A Generic and Accurate RTOS-centric Embedded System Modelling and Simulation Framework

Ke Yu, Neil C. Audsley
Dept. of Computer Science, University of York
York, UK
Email: {ke, neil}@cs.york.ac.uk

***Abstract*— Real-time Operating System (RTOS) modelling and SystemC-based system-level hardware/software co-simulation have become important issues for early design space exploration in the development of real-time embedded systems. This paper presents a generic and accurate RTOS-centric embedded system modelling and simulation framework. It allows modelling and simulating applications, the RTOS, the CPU processing element and hardware components in a unified SystemC-based framework. Compared with previous system-level RTOS modelling works, this framework (1) enhances modelling flexibility by supporting hybrid simulation of abstract software models and delay-annotated native application codes, (2) improves functionality of the RTOS model by providing generic and POSIX-like services, and (3) achieves accurate simulation in terms of both timing accuracy and the simulation flow. Experimental results show the high accuracy and fast performance of our simulation, with small accuracy loss compared with cycle-accurate instruction set simulation.**

I. INTRODUCTION

In recent years, with embedded systems moving toward System-on-Chip (SoC) platforms, the complexity of embedded software (SW) is also increasing rapidly. The Real-Time Operating System (RTOS) has become an essential component in many real-time embedded systems. It provides efficient controlling facilities as well as guaranteed services for upper-layer application software and underlying hardware (HW) resources. The traditional SW simulation approach, which executes a real RTOS and applications in an instruction set simulator, appears to be time consuming and available too late when confronting ever-increasing design complexity. To cope with the design challenge, system-level RTOS modelling and simulation have been proposed as enabling techniques, to simulate and evaluate different embedded SW (including both applications and the OS) design alternatives at early design phases. By means of System Level Design Languages (SLDL) (e.g., SystemC and SpecC), these methods usually build a generic RTOS model that dynamically simulates abstract SW models or native application codes on a host machine, in order to mimic SW timing and functional behaviour on the target machine. They are used to evaluate system-wide dynamic SW functional and timing properties, such as scheduling policies and applications execution times.

However, there still exist some problems in this area, which affect the functional and timing accuracy of models, as well as their simulation performance. For example, from the perspective of maximising flexibility of system-level design, designers may want to simulate multiple abstraction-level SW models together. However, current RTOS modelling research is incapable of integrating coarse-timed abstract task models (i.e., associated with best-case and worst-case execution times) and fine-timed native application codes (i.e., associated with line-by-line delay annotations) in one simulator. Besides, from the perspective of RTOS engineering, some RTOS models provide simplistic task management and limited synchronisation services, which are inadequate to imitate behaviour of a real multitasking RTOS. Furthermore, the low timing accuracy is a common, yet critical, problem borne by many RTOS modelling approaches. On one hand, this is due to the lack of RTOS services' timing overheads in modelling. On the other hand, many methods rely excessively on the un-interruptible SLDL "*wait-for-delay*" time advance mechanism [1] [2], consequently task switches and HW/SW synchronisation only happen at limited pre-defined pre-emption points.

In this paper, we present a system-level SystemC-based RTOS-centric modelling framework. Its main objective is to fast simulate and evaluate behaviour of real-time embedded software with good accuracy in early design phases. The simulated target system's dynamic execution scenarios can be exposed by tracing diverse system events and values, e.g., RTOS kernel calls, RTOS runtime overheads, task execution times, dynamic scheduling decisions, task synchronisation and communication activities, interrupt handling latencies, context switch times, and other user concerned properties. It does so by integrating multiple-level applications, RTOS, processing element and hardware component models in a unified SystemC prototyping environment. The core is a generic RTOS simulation model, which supplies a set of fundamental services including thread management, scheduling services, synchronisation and inter-task communication mechanisms, clock services, context switch and interrupt handling services, etc. These services partially conform to POSIX Dedicated Real-time System Profile (PSE53) specification [3] in order to supply standardised functions. To build a predictable RTOS timing model, timing overheads of various RTOS services are considered in models. This simulation framework includes the previously developed "Live CPU Model" [4], which supports SW timing simulation as well as guarantees good HW/SW synchronisation timing accuracy.

