RTOS Scheduling in Transaction Level Models

Haobo Yu, Andreas Gerstlauer, Daniel Gajski

CECS Technical Report 03-12
March 20, 2003

Center for Embedded Computer Systems
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{haoboy,gerstl,gajski}@ics.uci.edu

RTOS Scheduling in Transaction Level Models

Haobo Yu, Andreas Gerstlauer, Daniel Gajski

{haoboy,gerstl,gajski}@ics.uci.edu

Abstract

Rasing the level of abstraction in system design promises to enable faster exploration of the design space at early stages. While scheduling decision for embedded software has great impact on system performance, it's much desired that the designer can select the right scheduling algorithm at high abstraction levels so as to save him from the error-prone and time consuming task of tuning code delays or task priority assignments at the final stage of system design. In this paper we tackle this problem by introducing a RTOS model and an approach to refine any unscheduled transaction level model (TLM) to a TLM with RTOS scheduling support. The automation of the RTOS scheduling refinement process provides a useful tool to the system designer to quickly evaluate different dynamic scheduling algorithms and make the optimal choice at the early stage of system design. Experiments with the tool on a system design example shows the usefulness of our approach.

Contents

List of Figures

RTOS Scheduling in Transaction Level Models

Abstract

Rasing the level of abstraction in system design promises to enable faster exploration of the design space at early stages. While scheduling decision for embedded software has great impact on system performance, it's much desired that the designer can select the right scheduling algorithm at high abstraction levels so as to save him from the error-prone and time consuming task of tuning code delays or task priority assignments at the final stage of system design. In this paper we tackle this problem by introducing a RTOS model and an approach to refine any unscheduled transaction level model (TLM) to a TLM with RTOS scheduling support. The automation of the RTOS scheduling refinement process provides a useful tool to the system designer to quickly evaluate different dynamic scheduling algorithms and make the optimal choice at the early stage of system design. Experiments with the tool on a system design example shows the usefulness of our approach.

1   Introduction

Real time systems differs fundamentally from other systems in that both *computation result* and *time* affect the correctness of the whole system. These two aspects are addressed separately in system design. The computation correctness is usually determined at the early stage of system design by a high level model, whereas the actual timing properties are checked at run time through target specific binary code implementation. Wether a piece of computation can be finished on time or not largely depends on both the scheduling scheme (tasks and their priorities) and the system architecture. Since the scheduling behavior is hard to capture through high level model simulation, the timing properties of a system design usually change from high level model to implementation. As a result, the designer has to tune code delays or task priority assignments at final stage of system design which is both error prone and time consuming. However, this situation can be avoided if we provide a way to abstract the dynamic scheduling behavior and adjust the scheduling algorithm at higher abstraction levels.

Transaction level modeling is a high level approach to model digital systems where communication among system components is separated from the implementation of the processing elements (PE) [11]. This allows to abstract the communication between PEs independently from the implementation of the PEs. A high level of communication abstraction achieves high simulation speeds, hence enabling early architecture exploration and speeding embedded software development.

Many designers use preemptive, priority-driven and task-based real time operating systems (RTOS) [1, 3] to support the dynamic real-time behavior of the the system. To capture the dynamic scheduling behavior at higher level, we need techniques to abstract the RTOS scheduling because using a detailed, real RTOS implementation would negate the purpose of a high level model. Furthermore, at higher levels, not enough information might be available to target a specific RTOS.

In this paper, we address this design challenge by introducing a high level RTOS model and a set of refinement steps to create a TLM with RTOS scheduling support from any unscheduled TLM. We make a scheduling refinement tool implementing these refinement steps. The output model generated by our tool provides simulation result close to the final implementation (in terms of RTOS timing) and the tool can be easily integrated into the existing system level design flows to accurately evaluate a potential system design (e.g. in respect to timing constraints) for early and rapid design space exploration.

