Timed RTOS Modeling for Embedded System Design

Zhengting He
Dept. of Elec. & Comp. Engr.
Univ. of Texas at Austin
Austin, Texas 78712, U.S.A
zhe@ece.utexas.edu

Aloysius Mok
Dept. of Computer Science
Univ. of Texas at Austin
Austin, Texas 78712, U.S.A
mok@cs.utexas.edu

Cheng Peng
DSP Catalog
Texas Instruments
Stafford, Texas 77477, U.S.A
c-peng2@ti.com

Abstract

With processor speed doubling every 18 months, more and more system functionalities are implemented as software (SW) in the design process of embedded systems. Selecting the "right" RTOS before the SW is developed is very important. In this paper, we present an RTOS modeling tool based on SystemC [2]. It is configurable to support modeling and timed simulation of most popular embedded RTOSes. Timing fidelity is achieved by using delay annotation. The OS timing information is derived from published benchmark data.

Experiments show that the accuracy of our approach is able to help designers gain confidence in their RTOS selection. By avoiding using an instruction set simulator, the simulation can be speeded up by more than 3 orders of magnitude. Any other component integrable with SystemC can also be integrated in our simulation environment.

1. Introduction

With the speed of integrated circuit (IC) doubling every 18 months, programmable high performance microcontrollers and DSPs have appeared. System functionalities that used to be realizable by hardware (HW) can now be implemented by software (SW). Compared to HW implementation, SW implementation is more attractive because of the flexibility afforded by programmability and because of the expectation of lower development cost. On the other hand, with the increasing complexity of embedded devices, more and more functionalities need to be integrated into a single system. As a result, it is common to see in the modern embedded system several SW programs running concurrently on a processor managed by a real-time operating system (RTOS).

Embedded SW typically has real-time constraints to satisfy in addition to functionality requirements. It is important to be able to validate all these properties together as early as possible in the design cycle, and in the context of running the embedded SW on top of an RTOS. Traditionally, designers perform design space exploration (DSE) by manually porting the application SW to the target architecture which consumes a lot of time and is often error-prone [1]. An alternative is RTOS modeling. RTOS modeling is a technique that tries to model existing RTOSs within a single environment and captures the abstracted RTOS behaviors at the system level. Ideally, RTOS modeling should fulfill the following requirements:

1. The simulation of the model should be fast enough. It also implies that it should model the target RTOS at a high level of abstraction than, say, using an instruction simulator (ISS) which is not only too slow for simulating a meaningful amount of work, but may also not available at the early design stage.
2. The simulation result of the model should be reasonably accurate for design space exploration (DSE).
3. The model can be easily integrated with other components under the same simulation framework.
4. The model is configurable and easy to use. Different candidate RTOSes can be simulated with little or no change to the application programs.

In this paper, we present an RTOS modeling tool based on SystemC [2]. With our tool, an RTOS model is easily implementable by employing the SystemC thread primitives (SC_THREAD) and simulation engine. Other simulation components written by C/C++ or VHDL language can also be integrated into the SystemC framework to support HW/SW co-simulation. Our model is configurable to support modeling and timed simulation of most existing embedded RTOSes, i.e., Linux, DSP/BIOS etc. To model a specific RTOS, a user only needs to: (1) set a few parameters which will be used to configure the RTOS state machine, (2) provide some OS timing information such as scheduler execution delay, context switch delay etc., so that delay annotation can be automatically generated into the model, and (3) "plug in" necessary peripheral driver module(s).

Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)
1080-1812/05 $ 20.00 IEEE

We believe that timed OS simulation by delay annotation is an appropriate solution for DSE at the early design phase compared to the traditional approach of using an ISS since the former is much faster and able to achieve reasonable timing fidelity. In [10], it was reported that most ISSs simulate one target instruction by every 50-100 host instructions. To make it worse, most ISSs do not have RTOS support. Another drawback of ISS is that it is not able to provide accurate timing estimation to application programs at the early design stage since in most cases application programs have not been implemented or optimized at that time.

The OS timing information can be measured by experiment [3], or even estimated by software estimation techniques [15] [16] [18] [19] if the OS source code is available. In our model, we demonstrate a simple yet effective approach to derive the information from benchmark data provided by the OS vendor. The benchmark results are readily available and quite accurate. Application program timing can adopt a similar strategy and can be isolated from OS timing so that changes to either one will not affect the other unless the HW architecture (processor) is changed.

One of the problems associated with delay annotation is that the simulation clock advances in macro steps. As a result, the OS being modeled may advance past a time-stamp at which it is supposed to handle an input event (interrupt). Bounding the size of the macro clock step to a fixed small value and checking input event more frequently [1] does not completely solve the problem. A more adaptive solution is needed. The need for adaptiveness in choosing step size is particularly important for RTOS simulation since one of its important goals is to identify the interrupt response latency. Without solving the step size problem, it is not possible to decide whether a delay is truly caused by the modeled OS or because of a large clock step advance. To the best of our knowledge, this problem has not been addressed comprehensively so far. In this paper, we solve the problem by splitting the clock step δ if the current simulation clock $C + \delta > T_{in}$, which is the time stamp when the next input event will happen; the process that advances the clock will only be allowed to proceed to T_{in} and then blocks. By the time the input event is handled and the process is resumed, the remaining amount $(C+\delta)-T_{in}$ will be added to the simulation clock in a similar manner. Details are given in section4.