Our work also remedies current SystemC language's (v2.2) deficient capability on real-time software modelling,

i.e., there is neither the priority concept nor pre-emptive scheduling. All models in the modelling framework are built on the top of SystemC library (Refer to Figure 2). We use both basic SystemC core language and its simulation kernel, without modifying the library at all. This is helpful to popularise our model to the emerging SystemC-based embedded system design world.

II. RELATED WORK

RTOS modelling has been an important topic in embedded systems simulation-based design. Various RTOS models have been developed in the context of high-level abstract SW simulation [5] [6] [7], delay-annotated native SW simulation [8] [9], HW/SW co-simulation [10] [11] and system-level design refinement research [12] [13]. These models can be categorised depending on their locations in the top-down embedded systems design flow, and with regard to their timing accuracy levels (Refer to Figure 1).

Abstract RTOS modelling is applicable to early system design phases, such as specification, system analysis, and SW/HW partitioning stages, when the target platform is undetermined and SW codes are not implemented. In this approach, applications are normally organised into some abstract task models associated with coarse-grained temporal estimates, e.g., period, deadline, and execution times. An RTOS model provides basic primitives to control "*start*" and "*termination*" of a task, between which there is a time interval representing the task's execution cost. Inter-task synchronisation/IPC services and interrupt handling are usually not considered in this kind of model. A SpecC-based abstract RTOS model for system-level design is presented in [12] [13]. It provides sixteen basic primitives to support task management and scheduling. Its subsequent work in [1] resolves the initial HW/SW synchronisation problem by using an improved "*wait-for-delay*" method named "Result-oriented Modelling". In [5] the authors present an abstract SystemC-based RTOS model and extend its use for MPSoC design space exploration [14]. This work decomposes a real-time embedded system into three compact submodels: the task graph model, the scheduler model, and the link communication model. Real-time scheduling assessments have been addressed in [6] [7], in which different scheduling policies are evaluated through abstract RTOS simulation. In most models, overheads of RTOS services are not adequately considered, but [15] has the advantage of taking three services' overheads into account in a generic RTOS model.

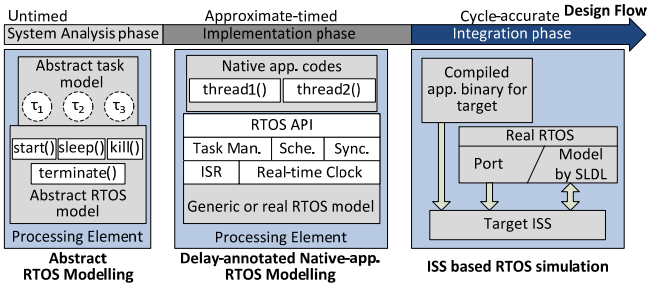


Figure 1. RTOS models in embedded systems design

Delay-annotated native-application RTOS modelling aims to simulate SW in the system implementation phase, when the target platform is in the process of being selected and application codes are being developed. SW execution delays are measured and annotated in models at some finer granularities (e.g., function level, block level, and line level), so timing accuracy becomes a major focus point in this approach. The RTOS model often supplies comprehensive and specific services, and contains more timing information. In [9], a SystemC-based Texas Instrument DSP/BIOS RTOS simulation tool is presented. It comprises detailed OS timing information, which is derived from benchmark tests. It proposes a time-stamp prediction technique in order to solve HW/SW synchronisation problems, but it has a tight requirement that applications should report their interrupt times to the RTOS kernel. The TIMA laboratory presents some results on native SW simulation for SoC HW/SW co-simulation research in [11] [16]. In [11] they propose a "variable timing granularities" method to solve HW/SW synchronisation problems by trading off the simulation performance with the timing accuracy; to further tackle this problem, in [16] they present a different method by using a "simulation environment abstraction layer" to synchronise HW and SW simulation clock. In [8] a POSIX compliant RTOS model is developed on top of SystemC. It applies a dynamic delay annotation method by assigning each C++ operator with a corresponding target-platform execution cost. However, the interrupt handling problem is not explicitly solved.