The rest of this paper is organized as follows: Section 2 gives an insight into the related work on software modeling and synthesis. Section 3 describes how the scheduling refinement process is integrated with the system level design flow. Section 4 provides information of the RTOS model used to model dynamic scheduling, Section 5 gives the detailed information of the RTOS scheduling refinement process. Experimental results are shown in Section 6 and Section 7 concludes this paper with a brief summary and an outlook on future work.

2 Related Work

A lot of work recently has been focusing on automatic RTOS and code generation for embedded software. In [7], a method for automatic generation of application-specific operating systems and corresponding application software for a target processor is given. In [4], a way of combining static task scheduling and dynamic scheduling in software synthesis is proposed. While both approaches mainly focus on software synthesis issues, their papers do not provide any information regarding high level model of dynamic scheduling integrated into the whole system.

In [12], a technique for modeling fixed-priority preemptive multi-tasking systems based on concurrency and exception handling mechanisms provided by SpecC is shown. However, their model is limited in its support for different scheduling algorithms and inter-task communication, and its complex structure makes it hard to use.

In [5], a high-level model of OS called SoCOS is introduced as a high level RTOS model supporting software generation. The main difference between our approach and theirs is that SoCOS requires its own proprietary simulation engine while our RTOS model is build on top of existing system level design luange (SLDL) and can be directly integrated into any system model and design flow supported by the chosen SLDL. Besides, we generate RTOS based dynamic scheduling TLM automatically while the SoCOS based system model is created manually.

3 Design Flow

Figure 1 shows a typical system level design flow [9]. The system design process starts with the specification model written by the designer to specify the desired system functionality. During system design, the specification functionality is partitioned onto multiple processing elements (PEs). The result is a TLM in which each PE executes a specific behavior in parallel with other PEs and communication between PEs takes place through abstract channels. After that, the communication synthesis step generates the bus functional model in which a communication architecture consisting of busses and bus interfaces is synthesized to implement communication between PEs.

Due to the inherently sequential nature of PEs, processes inside the same PE need to be serialized. Depending on the nature of the PE and the data inter-dependencies, processes are scheduled statically or dynamically. In case of dynamic scheduling, a RTOS is required for the final implementation. Usually, the scheduling process takes place after the bus functional model has been generated. In our approach, we move the scheduling into higher level of abstraction, i.e. perform scheduling at TLM level. Since a detailed communication architecture is not required to evaluate scheduling
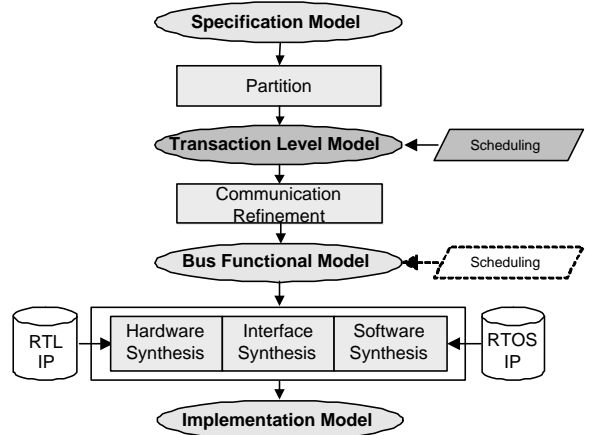

Figure 1. Design flow

results, using a TLM can improve the simulation speed and result in faster design space exploration.

Current definition of TLM is general and ambiguous. Depending on the abstraction of transaction, there are different kind of TLMs. In the higher abstraction level, transaction between the PEs is represented by the message passing channels. On the other hand, abstracting only low level bus protocol primitives (i.e. *send*, *receive*) between the PEs results in a different TLM where the bus drivers are used inside each PE to drive the protocol channels. Note that interrupt handlers are used as part of the bus drivers. Our scheduling refinement tool can be used in both of the TLMs. However, in order to demonstrate the effect of the interrupt scheduling, we use the latter TLM in our example.