HW/SW co-simulation often suffers from high synchronization overhead, which gets more severe for co-simulation of communication-intensive or interrupt-based systems where interrupt is used as the communication protocol between modules [4]. Research on synchronization overhead reduction can be classified into three categories: (1) optimistic approaches, (2) conservative approaches, and (3) optimized conservative approaches. In our work, we adopt the optimized conservative approach where each sender module informs the receiver beforehand when it will send an event. The receiver cannot advance its clock over this time-stamp. A lot of research has been done to estimate the time stamp of the next output event [16] [17]. However, most of them are based on the unrealistic assumption that a SW process executes on dedicated HW which is not applicable to the general case of multiple processes running concurrently on a processor. [5] presented some results on priority-driven multitasking system. However, their work is subject to two assumptions: 1) the task priority is static; and 2) the scheduler is called only when the timer interrupt occurs. In general, an RTOS performs rescheduling whenever necessary and not only at timer interrupts. In this paper, we derive an algorithm to estimate the time stamp of the next OS-wide output event for any RTOS with a static/dynamic priority driven scheduler.

The rest of the paper is organized as follows. Section 2 gives a survey to the related work. The overall OS modeling framework is described in section 3. Section 4 explains how to derive the timing information and delay annotation interface. Section 5 describes the synchronization protocol. Some experimental results are shown in section 6. Section 7 summarizes the paper.

2. Related Work

Research on RTOS modeling and simulation can be categorized into the following three directions:

- *System Call Translation*, where any system calls from the application being simulated are re-directed into the host OS and executed. This approach is often used by those ISSs without OS simulation ability to verify the functionality of application SW [10] and provides limited timing information. Clearly it is not able to assist in RTOS selection.
- *Native OS*, which compiles the target OS code and executes it on the simulation host. For example, WindRiver Systems provides VxSim as a simulation model for its RTOS [7]. It is also able to verify the functionality of application SW. One of its drawbacks is that it does not support modeling of the HW on which the target RTOS runs, and thus the timing of target RTOS is not accurate. Another drawback is that modeling other RTOSes becomes impossible.
- *Virtual OS*, which simulates the functionality and timing of a real OS. Ideally the virtual OS should be configurable to model any existing RTOSes. OS timing can either be achieved by delay annotations inserted into the virtual OS source code, or calculated by an aggregate timing model. For instance, the scheduling delay can be calculated as a function of the number

of ready tasks. Commercial tools such as SoCOS [8] and CarbonKernel [9] are available. The main problem of these tools is that it is not easy to use. To model a specific OS, a user often needs to manually "personalize" the virtual OS by inserting appropriate delay annotations. Gerstlauer in [11] presented an RTOS model written on top of SpecC. The primitives in SepC are used so that the implementation is simple. The application code can be synthesized and compiled to run on the target RTOS by replacing system calls to the OS model with those to the target RTOS. However, the timing accuracy is not satisfactory. [6] presented an RTOS model on top of SystemC. Its focus is modeling the scheduling algorithm and the process communication mechanism.

Yoo in [1] proposed an idea of automatic generation of OS model. The designer can choose a set of OS services from a library. The service code will be compiled and linked with a micro kernel to generate the RTOS that is executable on the target processor. Delay annotations are automatically inserted in the target OS source code. The main problem associated with this approach is that it can only generate an RTOS from the library written by the author, but not able to model and simulate any other commercial RTOSes.

Our approach is similar to the virtual OS concept. Compared to [8] [9], out tool is easier to use since delay annotations can be inserted to appropriate places automatically. Compared to [6], our tool requires little changes to the application simulation code other than the delay annotation insertion. To model a specific RTOS, the user only needs to set a few parameters for OS state machine generation, and provide OS timing information which is often readily available from the OS and IC vendors. In [6] the application simulation code has to explicitly wait for messages from the OS modeler which makes the simulation code significantly different from the real code. Compared to [11], our model is also easy to implement by using SystemC simulation engine and its thread primitives. The timing information derived from benchmark results is accurate enough to assist in the selection of an RTOS for design space exploration (DSE) at an early design phase.

3. RTOS Modeling

3.1. RTOS State Machine

A generic RTOS state machine shown in Fig. 1 is implemented on top of SystemC. The state machine can be configured to model different RTOSes.

There are seven states in the RTOS state machine, as explained below.

1. *power-off*: the processor has not started running. Before powering on an RTOS, the user needs to connect necessary peripherals such as the timer, to the OS, and also specify a set of parameters to configure the specific state machine for the RTOS being modeled.
2. *curr_run_in_user_mode*: the processor is executing a task in the user mode. In case the professor doesn't differentiate the kernel and user space, the state simply means that a task is executing application code before making a system call into the kernel.
3. *curr_run_in_kernel_mode*: the processor is executing a task in the kernel mode.
4. *sync_for_in_intr*: an RTOS blocks to wait for an external event (interrupt). This is a state that does not exist when the RTOS executes on the real HW, but only for simulation synchronization purpose. For optimized conservative simulation, a module needs to enter this state when it reaches a clock value when an external event may come.
5. *handle_intr*: the RTOS is handling an input event.
6. *schedule*: the RTOS scheduler is selecting a ready task.
7. *idle*: there is no ready task in the RTOS.