Instruction set simulation (ISS) is conventionally used for SW simulation at the final system integration phase. Finished SW codes are cross-compiled and simulated in a cycle-accurate instruction set simulator that represents the target processor's behaviour. The high accuracy and low simulation performance are its two contradictory characteristics. In this approach, a real RTOS (e.g., µC/OS-II in [17] and µClinux in [18]) is usually ported in the ISS to manage applications execution. To speed up simulation, some variation approaches propose to build an RTOS model on top of SLDL [2] [10], and combine it with an ISS which is still used to simulate applications.

Our RTOS model combines many features of both "abstract RTOS modelling" and "native-application RTOS modelling". It applies to the conjunction area of system analysis and implementation design phases. Compared with existing works, we improve the RTOS model's functionality to supply realistic SW simulation for. It can integrate hybrid abstract task models and native-code task models in a single simulator to enhance modelling flexibility. The high simulation performance and good timing accuracy are preserved at the same time in our simulator because of the use of the "Live CPU Model".

III. SOFTWARE SYSTEM MODELLING

As shown in Figure 2, a real-time embedded system can be generally decomposed into three layers: the application layer, the RTOS layer and the hardware layer.

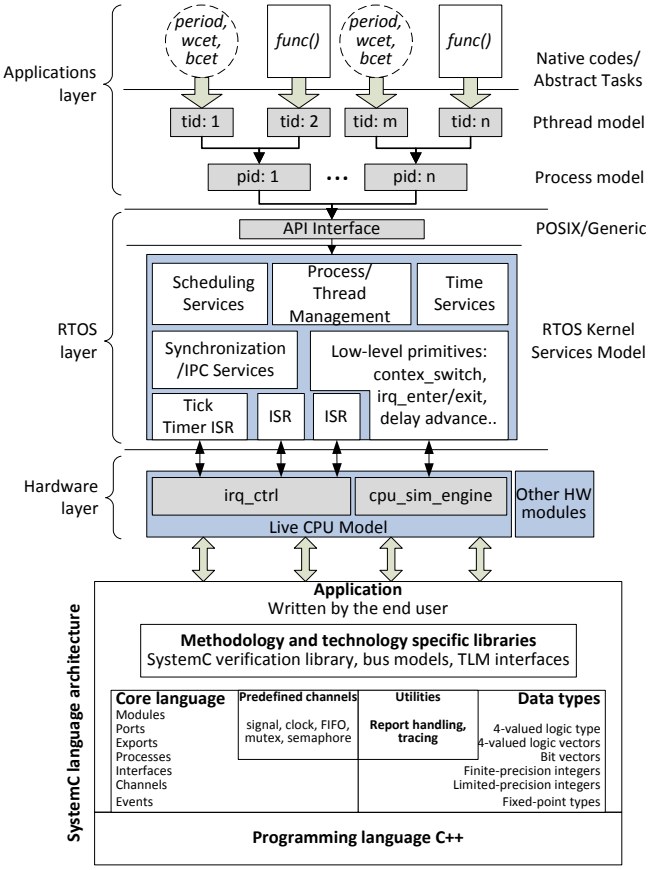


Figure 2. Layerd system model

In our previous work, we developed a hardware layer model whose core is a "Live CPU Model". Its main purpose is to support and improve system-level SW simulation from the perspective of hardware platform. In conventional system-level SW simulation (e.g., the left two models in Figure 1), the application model and the RTOS model construct a processing element, and in fact there is not a hardware CPU model. Unlike them, the "Live CPU Model" executes SW delay annotations in a way comparable to the way a real CPU executes instructions. The "Live CPU Model" also monitors external interrupts and can start, stop, and resume a SW delay process without any undesired latency. This hardware layer model is then combined to construct a whole embedded system simulation framework in this paper. In the following, we describe software layers modelling in detail.

A. *Applications Modelling*

In embedded software development, applications are usually subdivided into a set of concurrent and cooperating units. These concurrent units are commonly implemented in three concepts: the *task* model, the *thread* model and the *process* model. In simple systems, there is only one kind of unit, so people may synonymously use three concepts. Many RTOS models are implemented in this way as well. However, in some complex RTOSs (e.g., QNX, LynxOS and other POSIX-compliant ones), applications are managed in both