In order to validate the scheduling in TLM, a representation of the dynamic scheduling implementation, which is usually handled by a RTOS in the real system, is required. Therefore, a high level model of the underlying RTOS is needed for inclusion into TLMs during system design. The RTOS model provides an abstraction of the key features that define a dynamic scheduling behavior independent of any specific RTOS implementation.

The scheduling refinement tool (Figure 2) refines the unscheduled TLM into a scheduled TLM based on the refinement decisions from the designer. In general, for each PE in the system a RTOS model corresponding to the selected scheduling strategy is imported from the library and instantiated in the PE. Processes inside the PEs are converted into tasks with assigned priorities. Synchronization as part of communication between processes is refined into OS-based task synchronization. In the scheduled output TLM, each PE runs multiple tasks on top of its local RTOS model instance. Therefore, the output model can be validated through simulation or verification to evaluate different dynamic scheduling approaches (e.g. in terms of timing) as
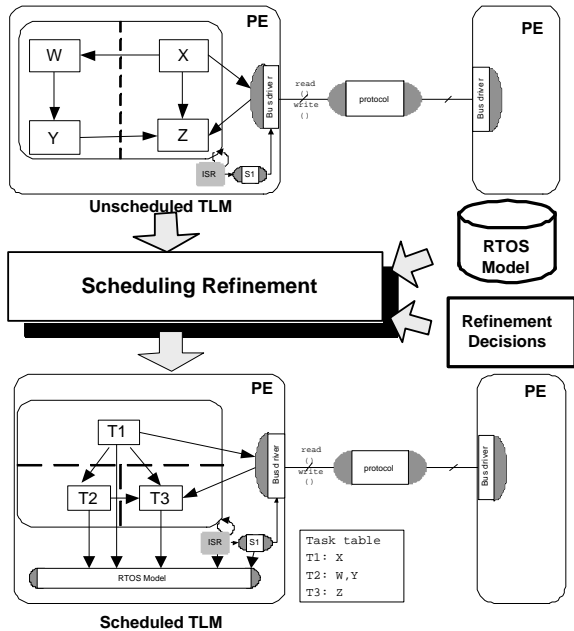
Figure 2. Scheduling refinement tool

```
1   interface RTOS
2   { /* OS management */
3     void init();
4     void start(int sched_alg);
5     /* Task management */
6     Task task_create(const char *name,
7             int type,sim_time period);
8     void task_terminate();
9     void task_sleep();
10    void task_activate(Task t);
11    void task_endcycle();
12    void task_kill(Task t);
13    Task fork();
14    void join(Task t);
15    /* Event handling */
16    Task enter_wait();
17    void wakeup_wait(Task t);
18    /* Delay modeling */
19    void time_wait(sim_time nsec);
20  };
```

Figure 3. Interface of the RTOS model

part of system design space exploration.

As the last step of the design flow, each PE in the bus functional model is then implemented separately. Custom hardware PEs are synthesized into a RTL description. Communication interfaces are synthesized in hardware and software. Finally, embedded software is generated from the scheduled output TLM of the schedule refinement tool. In this process, services of the RTOS model are mapped onto the API of a specific standard or custom RTOS. The code is then compiled into the processor's instruction set and linked against the RTOS libraries to produce the final executable.

4 The RTOS Model

As mentioned previously, the RTOS model is a very import component of the scheduling refinement tool. We implemented the RTOS model on top of the SpecC SLDL [6]. It is incorporated into the RTOS model library of the refinement tool. The library provides RTOS models with different scheduling algorithms typically found in RTOS implementations, e.g. round-robin or priority-based scheduling. In addition, the models are parametrizable in terms of task parameters, preemption, and so on. The detailed information about the RTOS model can be found in [10].

Figure 3 shows the interface of the RTOS model. The RTOS model provides four categories of services: operating system management, task management, event handling, and time modeling.

