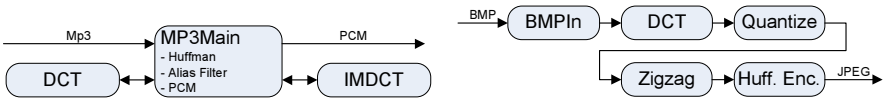


Fig. 5. sem_wait() without context switch



(a) MP3 decoder (b) JPEG encoder

Fig. 6. Example applications.

We capture the delay results in a table, and use a linear extension to estimate values beyond the table boundaries.

We store the analyzed RTOS characteristics in our database, with a separate delay for each used RTOS primitives, and the basic context switch. During TLM generation, the code for instantiating an RTOS is created. The selected RTOS' characteristics are retrieved from the database and the abstract RTOS is configured to reflect the selected RTOS. Upon execution of an abstract RTOS primitive, the characterized delay – without context switch – (e.g. Dur(*sem_wait()*)) is executed. Our abstract task dispatcher, switching between pthreads, is annotated with $Dur(switch)$. We use this basically state less delay model to simplify abstract RTOS implementation maintaining a high simulation performance. While our analysis and modeling approach abstracts away many influences on RTOS overhead (e.g. number of total, waiting, and manipulated tasks, scheduler implementation) and therefore is not cycle-accurate, it already yields valuable feedback for estimating system performance.

6 Experimental Results

To evaluate the benefits of our approach, we have applied it to three designs based on an MP3 decoder, a JPEG encoder and a combined design running both applications. Figure 6 shows the application flow. The MP3 decoder executes in 3 tasks, and JPEG encoder with 5 tasks. The designs originate from an hardware oriented design showing a fine grained task split. All tasks are mapped in the target plaform to the same processor, a Microblaze [19] with 100Mhz, and scheduled by Xilinx's RTOS, Xilkernel [18].

We generate the timed TLMs with timed RTOS using our generation approach. SW binaries were generated for execution on the Xilinx Virtual Platform (XVP) and for execution on the actual processor. Matching hardware designs were synthesized using Xilinx ISE and EDK and downloaded to an Xilinx FF896 board to provide a cycle-accurate reference platform.

Table 1 shows the average accuracy of our abstract models for each of our designs in comparison to cycle-accurate execution on the Xilinx FF896 board. We analyze several abstraction levels providing insight about each delay's contribution. We analyze:

Timed TLM captures D_{exec} at basic block level (Section 3). Tasks may execute concurrently.

TLM w/ RTOS adds an abstract RTOS (Section 4) resolving dynamic scheduling, capturing D_{exec} and D_{sched}.

TLM w/ Timed RTOS additionally models system overheads (Section 5), reflecting D_{exec}, D_{sched}, and D_{sys}.

In addition to our models, we also compare to the XVP.

	Timed TLM	TLM w/ RTOS	TLM w/ Timed RTOS	XVP
JPEG	-75%	-35.56%	-9.98%	50%
MP3	-41%	-25.95%	5.29%	7%
MP3+JPEG	-83%	-33.25%	-6.20%	37%

Table 1. Accuracy of abstract models [%].

Our results show that only modeling D_{exec} is not sufficient for parallel applications. The timed TLM shows 66% average error, up to 83% depending on application parallelizm. Adding dynamic scheduling by an abstract RTOS dramatically improves accuracy. However, with the fine grained IPC, the designs exhibit a significant system overhead and thus the TLM with RTOS underestimates by 32% on average. This result can be compared to other state of the art solutions, which all do not model any RTOS overhead. Adding RTOS overhead modeling reduces the error to less than 10%, yielding already sufficiently accurate timing information. The remaining error is due to our abstract analysis and modeling of RTOS overheads, which we chose in favoring automate ability and simulation speed. Lastly, we compare against the commercial XVP. It significantly overestimates our designs with 31% on average, which can be traced back to inaccurate modeling of memory accesses [11]. Comparing all solutions, our TLM with Timed RTOS yields the most accurate timing estimation. With decreasing system overhead, an even smaller error can be expected. For a single task version of the MP3 decoder our generated TLM exhibited only 0.9% error.

Table 2 summarizes the simulation time in seconds of real-time (or wall clock time). In addition the above discussed models, we also included a purely functional TLM without any timing annotation and execution on real HW.

As expected, native execution on the simulation host yields high performance. Increasing model complexity increases simulation time. The functional TLM executed the fastest (within milliseconds), as it requires the fewest context switches. Executing time annotations *(sc_core::wait())* in the timed TLM, increases simulation time to tens of milliseconds. Our TLM with RTOS model, which models the virtual threads, executes in fractions of a second. No significant increase is measurable for reflecting RTOS overheads. Our TLM with timed RTOS executes about 3 times faster than real-time comparing to execution on the Microblaze.