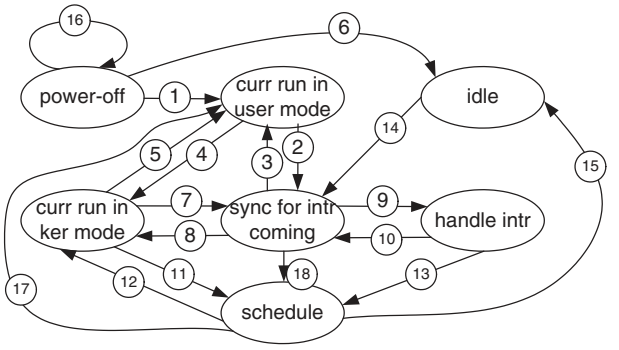

Figure 1. Basic RTOS State Machine

Transitions (3) and (8) represent the case where the OS was waiting to receive an event (interrupt) but it did not actually arrive. Note that in practice it is hard to predict the exact time-stamp of the next event because of the dynamic behavior of SW. In our model, an event sender tries to predict and notify the receiver the earliest possible timestamp when it will issue an event. When the receiver reaches that time, it waits for the event. If it is sure that the event will not occur because the previous prediction is too conservative, it exits from the synchronization state and continues. The details on the synchronization protocol will be explained in section 5.

To model a specific RTOS, a set of parameters need to be set to configure the state machine. The main parameters are: 1) scheduling policy, 2) kernel preemption points

(when to a re-scheduling decision), 3) whether or not to support thread, 4) whether or not to support interrupt thread, 5) inter-process communication (IPC) mechanism, and 6) whether or not to have memory protection etc.

For example, the embedded Linux implements threads in the kernel and uses threads instead of processes as scheduling entities. It distinguishes three classes of threads: real-time FIFO threads which have highest priority and are not preemptable, real-time round robin threads which have the same priority as real-time FIFO threads but are preemptable, and timesharing threads which have the lowest priority and are scheduled by an priority aging policy [12]. Interrupt thread is not supported. Interrupt is handled by jumping to the interrupt service routine (ISR). A re-scheduling decision is not made after returning from an ISR unless it is a timer interrupt. Other cases where re-scheduling happens are: current thread blocks; and current thread forks a child thread which will block the parent thread and starts itself. Linux also supports a rich set of IPC mechanisms such as signals, System V IPC, Unix domain sockets etc.

Take Texas Instruments' (TI) RTOS DSP/BIOS as another example [13]. It is a single address space OS without memory protection. Processes and threads are not differentiated. It has three classes of threads: HW interrupt threads which have the highest priority and are not preemptable, SW interrupt threads which have lower priority than HW interrupt threads and are preemptable, and task threads which have the lowest priority and are also preemptable. The difference between SW interrupt threads and task threads is that SW interrupt threads run on a common stack while each task thread has its own stack. The scheduling policy is priority driven without aging. Since interrupt service thread (IST) is supported, an ISR only does a minimum amount of work, and then resumes the corresponding IST to make it do the rest work. After returning from an ISR, a re-scheduling decision is always made so that an IST can be started immediately. A re-scheduling is also made when any thread with a higher priority than the current one is resumed. For instance, if the current thread releases a lock, which resumes a thread with a higher priority, a re-scheduling decision is made. The IPCs supported by DSP/BIOS are memory sharing, pipe and mailbox.

3.2. OS Sub-Module Design

We adopted the object oriented approach to design the following components in our OS modeling tool: 1) task management, 2) OS kernel, 3) IO module, 4) IPC, 5) timing, and 6) event handling and synchronization protocol. 5) and 6) will be explained in section 5.

3.2.1. Task Management The task management interface consists of standard routines for

- task creation, i.e. *task_create()*,
- task termination, i.e. *task_exit()* and *task_kill()*,
- task suspension, i.e. *task_yield()* and *task_sleep()*,
- task activation, i.e. *task_resume()*.
- task priority change, i.e. *task_set_prio()*.

Each task is implemented as a SystemC thread (SC_THREAD) with a sc_event object so that context switching can be easily implemented by calling its sc_event methods: *wait()* and *notify()*. To work around the limitation that the SystemC simulation engine does not support dynamic thread creation, a static thread pool is created before the OS is powered on. These threads block on their own sc_event at the beginning. When a *task_create()* call is made, a thread object is retrieved from the pool and a function pointer to the actual thread function is saved in the task object. By the time the thread is scheduled to execute, it unblocks from its sc_event and calls the thread function to start.

If the modeled OS has memory protection mechanism, only threads belonging to the same process can share the resource. Otherwise, threads and processes are not differentiated.

3.2.2. OS Kernel The kernel consists mainly of the task scheduler and synchronization objects. We have implemented a flexible priority driven scheduler which supports a mix of preemptive/nonpreemptive scheduling policies with/without priority aging. When a task is created, three fields in the thread object are specified: $priority$, $aging$, and $preemptible$. The scheduler schedules the tasks based on these fields. For example, to create a DSP/BIOS HW IST, a user can set $priority = 0$ which is the highest priority, $aging = no$ and $preemptible = no$. Associated with each priority is a task queue. Scheduling of the tasks with the same priority can either be FIFO or Round Robin, depending on the configuration specified by the user.

Right now the only synchronization object supported by our tool is the semaphore. Other synchronization objects such as mutex and lock can be derived from semaphore. A semaphore has the following interface to user:

- *sem_init(int count)*, which creates a semaphore with the specified count.
- *sem_inc()*, which increases the count. A blocking thread may be resumed if $count > 0$. Depending on the configuration specified, the order of the threads being resumed can be FIFO or priority based.
- *sem_dec()*, which decreases the count. The calling thread may be blocked if $count \leq 0$.
- *sem_kill()*, which destroys the semaphore object.