Operating system management mainly deals with initial-

ization of the RTOS during system start where *init* initializes the relevant kernel data structures while *start* starts the multi-task scheduling.

Task management is the most important function in the RTOS model. It includes various standard routines such as task creation (*task_create*), task termination (*task_terminate*, *task_kill*), and task suspension and activation (*task_sleep*, *task_activate*). Two special routines are introduced to model dynamic task forking and joining: *fork* suspends the calling task and waits for the child tasks to finish after which *join* resumes the calling task's execution. Our RTOS model supports both periodic hard real time tasks with a critical deadline and non-periodic real time tasks with a fixed priority. In modeling of periodic tasks, *task_endcycle* notifies the kernel that a periodic task has finished its execution in the current cycle.

Event handling in the RTOS model sits on top of the basic SLDL synchronization events. Two system calls, *enter_wait* and *wakeup_wait*, are wrapped around each SpecC wait primitive. This allows the RTOS model to update its internal task states (and to reschedule) whenever a task is about to get blocked on and later released from a SpecC event.

During simulation of high level system models, the logical time advances in discrete steps. SLDL primitives (such as waitfor in SpecC) are used to model delays. For the RTOS model, those delay primitives are replaced by *time_wait* calls which model task delays in the RTOS while enabling support for modeling of task preemption.

The RTOS model interface introduced in this section will be later implemented by using the real RTOS APIs during software synthesis. Generally, this means that each routine of the RTOS model interface will be mapped to 1 or *N* target
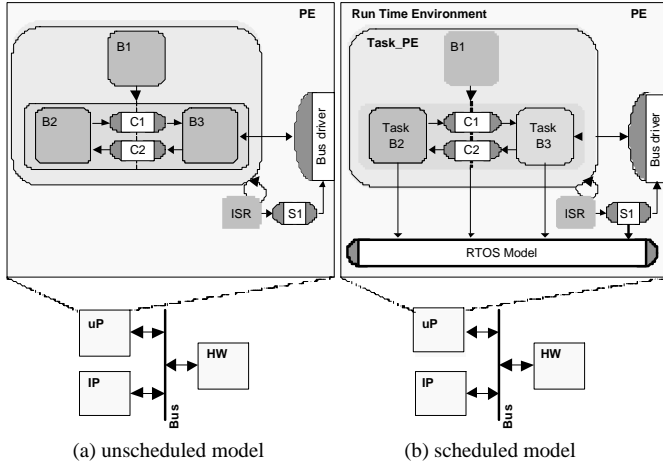
(a) unscheduled model   (b) scheduled model

Figure 4. Refinement example

RTOS APIs.

5 Scheduling Refinement

The scheduling refinement tool refines the input unscheduled model into a RTOS based multi-task model. The refinement process can be divided into four relatively independent steps, namely, *RTOS model instantiation*, *task creation*, *synchronization refinement* and *preemption point creation*, each can be further divided into sub-steps.

In this section, we illustrate the scheduling refinement process through a simple yet typical example (Figure 4). The unscheduled model (Figure 4(a)) executes behavior *B1* followed by the parallel composition of behaviors *B2* and *B3*. Behaviors *B2* and *B3* communicate via two channels *C1* and *C2* while *B3* communicates with other PEs through a bus driver. As part of the bus interface implementation, the interrupt handler *ISR* for external events signals the main bus driver through a semaphore channel *S1*.

5.1 RTOS Model Instantiation

As the first step of the scheduling refinement, a RTOS model implementing `interface` *RTOS* is selected from the RTOS library and a run time environment which coordinates the interaction between the RTOS model and tasks is created for each PE. The run time environment is implemented as a behavior which wrappers around the top-level PE behavior. The RTOS model gets instantiated in the run time environment and the initial values of the internal data structures for the RTOS model are set. At the same time, a main task (*task_PE*) for the PE is created which is the only task available for the RTOS model to schedule at system start time.