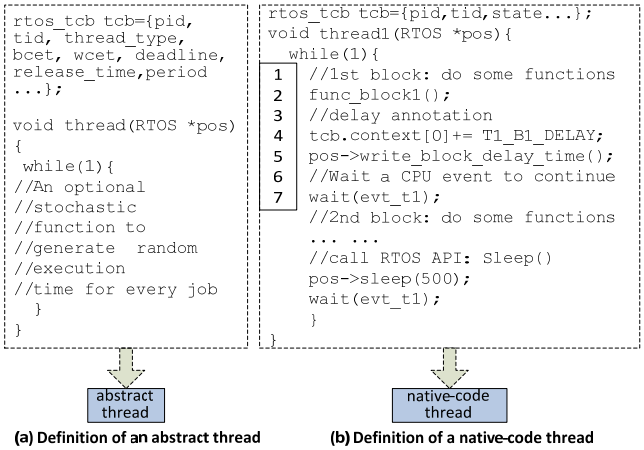


Figure 3. Two types of application models

the *process (i.e., an application)* model and the *thread (i.e., a task)* model. This is mainly for such a consideration that the threads' context-switch overheads are lower than processes', because threads share some contexts within a process. This two-level structure can also better utilise the high-performance parallel multi-threading HW architecture and flexible multi-level real-time scheduling policies. Our application model conforms to the Real-time POSIX PSE53 specification, so it assumes a single processor computing system consisting of multiple processes and threads (*pthread*). The *pthread* is the only SW functional execution unit scheduled for the CPU resource.

According to our hybrid application model concept, an application thread can be modelled as an abstract model, or as a native-code model.

The first case applies to situations when application codes have not been finished for modelling, or the simulation user does not have interests in functional simulation. In Figure 3(a), the behaviour of an abstract thread model is characterised by a set of timing parameters, e.g., best-case execution time (*bcet*), worst-case execution time (*wcet*), deadline, release time, and period, etc. These temporal parameters, along with thread identity information (e.g., process ID (*pid*), thread ID (*tid*) and the thread type), are defined in a structure variable, namely a temporary thread control block argument, which will be passed to the RTOS during thread creation. The thread's function body usually contains nothing functional codes. Possibly, the simulation user can also appoint a probabilistic function in order to generate random execution times for every job, like the idea in [6].

If applications come with functional codes and corresponding delay estimations, then we can build a native-code thread model. In Figure 3(b), a structure variable still defines a thread's identity information, but it no longer contains timing properties. A RTOS object is passed to the function body, in order to let a thread use RTOS services. Timing delay information interleaves with code blocks in the function body. The delay annotation granularity depends on the simu-

lation user's choice, such as source code line level, function level, and task level. A delay annotation and time advance process includes three steps:

1) Adding the delay to thread's current delay value (Figure 3(b) line4);

2) Using the RTOS function *write_block_delay_time()* to inject the latest thread's delay value into the Live CPU Model (line5);

3) Waiting for a SystemC `sc_event` (exclusive to each thread) that will be released by the Live CPU Model after the thread's delay is consumed totally (line7).

This native-code simulation method has a significant difference from other similar methods. Unlike [6] [11], delay annotation statements in our model are not used to define explicit pre-emption points for HW/SW synchronisation. The main purpose is to notify the processing element (the Live CPU Model) as to how much computing resource a code block needs, and then to wait for simulation time advance. Interruption and pre-emption can happen at any necessary (i.e., there is an interrupt) and possible (i.e., interrupts are enabled) time points during a delay.

B. RTOS modelling

An RTOS usually consists of a kernel to provide minimal services that enable concurrent threads to utilise hardware resources efficiently and predictably. Figure 2 depicts the block diagram of the RTOS kernel model. It is dedicated to the following services: thread/process management, scheduling services, synchronisation/IPC services, time/clock services and interrupt handling services. An application thread can access these services via the Application Program Interface (API) layer, which is implemented with real-time POSIX standard.