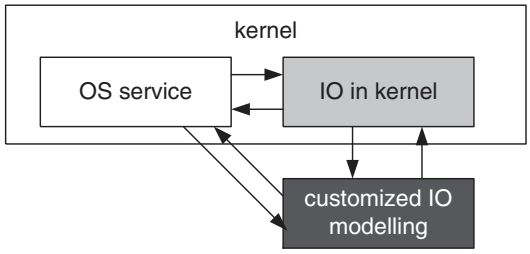

Figure 2. IO Module Layer

Another interface in kernel module is the *power_on()* method which initializes and starts the OS.

3.2.3. IO Module Modeling IO is a difficult problem because different devices can work in various ways, and each OS has its own IO structure and handles IO in its own way. Modeling individual IO device is not our concern. Instead, our focus is on creating a flexible IO framework so that (1) device models and drivers can be easily plugged in without major modifications, and (2) different IO structures and IO handling mechanisms from different OSes can be easily modeled and simulated in our framework.

The overall IO module has two layers as shown in Fig. 2. The black block represents the models and drivers of each individual device. The grey block residing in the kernel is the IO interface between applications and device drivers. Driver can request OS service via the IO layer in kernel, i.e., blocking the calling thread, or requests the service directly. A typical example of the former case is the DSP/BIOS whose IO manager (IOM) resides in the kernel and manages the state of all tasks requesting IO. Linux is an example representing the latter case where drivers manage the state of the calling thread by themselves.

The IO interface to application program are: *io_open*, *io_close*, *io_read*, *io_write*, and *io_ctrl*.

Each device in the system has a *StructDev* struct as shown in Fig. 3. It is shared by both layers. *wQueue* and *rQueue* (line 5-6) are queues saving the pending write/read requests. *drvBuf* (line 8) is the buffer allocated by kernel to the device. *drvStruct* (line 10) is a pointer to specific driver information. *kernelCallback* (line 13) is a function callable by the driver to inform the kernel that an IO request has been finished. It is used by DSP/BIOS. Line 14 - 24 are pointers to the corresponding driver functions to which the IO layer in kernel passes all the requests from applications. The interface between the driver and the kernel is clean but flexible. To model a specific device, the designer only needs to implement these functions in the desired abstract level. *HWIThread* and *SWIThread* are pointers to interrupt threads if they are used.

Different drivers often have different request formats. Simulating the exact request format for each device is not only unnecessary, but also inconvenient for OS modeling. We provide a unified request format *IoStruct* as shown in Fig. 4. It is broad enough to fulfill most drivers' requirement. When sending a request to a specific device, the necessary information from IoStruct is retrieve for the driver. When the request is completed, the return information from the driver is placed in IoStruct.

```
1   struct DevStruct {
2     char name[20];
3     int mode; // device open mode
4     int state; // state of the device
5     Queue *wQueue; // queue of pending write requests
6     Queue *rQueue; // queue of pending read requests
7     Sem *sem; // semaphore of the device
8     char *drvBuf; // driver buffer and length
9     int drvBufLen;
10    void *drvStruct; // customized driver structure
11    OS *os; // pointer to os
12    PE *device; // pointer to device communicate with
13    FuncPtr kernelCallback; // driver to kernel callback
14    int intrId; // interrupt line used
15    Interrupt_Data *intrData; // interrupt data
16    /* pointers to driver functions */
17    int (*init)( struct DevStruct *dev );
18    int (*open)( DevStruct *dev );
19    int (*close)( DevStruct *dev );
20    void (*write) ( DevStruct *dev, IoStruct *ios );
21    void (*read)( DevStruct *dev, IoStruct *ios );
22    void (*io_ctrl)( DevStruct *dev, IoStruct *ios);
23    void (*destroy)( DevStruct *dev );
24    void (*isr)( DevStruct *dev );
25    Task* HWIThread; // HW interrupt thread
26    Task* SWIThread; // SW interrupt thread
27  }
```

Figure 3. Device Struct

Line 4 - 9 is the buffer information for the request. For an OS with virtual memory system, an IO request is typically associated with both the buffer in user space and kernel space. *cmd* (line 11) is the command of the request. *commID* (line 12) is used to by multiple components sharing the same communication link to identify each other. For instance, if two processes communicate to each other via ethernet, it can be used to save the IP address.

Our IO module supports three types of IO operation mode: (1) synchronous blocking, (2) synchronous non-blocking, and (3) asynchronous. The first two modes are common and supported by most OSs. The third one is special yet proved to be efficient both by DSP/BIOS and Linux 2.6. In mode (3) when an application process issues an IO request, it also provides an application callback pointer (line 13 in Fig. 4). If the request cannot be completed immediately, it will be queued to the driver and the application thread can continue doing other work. By

```
1   struct DevStruct {
2     Task *task; // pointer to calling thread
3     DevStruct *dev; // pointer to the device
4     char *uBuf; // user buffer and length
5     int uBufLen;
6     int uDataLen; // data length in user buf
7     char* kBuf; // kernel buf and length
8     int kBufLen;
9     int kDataLen; // data length in kernel buf
10    int ioMode;
11    int cmd; // request command
12    ID commID; // communication id
13    FuncPtr appCallback; // kernel to app. callback
14  }
```

Figure 4. IoStruct

the time the request is completed, the driver thread, i.e., HW IST, calls back to kernel which in turn calls the application callback to notify that the request is completed. Inside the application callback, a new request can be issued.