Algorithm 1 TaskCreate(IR_{Design}, B_{PE})

1: **for all** Behavior $B \in IR_{Design}$ **do**
2:   **if** IsChildBehavior(B,B_{PE}) **then**
3:     $BInst$ = FindInstance(B);
4:     **while** $BInst \neq NULL$ **do**
5:       **if** IsParallel($BInst,B_{PE}$) **then**
6:         GenTaskFromBehavior($B,BInst$);
7:       **end if**
8:       $BInst$ = FindNextInstance($B,BInst$);
9:     **end while**
10:   **end if**
11: **end for**
12: **for all** Function $F \in IR_{Design}$ **do**
13:   **if** IsMemberFunction(F,B_{PE}) **then**
14:     $Stmnt$ = GetFirstStatement(B);
15:     **while** $Stmnt \neq NULL$ **do**
16:       **if** IsParStatment($Stmnt$) **then**
17:         GenDynamicTasks($Stmnt$);
18:       **end if**
19:       $Stmnt$ = GetNextStatment($B,Stmnt$);
20:     **end while**
21:   **end if**
22: **end for**

5.2 Task Creation

The task creation step converts parallel processes/behaviors in the specification into RTOS-based tasks. This is by far the most important and time consuming part of the scheduling refinement process. The task creation process is shown in Algorithm1. The input to Algorithm1 is the internal representation for the whole design IR_{Design} and the top level behavior for the PE B_{PE}.

Task creation is carried out in a two-step process. In the first step (line 1-11), each behavior instance *BInst* inside B_{PE} are checked to see if they are running in parallel with other behavior instance inside B_{PE}. If such a behavior instance is found (line 5), a task definition for this behavior instance is created (line 6).

In our example, since `behavior` *B2* and *B3* are running in parallel (Figure 4(a)), function *GenTaskFromBehavior* create the task definition *Task_B2* (Figure 5(b)) for `behavior` *B2* (Figure 5(a)). The task is modeled as a `behavior`[2] and there's an method *os_task_create* inserted into to the behavior for construction of the task. Finally, the main body of the task (method *main*) is enclosed in a pair of *task_activate* / *task_terminate* calls so that the RTOS model can control the task activation and termination.

The second step (line 12-22) involves dynamic creation of child tasks in a parent task. The tool goes through each statement of the member functions of `behavior` B_{PE} or

4

Algorithm 2 SyncRefine(*IR*$_{Design}$, *B*$_{PE}$)

1: **for all** Channel $C \in IR_{Design}$ **do**
2:   **if** IsUsedInBehavior(*C*,*B*$_{PE}$) **then**
3:     **for all** Function $F \in C$ **do**
4:       *Stmnt*=GetFirstStatement(*B*);
5:       **while** *Stmnt* $\neq$ *NULL* **do**
6:         **if** IsWaitStatment(*Stmnt*) **then**
7:           RefineWait(*Stmnt*);
8:         **end if**
9:         *Stmnt* = GetNextStatment(*B*,*Stmnt*);
10:       **end while**
11:     **end for**
12:   **end if**
13: **end for**
14: **for all** Function $F \in IR_{Design}$ **do**
15:   **if** IsMemberFunction(*F*,*B*$_{PE}$) **then**
16:     *Stmnt* = GetFirstStatement(*B*);
17:     **while** *Stmnt* $\neq$ *NULL* **do**
18:       **if** IsWaitStatment(*Stmnt*) **then**
19:         RefineWait(*Stmnt*);
20:       **end if**
21:       *Stmnt* = GetNextStatment(*B*,*Stmnt*);
22:     **end while**
23:   **end if**
24: **end for**

```
1  behavior B2()
2  {void main(void)
3   { ...
4    waitfor(BLOCK1_DELAY);/*model delay*/
5    ...
6    waitfor(BLOCK2_DELAY);/*model delay*/
7    ...
8   }
9  };
```