1) Thread and Process Management

The RTOS kernel model provides direct support for *pthread* management. A pre-defined number of processes and threads are created during the RTOS initialisation, and are put in two pools *"rtos_pcb_array[]"* and *"rtos_tcb_array[]"*. Figure 4 illustrates how to create a *pthread* in modelling. We implement the standard POSIX *pthread_create()* function on top of SystemC. Calling this function will create a SystemC module that includes two SystemC processes: *create_thread_routine()* and *run_thread_routine()*. As we have discussed before, a thread

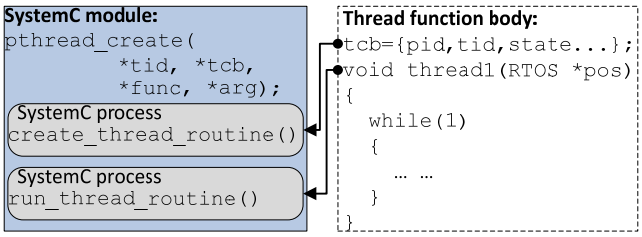


Figure 4. Pthread creation in SystemC

is defined by both a structure variable with some identity attributes and a function body. The structure variable is processed in the *create_thread_routine()* to generate an associated *thread control block* (*tcb*), whilst the function body is wrapped in the *run_thread_routine()* for timing simulation. Similarly, a process is created by a function *spawn()* and a *process control block* (*pcb*) contains its attributes.

The *thread control block* comprises all information required by the RTOS kernel about a thread, for example the unique thread ID (*tid*), the belonged process ID (*pid*), the base priority, the effective priority, timing attributes, pointers to synchronisation events, pointers to other *tcbs* etc. More importantly, a *delay context* of a thread is also kept in the *tcb*. The concept *delay context* refers to a thread's temporal information that is useful for timing simulation on the Live CPU Model. It is similar to the *thread context* about CPU registers' values (e.g., the program counter and the status register) in a real computer. Table I shows six members of a *delay context*. At simulation runtime, the *delay context* changes continuously. When a thread is simulating (i.e., occupying the CPU), its *delay context* is loaded into the Live CPU Model. When a thread stops simulating (i.e., releasing the CPU), the latest *delay context* is saved back to the thread's *tcb*. These "*load*" and "*save*" primitives constitute the RTOS context switch function in modelling.

TABLE I. DELAY CONTEXT OF A THREAD

Context Member	Member Name	Description
context[0]	block_exec_time	delay slice of the running code block
context[1]	thread_exec_time	total delay of this thread job
context[2]	thread_abs_dln	absolute deadline of this thread job
context[3]	thread_used_time	the delay time has been used
context[4]	thread_cur_sta_time	start time of current delay slice
context[5]	thread_sleep_length	the time length to suspend a thread for

In order to accommodate novel hybrid hard/soft/non real-time embedded applications, our RTOS model supports real-time periodic, aperiodic, sporadic and non real-time threads. We notice that many RTOSs (e.g., μC/OS-II, RTEMS, QNX, and VxWorks) do not have special system service to support periodic threads. The periodic execution is realised by using time functions at the user level, whereas some other RTOSs (e.g., RTLinux, RTAI) and most abstract RTOS models provide direct system primitives to support periodic threads. Our RTOS model implements both approaches. The periodic execution related information, such as thread type, relative deadline, and next release time etc. can be stored in the thread *tcb*. The RTOS kernel can track and update them for a periodic execution. As well, the POSIX *sleep()* function is implemented in our model to implement user-manipulated periodic execution. Aperiodic and sporadic threads with critical deadlines are triggered by external interrupts through interrupt service routines. Non real-time threads are usually given the lowest priorities.