3.2.4. IPC A rich set of IPCs exist in today's OSes. Modeling each IPC type in its exact form is not only tedious, but also unnecessary. Instead, we abstract them into the following three classes which are able to cover most IPCs.

- *Memory sharing*, which is the most widely used IPC form in embedded system since most embedded systems have limited memory and a single address space. Memory sharing is easy to implement and also saves the expensive memory resource.
- *Message copying*, which is also a commonly used IPC. A message queue protected by a semaphore is created.
- *Signal*, which is used by Linux based embedded systems to quickly inform a thread to take the desired action. The default actions supported by us are THREAD_KILL, and THREAD_BLOCK.

4. Simulation Timing

Interface *inc_clock(δ)* is provided to allow the OS clock advance δ units in discrete steps. Fig. 5 shows how the simulation clock is incremented without going beyond the time-stamp when the next interrupt will occur. If δ is too large, the clock is only advanced to intrArrTime (line 4). The OS will handle the interrupt if it actually shows up. A re-scheduling decision may be made after handling the interrupt, which may preempt the thread calling inc_clock(). When the thread is resumed later, the remaining amount of δ will be accumulated to the OS clock similarly.

Delay annotations are inserted in application and OS code to call *inc_clock()*. Our approach has two advantages compared to the work in [1] [11]. Firstly, simulation can accurately measure interrupt response latency caused by the OS since interrupts are handled at the exact time when they are supposed to be. Secondly, the number of delay annotations can be reduced to alleviate user's work. In [1] [11] since the timing accuracy is directly affected by the clock increase steps and delay annotation frequency, user often needs to manually insert delay annotations in the source code as much as possible to achieve the desired accuracy.

The timing information can be obtained in various ways. A common strategy is to perform compiler analysis on the source code. [16] and [18] compile the source code to java byte code, and an estimation is made from the java byte code based on the specific features of the target processor. [15] and [19] directly compile the source program to the target instructions to do the timing analysis. Such techniques are applicable to application timing estimation but not to OS modeling since the OS source code is often not available. Wang in [3] proposed an experimental method to measure the OS timing information, i.e., context switch overhead, timer jitter, scheduling cost etc. Such an approach is not applicable unless both the OS binary and the target HW are available. In this paper, we propose an idea to derive the OS timing information from benchmark results. The calculation is simple, and the benchmarks are readily available from OS vendors and/or research publications. The timing accuracy is determined by the benchmark data. Since most OS kernels are expected to execute in on-chip memory of embedded processors and directly interface the HW, compared to application programs, the benchmark can be trusted.

Take DSP/BIOS as an example; TI has measured the following timing benchmark on its DSP processors:

- *HW interrupt*, which includes the interrupt latency, interrupt enable/disable costs, interrupt prolog/epilog costs etc.
- *SW interrupt*, which includes SW interrupt enable/disable costs, and the cost of resuming a SW interrupt thread.
- *Task*, which includes all kinds of task management costs, i.e., creating a task with/with context switching.
- *Sem* and *Lock*, which includes the costs of operations on semaphores and locks.
- *Memory*, which includes the costs of dynamic memory allocation and de-allocation.
- *Pipe* and *Mailbox*, which are the costs of IPC operations.
- *Queue*, which are the costs of all kinds of queue operations.

Other information can be derived from the benchmark results. For instance, suppose it has been measured on the

```
1   os.inc_clock( δ) {
2     while( clock+δ > intrArrTime ) {
3       δ− = intrArrTime-clock;
4       clock = intrArrTime;
5       if( intr_arrive() )
6         handle_intr(); // thread may be preempted
7     }
8     clock += δ;
9   }
```

Figure 5. Clock Advance

TMS320C64X DSP (on which the I-cache is disabled and the kernel executes in on-chip memory) that creating a task with and without context switch takes 840 and 744 cycles, respectively, then the context switch cost can be calculated as 96 cycles. For another instance, it has also been measured that it takes 752 cycles from the time when an interrupt occurs to the time after OS handles the interrupt and resumes a new task to run. The 752 cycles consists of interrupt latency, interrupt prolog and epilog, ISR execution, scheduler execution, and task context switch. If the ISR does nothing but increases a semaphore count, and the cost of all the other parts are known except for the scheduler execution, the scheduler execution cost can also be calculated.

5. Simulation Synchronization

5.1. Event Time-Stamp Prediction

HW/SW co-simulation of interrupt-based systems often suffers from high synchronization overhead. In the worst case when the next interrupt will occur is unknown, simulation has to be carried out in a lock-step manner. Optimized conservative simulation tries to solve this problem by predicting the time-stamp of the next event. During the simulation, each module predicts the time stamp t when it will send out an event and informs the receiver. The receiver makes sure that it will not run beyond t before waiting for the event. A rich set of research results on predicting the event time-stamps have been published [16] [17]. Owing to the dynamic behavior of the SW, event time prediction of SW programs is often specified by an interval (t_{min}, t_{max}). The main problem associated with the previous work is that they are all based on the unrealistic assumption that a SW process executes on its dedicated HW. Unfortunately, in most of today's embedded systems this is not the case. When multiple processes execute concurrently on a processor, the existing research results are still applicable but they are not effective for reducing the synchronization overhead. This issue is illustrated by the following example.