(a) unscheduled model

```
1  behavior task_B2(RTOS os) implements Init
2  {Task h;
3   void init(void) {
4    h = os.task_create("B2", APERIODIC, 0);
5   }
6   void main(void) {
7    os.task_activate(h);
8    ...
9    os.time_wait(BLOCK1_DELAY);/*model delay*/
10   ...
11   os.time_wait(BLOCK2_DELAY);/*model delay*/
12   ...
13   os.task_terminate(h);
14   }
15 };
```

(b) scheduled model

Figure 5. Task modeling

any of it's child behaviors. If a parallel statement (`par` statement in SpecC) is found (line 16), a dynamic task instances are created for this statement (line 17).

This step is illustrated by our example in Figure 6. The `par` statement in the input model (line 9-12 in Figure 6(a)) is converted to dynamically fork and join child tasks as part of the parent's execution (line 6-13 in Figure 6(b)). During this refinement process, the *init* methods of the children are called to create the child tasks (line 6,7 in Figure 6(b)). Then, *fork* is inserted before the *par* statement to suspend the calling parent task by the RTOS model before the children are actually executed in the `par` statement. After the two child tasks finish execution and the `par` exits, *join* is inserted to resume the execution of the parent task by the RTOS model.

5.3 Synchronization Refinement

Replacing SLDL synchronization primitives with RTOS calls is necessary to keep the internal task state of the RTOS model updated. This is achieved by synchronization refinement which wraps event wait primitives in the input model with the RTOS model interface routines *enter_wait* and *wakeup_wait*. The two routines make sure that the RTOS model can intercept event wait primitives thus takes care of task switching.

Algorithm 2 shows how the synchronization refinement works. It is also a two step process: the first step (line 1-13) refines all the `wait` statements inside the channels used in the selected PE while the second step (line 14-23) refines the `wait` statements inside all the member functions of behavior *B*$_{PE}$ and its child behaviors.

Figure 7 shows the synchronization refinement for our example: the `wait` statement inside `channel` *C*1 in the input model (line 10 in Figure 7(a)) is refined into three lines of code in the output model (line 9-11 in Figure 7(b)).

5.4 Preemption Point Creation

In high level system models, simulation time advances in discrete steps based on the granularity of `waitfor` statements used to model delays (e.g. at behavior or basic block level) (line 4,6 in Figure 5(a)). The time-sharing implementation in the RTOS model makes sure that delays of concurrent task are accumulative as required by any model of serialized task execution.

Usually the task switch happens when a task calls the RTOS routine (e.g. `wait` *event*), however, additionally replacing `waitfor` statements with corresponding RTOS time modeling calls is necessary to accurately model preemption. The *time_wait* method (line 9,11 in Figure 5(b)) is

```
 1  behavior B2B3()
 2  {B2 b2();
 3   B3 b3();
 4  void main(void)
 5  {
 6
 7
 8
 9    par
10      { b2.main();
11        b3.main();
12      }
13
14  }
```

(a) before

```
 1  behavior B2B3(RTOS os)
 2  {Task_B2 task_b2(os);
 3   Task_B3 task_b3(os);
 4   void main(void)
 5  {Task t;
 6   task_b2.init();
 7   task_b3.init();
 8   t = os.fork();
 9   par   {
10      b2.main();
11      b3.main();
12   }
13   os.join(t);
14  }
```

(b) after

Figure 6. Task creation

```
 1  channel C1()
 2  {event eRdy;
 3   event eAck;
 4   void send(...)
 5  {
 6    ...
 7    notify eRdy;
 8    ...
 9
10    wait(eAck);
11
12    ...
13    }
14  };
```