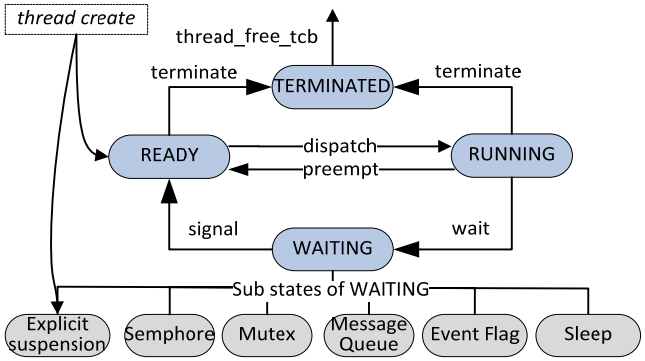


Figure 5. Task state transitions diagram

2) Scheduling services modelling

Task state machine is the basis of thread scheduling services. The state machine models in [14] and [19] have a similar structure, including four states: *idle*, *ready*, *running*, and *preempted*. However, their *ready* and *preempted* states in fact represent the same *ready* state in a classical RTOS structure; namely, this task state machine is lack of the *waiting* state that is important for synchronisation services. The model in [20] builds a seven-state traditional UNIX task model, which is completed but is not very common in RTOSs. The task state model in [15] implements a canonical structure to support concurrent execution, which consists of three basic states: *ready, running*, and *waiting*. Besides, we observe that, in some real RTOSs, the task state machine usually contains additional states to support specific kernel services, such as a *sleeping* state for a timed suspension and an *explicit suspension* state for an unlimited suspension. With reference to µC/OS-II, µTRON4.0, QNX, RTEMS, and classical RTOS models in [21], we apply a four-state extensible task state machine: *ready*, *running*, *waiting (with six sub-states by default)*, and *terminated* (Figure 5). In addition to three basic states, the *terminated* state means that a thread has been deleted. This thread state machine is extensible because the important *waiting* state can be specified into different sub-states by just setting a *wait_flag* in the thread *tcb*. For example, if we want to model the µTRON4.0 RTOS, we just need to shrink the *waiting* state into three sub-states: *waiting*, *waiting-suspended*, and *suspended*.

According to a thread's state, its *tcb* is organised in several priority-descending queues (e.g., *ready_queue*, *waiting_queue* and *terminated_queue*) in the RTOS model. In a uniprocessor system, only one thread can execute at any time, so *running* state does not need a queue. Thread management, e.g., creation, suspension, dispatch, resumption, and termination, is implemented by various RTOS system calls through moving *tcbs* between these queues.

Like most RTOSs, we model a priority-based pre-emptive scheduler in two ways: tick scheduling and event-driven scheduling [22]. In tick scheduling, the tick timer Interrupt Service Routine (ISR) *tick_isr()* is periodically triggered by a clock interrupt *rtc_clk*. The interval of this clock interrupt is defined as the time resolution of the system, the so-called system tick. The tick length is fully configurable in our model. The function *tick_isr()* takes charge of updating threads' timers, and invoking the priority-based *scheduler()* to make a scheduling decision. In event-driven scheduling, the *scheduler()* function is invoked by RTOS services after they change any thread's state, such as the *sleep* service and the *wait semaphore* service.

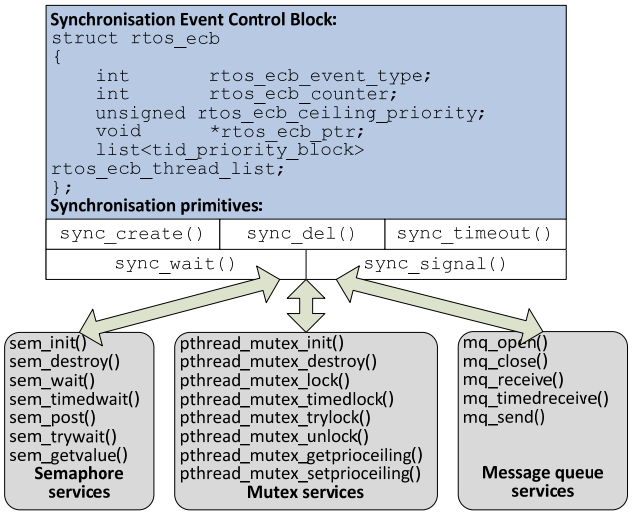


Figure 6. Synchronisation services

Two priority-based scheduling mechanisms are supported: Fixed-priority Scheduling (FPS) and Dynamic-priority EDF scheduling. The Deadline Monotonic (DM) algorithm and the Rate Monotonic (RM) algorithm can be applied for FPS. In EDF scheduling mode, the *tcb* member "*effective_priority*" and the *delay context* member "*thread_abs_dln*" are dynamically updated in simulation. For *ready* threads possessing the same priority, POSIX First-In-First-Out (SCHED_FIFO) and Round-robin (SCHED_RR) algorithms are applied. If there is no *ready* thread after scheduling, an *idle* thread with the lowest priority is dispatched to run, which indicates the processor's idle state. Some POSIX functions about *pthread* scheduling, such as *pthread_setschedparam()*, *pthread_getschedparam()*, and *pthread_setschedprio()*, are implemented in our model to control and change the scheduling policy and parameters.