Example 1 *Suppose process C_1 and C_2 execute on two different processors P_1 and P_2, respectively. If P_1 and P_2 were not shared by any other process, C_1 would send an interrupt to C_2 at clock = 35, and C_2 would send an interrupt to C_1 at clock = 40. Due to the dynamic behavior of SW processes, we assume that we can only predict that C_1 would send the interrupt at either clock = 25 or 35, and C_1 would send the interrupt at either clock = 30 or 40.*

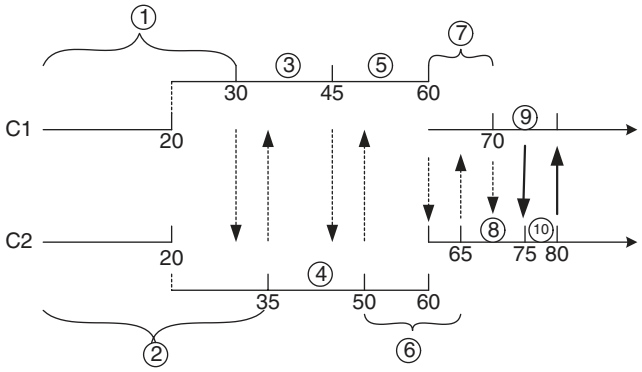

Figure 6. Synchronization Overhead

Now also assuming both processors are shared by other processes, and both C_1 and C_2 are preempted at 20 and resumed at 60, the problem is illustrated in Fig. 6.

1. *C_1 is preempted at 20 on P_1. The OS on P_1 stops to wait at 30 since C_2 on P_2 may send an interrupt to it at that time.*
2. *C_2 executes and is also preempted at 20. The OS on P_2 has to stop at 35 because when C_1 will be resumed is unknown, and a safe way is to assume C_1 may resume immediately at 30 on P_1 which makes it possibly send an interrupt at 35 after executing for another 5 cycles.*
3. *P_1 executes other processes until clock = 45 due to the same reason as explained in 2).*
4. *P_2 advances until clock = 50 due to the same reason as 2)*
5. *P_1 advances to 60 due to the same reason as 2). When it reaches time 60 and C_1 is resumed, it becomes known that the nearest time C_1 may send an interrupt is at 65.*
6. *P_2 advances to 60 and resumes C_2. Then C_2 executes until reaching 65. It becomes known that the nearest time C_2 may send an interrupt is at 70.*
7. *C_1 runs to 65 and it becomes known that C_1 is taking a longer path and will send an interrupt at 75. P_1 continues executing C_1 until reaching 70.*
8. *Similarly, P_2 executes C_2 until 75. It becomes known that C_2 will send the interrupt at 80.*
9. *C_1 runs to 75 and sends the interrupt. It continues until reaching 80.*
10. *C_2 receives the interrupt at 75. Then it advances to 80 and sends its interrupt to C_1*

As we can see, the synchronization operations during interval [20, 60] are due to the fact that how long C_1 and C_2

will be preempted is unknown. If such preemption intervals are also predictable, those synchronization actions are not necessary.

We have designed an algorithm to predict the time-stamp of the next OS-wide event for the more complicated but realistic case where multiple processes execute concurrently on a processor, managed by an RTOS. We assume that the RTOS scheduling policy is priority driven, which is true for most RTOSes. Processes can be preemptive or non-preemptive. Application programs are required to report two time predictions to the kernel: *nextEvt* which is the nearest time when it will issue a system call to cause an output event, and *nextPause* which is the nearest time when it will issue a system call which may make it block, i.e., a blocking I/O call, a task_sleep() call etc. Those predictions are made by analyzing the application program, assuming it uses the processor exclusively. The predicted results are relative values, i.e., if the clock is C when the prediction is made, and *nextEvt* = $\triangle$, then the nearest time when the application sends out an event is $C + \triangle$.

We provide an interface *reg_prediction(nextEvt, nextPause)* to allow insert predictions dynamically in application programs. The user needs to identify all the paths leading to system calls which either send out events or block the caller. In case the system call is a blocking I/O call, *nextEvt* will be the same as *nextPause*. With the help of some advanced research compiler that can performs the analysis automatically [20], we believe that the speedup gain in the simulation is worth the amount of additional work.

Owing to the page limitation, we only show the prediction algorithm in Fig. 7 for the case where the OS scheduling policy is a preemptive priority scheduler and each priority level has at most one process. Other cases can be derived similarly.

For a process (tsk) with priority i, *preemptInterval* is the time interval when it may be preempted by any higher priority threads each of which will execute for at least its predicted nextPause amount of time (line 9). Thus, the earliest time that it can send out an event is $clock + preemptInterval + tsk.nextEvt$ (line 8). The time-stamp of the next OS-wide event is the smallest among them. Suppose it is calculated that the next event will be sent out by a process T with priority i, the prediction needs to be updated whenever any of the following condition becomes true:

- Any process with a priority higher than i is created, resumed, or killed since the last prediction is made.
- Process T changes its priority
- Any other process raises its priority from lower to higher than i, or vice verse.

```
1   SimTime nextOutEvtTime = MAX_SIMTIME;
2   SimTime preemptInterval = 0;
3   SimTime outEvtTime;
4   for( i = 0; i < MAX_TASK_PRIORITY; i++ ) {
5     Task tsk = prioTaskQueue[i];
6     if( tsk == NULL )
7       continue;
8     outEvtTime = clock + preemptInterval + tsk.nextEvt
9     preemptInterval += tsk.nextPause;
10    if( outEvtTime < nextOutEvtTime )
11      nextOutEvtTime = outEvtTime;
12  }
```

Figure 7. OS-Wide Event Time Prediction

5.2. Synchronization Protocol

We model communications between the event sender and receiver at the transaction level which is appropriate for system modeling early in the design cycle. The interface of the synchronization protocol is given by:

- *send_event()*. This is called by the process which executes on the local HW and is about to send an event.
- *is_sync()* and *sync_resume()* which are provided by both communication sides to know whether the other one is in synchronization state and resumes it if necessary.
- *get_event()* and *return_ack()* which are provided by the sender but called by the receiver to retrieve the event and return acknowledge state to the sender.