(a) before

```
 1  channel C1(RTOS os)
 2  {event eRdy;
 3   event eAck;
 4   void send(...)
 5  { Task t;
 6    ...
 7    notify eRdy;
 8    ...
 9    t = os.enter_wait();
10    wait(eAck);
11    os.wakeup_wait(t);
12    ...
13    }
14  };
```

(b) after

Figure 7. Synchronization refinement

a wrapper around the `waitfor` statement that allows the RTOS kernel to reschedule and switch tasks whenever time increases, i.e. in between regular RTOS system calls. Normally, this would not be an issue since task state changes can not happen outside of RTOS system calls. However, external interrupts can asynchronously trigger task changes in between system calls of the current task in which case proper modeling of preemption is important for the accuracy of the model (e.g. response time results). For example, an interrupt handler can release a semaphore on which a high priority task for processing of the external event is blocked. Note that, given the nature of high level models, the accuracy of preemption results is limited by the granularity of task delay models.

5.5 Scheduling Refinement Example

Figure 8 illustrates the simulation result of the output model generated from our refinement tool for the example

from Figure 4. Figure 8(a) shows the simulation trace of the unscheduled model. Behaviors *B2* and *B3* are executing truly in parallel, i.e. their simulated delays overlap.

After executing for time d_1, *B3* waits until it receives a message from *B2* through the channel *c1*. Then it continues executing for time d_2 and waits for data from another PE. *B2* continues for time $(d_6 + d_7)$ and then waits for data from *B3*. At time t_4, an interrupt happens and *B3* receives its data through the bus driver. *B3* executes until it finishes. At time t_5, *B3* sends a message to *B2* through the channel *c2* which wakes up *B2* and both behaviors continue until they finish execution.

Figure 8(b) shows the simulation result of the scheduled model for a priority based scheduling. It demonstrates that in the refined model *task_B2* and *task_B3* execute in an interleaved way. Since *task_B3* has the higher priority, it executes unless it is blocked on receiving or sending a message from/to *task_B2* (t_1 through t_2 and t_5 through t_6), waiting for an interrupt (t_3 through t_4), or it finishes (t_7) at which points execution switches to *task_B2*. Note that at time t_4, the interrupt wakes up *task_B3* and *task_B2* is preempted by *task_B3*. However, the actual task switch is delayed until the end of the discrete time step d_6 in *task_B2* based on the granularity of the task's delay model. In summary, as required by priority based dynamic scheduling, at any time only one task, the ready task with the highest priority, is executing.

6 Experimental Results

We used the scheduling refinement tool in the design of a voice codec for mobile phone applications.The Vocoder contains two tasks for encoding and decoding in software, assisted by a custom hardware co-processor. For the implementation, the Vocoder was compiled into assembly code for the Motorola DSP56600 processor and linked against a small custom RTOS kernel that uses a scheduling algorithm where the decoder has higher priority than the encoder[8].

Table 1 shows the results for the vocoder model. The Vocoder models were exercised by a testbench that feeds a stream of 163 speech frames corresponding to 3.26 s of speech into encoder and decoder. Furthermore, the models were annotated to deliver feedback about the number of context switches and the transcoding delay encountered during simulation. The transcoding delay is the latency when running encoder and decoder in back-to-back mode and is related to response time in switching between encoding and decoding tasks.

Experimental results show that the simulation overhead introduced by the scheduling refinement tool is negligible while providing accurate results. As explained by the fact that both tasks alternate with every time slice, round-robin scheduling causes by far the largest number of context switches while providing the lowest response times.
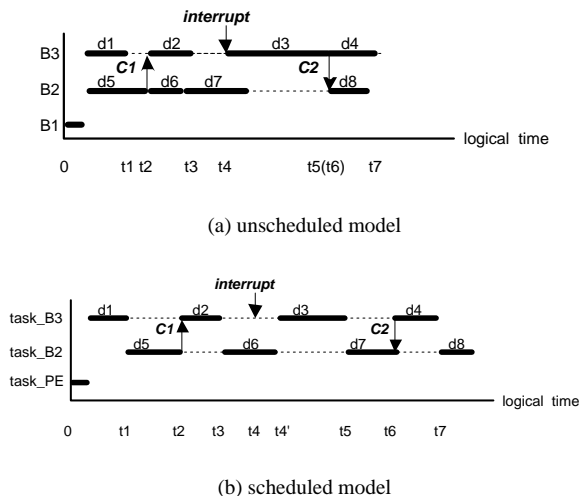
(a) unscheduled model

(b) scheduled model

Figure 8. Simulation trace for model example.