3) Thread Synchronisation and Communication modelling

In a multithreading RTOS environment, application threads need to access shared resources and exchange information and data with each other to cooperate properly. There is a rich variety of RTOS synchronisation and communication mechanisms, such as semaphores, mutexes, conditional variables, events, signals, message queues, etc. We notice that SystemC language also provides three built-in synchronisation primitive channels, e.g., *sc_semaphore*, *sc_mutex*, and *sc_fifo*. However, they are improper to use in an RTOS model directly, since they do not support the priority-based pre-emptive scheduling, but rely on the non-deterministic and non-preemptive SystemC kernel scheduler. For instance, if several threads are waiting for a semaphore, it is unsure which thread will be activated after a *post semaphore* opera-

tion. In the RTOS model, we realise three real-time POSIX synchronisation and communication methods: *semaphores*, *mutexes* and *message queues*. The Priority Ceiling Protocol (PCP) (so-called priority protection protocol in POSIX) is applied for *mutexes* to avoid the priority inversion problem.

We use the Event Control Block concept, which is similar to the mechanism in µC/OS-II, to allocate an *rtos_ecb* control block (See Figure 6) to each semaphore, mutex, and message queue object. Five primitives are responsible for managing these *rtos_ecbs*. Real-time POSIX synchronisation services are implemented by using these primitives. When the primitive *sync_create()* is called to create a synchronisation relationship, the related thread's *tcb* stores a pointer to an *rtos_ecb* object, at the same time, the *rtos_ecb* also stores the threads' *tid* and *priority* in its *rtos_ecb_thread_list*. In contrast, the *sync_del()* primitive destroys the link between an *rtos_ecb* and the *tcb*. The *sync_wait()* primitive performs the P operation to make the *running* thread go to *waiting* state. The *sync_signal()* primitive executes the V operation to let the highest-priority *waiting* thread enter *ready* state. The *sync_timeout()* is used to make a *waiting* thread *ready* when time is out.

4) Interrupt handling modelling

Interrupt handling is a crucial mission of the RTOS to service interrupt requests (IRQ) generated by external peripheral devices. Our RTOS model provides a configurable interrupt handling model. As shown in Figure 7, this model is constructed with three layers: HW interrupt controller[1] (*irq_ctrl*), RTOS-kernel handler, and ISRs. Once the *irq_ctrl* catches an IRQ, the Live CPU Model will stop current SW simulation and branch to a SW handler. There are a number of different SW interrupt handling schemes regarding various experimental or commercial RTOSs and processors. Our model carries out two main schemes: RTOS-assisted interrupt handling and HW vector interrupt handing. It is worth noting that, no matter in which scheme, nested, prioritised, and maskable interrupt handling is supported.

Figure 7(A) depicts the process of RTOS-assisted interrupt handling. In this scheme, an RTOS-kernel handler *interrupt_handler_enter()* is the entry point for all IRQs (Step1.1). This *interrupt_handler_enter()* is implemented as a SystemC process, so it can be aware of a related SystemC *sc_event* sent by the HW interrupt controller. After this entry handler identifies the external interrupt source, it executes an appropriate ISR in Step1.2. This ISR is programmed as a high-priority (i.e. higher than any applications) thread, which contains simple and non-blocking functions to serve an IRQ quickly. Possibly, it could call a synchronisation function to activate a *waiting* thread. When servicing is complete, control is passed to another kernel handler *interrupt_handler_exit()* in Step1.3. This handler checks and processes any possible nested IRQ firstly, and then it schedules the highest-priority *ready* thread to run in Step1.4.

[1] The HW interrupt controller model has been introduced in our previous work; its main function is to monitor external IRQ lines.