Fig. 8 is an illustration of how the protocol works. When a sender is about to issue an event, it prepares the event data and calls *send_event()*. *send_event()* first checks whether the receiver is waiting for the event (by calling *receiver.is_sync()*). If it is true, it resumes the receiver (*receiver.sync_resume()*). Then the sender blocks itself to wait for the receiver to retrieve the event.

At the receiver side when its clock reaches the predicted time-stamp when an event may arrive, it first checks whether the sender is waiting for sending the event (by calling *sender.is_sync()*). If not, it blocks itself. By the time it is resumed by the sender, or the sender is already in synchronization state, it calls *sender.get_event()* and *sender.return_ack()* to finish the transaction. The sender resumes itself inside *return_ack()*.

Note that a receiver may enter synchronization state "prematurely" because of the conservative prediction by the sender. In that case when the receiver is resumed by the sender, it needs to advance to the actual clock when the event actually happens before receiving the event.

Deadlocks can happen when multiple modules wait to receive events from each other, owing to the conservative

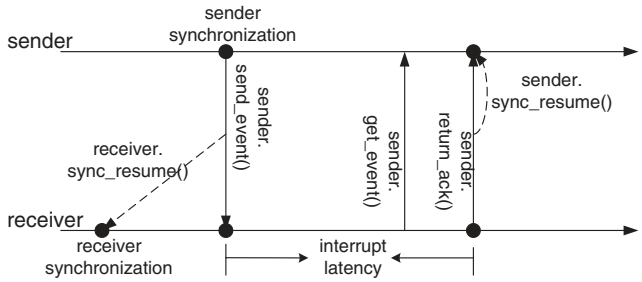

Figure 8. Synchronization Protocol

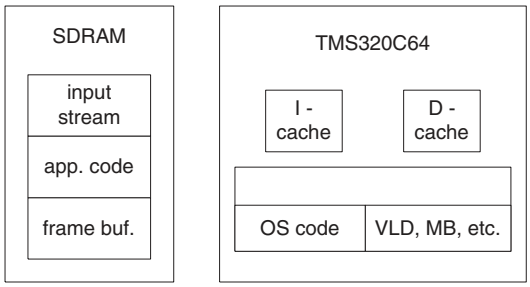

Figure 9. H.263 Decoder System

event time prediction. For instance, if module A predicted to send an event to B at 10 but actually it will send it at 20, while B predicted to send an event to A at 15 but it actually will send it at 30, both A and B will wait each other to send an event at 15 and 10, respectively. To avoid deadlock in our tool, whenever a module is about to wait for an event, it checks whether any other module with a smaller clock time stamp is waiting for it to send an event. If any of such module exists, it gives an updated event time prediction and resumes that module.

6. Experiment

We used an H.263 baseline decoder application from TI [21] to test the effectiveness of our tool. This application runs on a DM642 EVM board which mainly consists of a 600 MHz TMS320C64 DSP and 32 MB external SDRAM. The TMS320C64 DSP has a 128-Kbit L1 instruction cache, a 128-Kbit L1 data cache and a 2-Mbit internal memory. The system block diagram is shown in Fig. 9. A 4CIF format input stream is stored in the external SDRAM. After the a frame is decoded, it is placed in the frame buffer which will be transferred to a monitor to display through a video port. The RTOS is TI DSP/BIOS. The data which are placed in internal memory are (1) all VLD tables, (2) zigzag index table, (3) TCOEF length table, (4) reconstructed MB buffer, (5) reference MB buffer, and (6) IDCT buffer. All application and OS code are placed in external SDRAM and managed by instruction cache. All data block transfer and instruction cache filling use DMA. Excluding the H.263 application code, the implementation of the RTOS model and other related HW behavior models such as DMA consists of 4200+ lines of C++ code.

	Worst Frame	Best Frame	Average
Sync. read	39.27 ms	6.64 ms	31.37 ms
Async. read	30.21 ms	4.78 ms	23.07 ms

Table 1. Simulation Result of H.263 Decoder

For each frame we measured the time interval from the point it starts to be decoded to the point when the decoded frame is copied into the frame buffer. The DSP processor transfers the decoded frames to the frame buffer via DMA in asynchronous mode as explained in 3.2.3. We tested two cases for which the DSP reads input stream either by (1) synchronous blocking mode or (2) asynchronous mode. The results are shown in table 1.

As expected, the simulation results show that the asynchronous input read mode is much more efficient than the synchronous blocking mode since IO actions are in parallel with the CPU operations. For the same application, TI has reported that DM642 is able to decode H.263 baseline 4CIF stream at about 20 ms/frame in average. Our simulation results are a little inferior but close to the TI benchmark. The reasons are threefold: (1) inaccuracy of the delay annotation model, (2) inaccuracy caused by the behavior model of the DMA and cache module, and (3) data cache is disabled.

The simulation speed of our model is much faster than an instruction set simulation (ISS). On our workstation with an Intel Cerelon 1.33 GHz CPU and 256 MBytes RAM, the simulation of 2×10^9 DSP cycles takes 7.1 seconds, which is about 2.817×10^8 cycles/sec. In contrast, TI CCS simulates 7.630×10^4 cycles/sec on the same machine. The speedup is more than 3 orders of magnitude in this example.