	Lines of code	Sim. time	Context switches	Transcoding delay
Unsched.	11,313	27.3s	0	9.7ms
Roundrobin	13,343	28.6s	3262	10.29ms
Encod>decod	13,356	28.9s	980	11.34ms
Decod>encod	13,356	28.5s	327	10.30ms
Impl.	79,096	5h	327	11.7ms

Table 1. Vocoder experimental results.

Note that context switch delays in the RTOS were not modeled in this example, i.e. the large number of context switches would introduce additional delays that would offset the slight response time advantage of round-robin scheduling in a final implementation. The simulation result shows that in priority-based scheduling, it is of advantage to give the decoder the higher relative priority. Since the encoder execution time dominates the decoder execution time this is equivalent to a shortest-job-first scheduling which minimizes wait times and hence overall response time. Furthermore, the number of context switches is lower since the RTOS does not have to switch back and forth between encoder and decoder whenever the encoder waits for results from the hardware co-processor. Therefore, priority-based scheduling with a high-priority decoder was chosen for the final implementation. Note that the final delay in the implementation is higher due to inaccuracies of execution time estimates in the high-level model. In summary, compared to the huge complexity required for the implementation model, the scheduling refinement tool enables early and efficient evaluation of dynamic scheduling implementations.

7 Summary and Conclusions

In this paper, we presented a RTOS model and the refinement steps for transforming an unscheduled TLM into TLM with RTOS scheduling support. In the design flow, our contribution is primarily the automation of the scheduling refinement process that facilitates rapid evaluation of scheduling algorithms in the early stage of system design using TLM. We developed a tool to automate the refinement process. Experiments are performed to show the usefulness of the tool in system design. Currently the tool is written for SpecC SLDL because of its simplicity. However, the concepts can be applied to any SLDL (SystemC, Superlog) with support for event handling and modeling of time.

Future work includes the development of tools for software synthesis from the scheduled TLM down to target-specific application code linked against the target RTOS libraries.

References

[1] QNX. Available: http://www.qnx.com/.

[2] SpecC. Available: http://www.specc.org/.

[3] VxWorks. Available: http://www.vxworks.com/.

[4] J. Cortadella. Task generation and compile time scheduling for mixed data-control embedded software. In *Proceedings of the IEEE Design Automation Conference*, June 2000.

[5] D. Desmet et al. Operating system based software generation for system-on-chip. In *Proceedings of the IEEE Design Automation Conference*, June 2000.

[6] R. Dömer, A. Gerstlauer, and D. Gajski. Specc language reference manual. In *SpecC Technology Open Consortium*, 2002.

[7] L. Gauthier et al. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Trans. on CAD*, November 2001.

[8] A. Gerstlauer et al. Design of a GSM Vocoder using SpecC Methodology. Technical Report ICS-TR-99-11, University of California, Irvine, Feburary 1999.

[9] A. Gerstlauer and D. Gajski. System-level abstraction semantics. In *International Symposium on System Synthesis*, October 2002.

[10] A. Gerstlauer, H. Yu, and D. Gajski. Rtos modeling for system level design. In *Proceedings of Design,Automation and Test in Europe*, March 2003.

[11] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[12] H. Tomiyama et al. Modeling fixed-priority preemptive multi-task systems in SpecC. In *SASIMI*, October 2001.