	Func. TLM	Timed TLM	TLM w/ RTOS	TLM w/ T. RTOS	XVP	Board
JPEG	0.003	0.02	0.25	0.27	168	0.83
MP3	0.002	0.01	0.08	0.08	60	0.34
MP3+JPEG	0.004	0.04	0.32	0.33	213	1.17

Table 2. Simulation time (real-time) [sec].

Feature	JPEG	MP3	MP3+JPEG
# IPC / switch	720 / 1440	392 / 783	1112 / 2225
appl. cycles	53.2E+6	25.0E+6	78.2E+6
system cycles	21.1E+6	10.6E+6	31.7E+6

Table 3. Complexity of Models.

In addition, our solution is three orders of magnitude faster than the commercial XVP. These results clearly demonstrate the advantages of our solution, simulating faster than real-time while exhibiting less error than the XVP.

Generation time is an addtional usability aspect. Our TLM generation time is dominated by the SW performance estimation (Section 3), as it executes the LLVM compiler. The additional effort for instanciating the timed RTOS is negligible. The measured total generation time ranges from 1.2 seconds (JPEG) to 33.3 seconds (MP3+JPEG).

In addition to advantages in accuracy and performance, our solution also provides vital statistics. Table 3 shows a small excerpt, listing IPC calls, context switches, as well as number of cycles for executing the application and system (indicating the RTOS overhead). Especially the latter two reveal important information for developing multi-tasking designs. It is alarming for the shown design examples: a significant portion (about 30%) of the total execution time is spend in system overhead. This should urge improving the SW implementation by coarsening granularity – a feedback previously only available when executing on the board.

7 Conclusions

In this paper, we have presented a tool for automatically generating cycle-approximate TLMs. Our approach offers a complete solution for SW simulation including three essential aspects: (a) cycle-approximate retargetable performance estimation for SW execution, (b) dynamic scheduling through an abstract RTOS, and (c) modeling of RTOS overheads. Especially the latter is important as it offers a competitive advantage in guiding developers while designing multi-tasking applications. It exposes performance bottlenecks earlier in the design, thus enabling designing more efficient systems in fewer design cycles.

Results with MP3 decoder and JPEG encoder designs showed that our TLM generation is scalable to complex platforms and simulation results are within 10% of actual board measurements even for applications with high system overhead, while simulating faster than real-time. Our results demonstrate generation of TLMs for fast, early, and accurate estimation.

In future we plan extend our RTOS overhead analysis to finer grained detail using non-intrusive time stamping methods and to extend state dependent overhead modeling.

Acknowledgment

We would like to thank the Center for Embedded Computer Systems at University of California, Irvine for supporting this research.

References

1. T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.
2. J. R. Bammi, W. Kruijtzer, and L. Lavagno. Software Performance Estimatioin Strategies in a System-Level Design Tool. In *CODES*, San Diego, USA, 2000.
3. L. Cai, A. Gerstlauer, and D. Gajski. Retargetable Profiling for Rapid, Early System-Level Design Space Exploration . In *DATE*, San Diego, USA, June 2004.
4. Y. Cho, N.-E. Zergainoh, K. Choi, and A. A. Jerraya. Low Runtime-Overhead Software Synthesis for Communicating Concurrent Processes. In *RSP*, Porto Alegre, Brazil, 2007.
5. M.-K. Chung, S. Na, and C.-M. Kyung. System-Level Performance Analysis of Embedded System using Behavioral C/C++ model. In *VLSI-TSA-DAT*, Hsinchu, Taiwan, 2005.
6. ESE: Embedded Systems Environment. http://www.cecs.uci.edu/ ese.
7. D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.
8. A. Gerstlauer, H. Yu, and D. D. Gajski. Rtos modeling for system level design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.
9. T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
10. S. Honda et al. RTOS-Centric Hardware/Software Cosimulator for Embedded System Design. Stockholm, 2004.
11. Y. Hwang, S. Abdi, and D. Gajski. Cycle-approximate Retargetable Performance Estimation at the Transaction Level. In *DATE*, Munich, Germany, Mar. 2008.
12. 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. In *DATE*, Munich, Germany, March 2006.
13. M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli. A Compilation-based Software Estimation Scheme for Hardware/Software Co-simulation. In *CODES*, Rome, 1999.
14. J.-Y. Lee and I.-C. Park. Time Compiled-code Simulation of Embedded Software for Performance Analysis of SOC design. In *DAC*, New Orleans, USA, June 2002.
15. LLVM(Low Level Virtual Machine) Compiler Infrastructure Project. http://www.llvm.org.
16. H. Posadas et al. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. 10(4):209–227, Dec. 2005.
17. J. T. Russell and M. F. Jacome. Architecture-level Performance Evaluation of Component-based Embedded Systems. In *DAC*, Anaheim, USA, June 2003.
18. Xilinx. *"OS and Libraries Document Collection"*, 2006.
19. Xilinx. *MicroBlaze Processor Reference Manual*. 2007.
20. 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: Principles and Practice*. 2009.