7. Conclusion

In this paper, we present an RTOS modeling tool which is built on top of SystemC [2]. Any component integrable with SystemC can also be integrated under one simulation framework.

Our model is configurable to support modeling and timed simulation of most popular embedded RTOSes. Timing is achieved by delay annotations. The OS timing information can be derived from benchmark data provided by the OS vendor or published research results. Since OS code directly interfaces with the HW and typically executes in on-chip memory, the benchmark data can

be trusted with high confidence. Application timing information can be profiled or estimated. It is also isolated from OS timing so that changes to either one will not affect the other unless the HW architecture (processor) being modeled is changed.

We take an optimized but conservative approach to carry out the simulation. Compared to other research work, we presented an algorithm to predict timing based on the more realistic assumption that multiple processes execute concurrently on a processor, managed by a static/dynamic priority driven scheduler.

One of the problems associated with delay annotations is that the simulation clock advances in macro steps whose granularity may be so large such that the simulation clock may advance beyond the time-stamp when an input event is supposed to be handled. Such a problem makes it impossible to measure the interrupt response latency incurred by the OS being modeled. In our work the problem is solved by automatically splitting the clock advance step when the step is too large.

Experiments show that the accuracy of our modeling approach is acceptable. Compared to instruction set simulation (ISS), our tool can speed up the simulation by more than 3 orders of magnitude. This makes it a valuable tool for RTOS selection at an early design phase of an embedded system.

References

[1] Sungjoo Yoo, Gabriela Nicolescu, Lovic Gauthier and Ahmed A. Jerraya. “Automatic Generation of Fast Timed Simulation Models for Operating Systems”, Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition, Pages:620 - 627.

[2] “Functinal Specification for SystemC 2.0”, and “SystemC Version User’s Guide”, available at http://www.systemc.org.

[3] Shige Wang, Sharath Kodase, Kang G. Shin, and Daniel L. Kiskis, “Measurement of OS Services and Its Application to Performance Modelling and Analysis of Integrated Embedded Software”, Real-Time and Embedded Technology and Applications Symposium, 2002, pages:113 - 122.

[4] S. Yoo and K. Choi, “Synchronization Overhead Reduction in Timed Cosimulation”, Proc. IEEE International High Level Design Validation and Test Workshop, pp. 157-164, Nov. 1997.

[5] Youngmin Yi; Dohyung Kim; Soonhoi Ha, “Virtual synchronization technique with OS modeling for fast and time-accurate cosimulation”, Hardware/Software Codesign and System Synthesis, Oct. 2003, Pages:1 - 6.

[6] Le Moigne, R.; Pasquier, O.; Calvez, J.-P.; “A generic RTOS model for real-time systems simulation with systemC”, Design, Automation and Test in Europe Conference and Exhibition, Feb. 2004, Pages:82 - 87 Vol.3

[7] WindRiver Systems Inc., “VxWorks 5.4”, available at http://www.wrs.com/products/html/vxwks54.html.

[8] D. Desmet, D. Verkest, and H. De Man, “Operating System Based Software Generation for System-on-Chip”, Proc. Design Automation Conf., June 2000, Pages:396 - 401.

[9] “CarbonKernel” available at http://www.carbonkernel.org.

[10] Bertil Roslund and Per Andersson, “A flexible technique for OS-support in instruction level simulators”, 27th Annual Simulation Symposium 1994, pages:134 - 141.

[11] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski, “RTOS Modeling for System Level Design”, Design, Automation and Test in Europe Conference and Exhibition 2003, pages:130 - 135.

[12] Catherine Lingxia Wang, Bo Yao, Yang Yang, Zhengyong Zhu, ”A Survey of Embedded Operating System”.

[13] “TMS320 DSP/BIOS User’s Guide (Rev. B)”, available at http://www.ti.com.

[14] “DSP/BIOS Timing Benchmarks for Code Composer Studio 2.2”, available at http:// www.ti.com.

[15] Jwahar R. Bammi, Wido Kruijtzer and Luciano Lavagno, “Software Performance Estimation Strategies in a System-Level Design Tool”, Hardware/Software Codesign, Proceedings of the Eighth International Workshop on, 3-5 May 2000, pages: 82-86.

[16] K. Hines and G. Borriello, “A Geographically Distributed Framework for Embedded System Design and Validation”, Proc. Design Automation Conf., pp. 140 - 145, June 1998.

[17] Sungjoo Yoo, “Performance Improvement of HW/SW Cosimulation Based on Synchroniztion Overhead Reduction”, dissertation 2000.

[18] Christian Kreiner, Christian Steger, Egon Teiniker, Reinhold Weiss, “A HW/SW Codesign Framework based on Distributed DSP Virtual Machines”, Digital Systems, Design, Proceedings. Euromicro Symposium on, 4-6 Sept. 2001 Pages:212 - 219.

[19] Marcello Lajolo, Mihai Lazarescu and Alberto Sangiovanni-Vincentelli, “A Compilation-based Software Estimation Scheme for Hardware/Software Co-Simulation”, Hardware/Software Codesign, Proceedings of the Seventh International Workshop on, 3-5 May 1999, Pages:85 - 89.

[20] “Broadway Compiler Project”, available at http://www.cs.utexas.edu/users/less/broadway.html.

[21] “H.263 Decoder: TMS320C6000 Implementation”, available at http://focus.ti.com/lit/an/spra703/spra703.pdf.