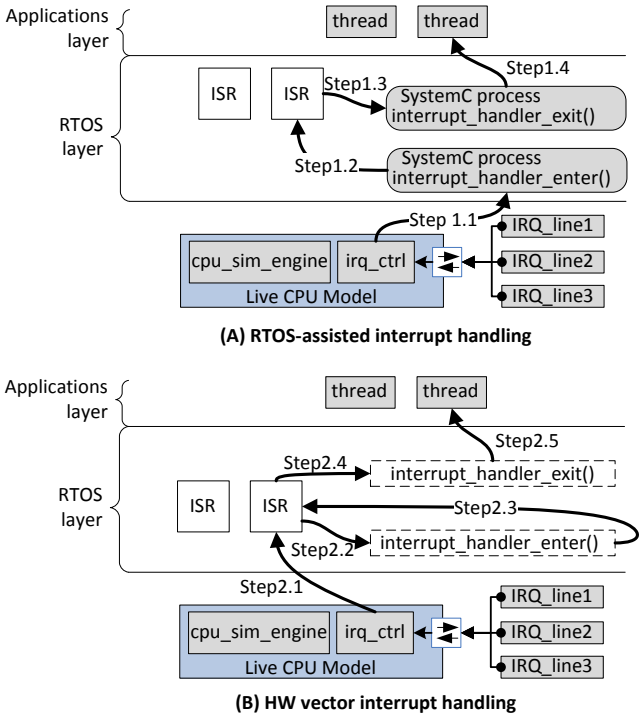


Figure 7. Two IRQ handling models

Figure 7(B) illustrates the HW vector interrupt handling scheme. In this model, the interrupt source can be obtained directly from the HW interrupt controller, so an ISR is loaded from the vector table to execute promptly (Step2.1). The ISR carries out servicing functions and invokes the RTOS function *interrupt_handler_enter()*, which is a light-weight function to increment system variables in this model (Step2.2). In Step2.4, the RTOS function *interrupt_handler_exit()* is called to schedule the new highest-priority *ready* thread. In final Step2.5, the highest-priority *ready* application thread is subsequently dispatched to run.

```
1  void thread(RTOS *pos)
2  {
3    ......
4    pos->sem_init(pecb0, 0, 0);
5    tcb.context[0] += SEM_INIT_FUNC_DELAY_TIME;
6    pos->write_block_delay_time();
7    wait(tid.evt);
8    ... ...
   }
```

(A) Annotate interruptible delay into the application thread

```
1  int sem_init(...)
2  {
3    ... ...            //interruptible codes
4    enter_critical(); //Enter critical section
5    ... ...            //un-interruptible codes
6    wait(SEM_INIT_CS,SC_NS);//Advance sim. time
7    exit_critical();  //Exit critical section
8    ... ...            //interruptible codes
   }
```

(B) Advance un-interruptible delay in the RTOS service

Figure 8. Simulation time advance methods for the RTOS service

IV. DISCUSSIONS ON MODEL IMPLEMENTATION IN SYSTEMC

A. Building timed RTOS service models

A major advantage of our RTOS model, compared to other works, is that we consider timing overheads of various RTOS services. Building a timed simulation model for an RTOS service includes two jobs: collecting delay estimation and then annotating them into models. Regarding the first job, we use the conventional ISS-based approach to measure overheads of a specific RTOS on a target processor ISS. The second job is more worthy of discussion. When we simulate an application thread model, we can accurately *begin*, *stop*, and *resume* its delay advance by recording its time delay information in its *tcb*. Delay data in the *tcb* are elaborately consumed and updated by the Live CPU model, which can guarantee the timing accuracy of simulation. However, the timing simulation of an RTOS service is not the case, because an RTOS service does not have a control block which can store its delay information. We solve this problem by dividing each RTOS service's time delay into two parts: the interruptible part and the un-interruptible part (Refer to Figure 8).

The interruptible part means that interrupts are allowed during this delay period. Consequently, we need to find a method to "consume" this delay whilst ensure HW/SW synchronisation accuracy in simulation. The use of the Live CPU model is a good choice in simulation. Hence, we annotate this interruptible delay part to the service's calling application thread and place it after the function call. For example, in Figure 8 (A), a semaphore initialisation function executes at line4. Subsequently, its interruptible delay is written into its residing thread's *delay context* at line5. Accordingly, the following delay advance process (line6 and line7) is the same as a normal one.

The un-interruptible delay part relates to a *critical section* concept in the RTOS, during which interrupts are disabled. Hence, it is not necessary to worry about the HW/SW synchronisation problem in this delay period. We hereby use a simple wait-for-delay statement to advance the simulation time (Figure 8 (B) line6). It is worthwhile to indicate that this method also avoids invoking the Live CPU Model and decreases SystemC engine switches, which will necessarily improve simulation speed.

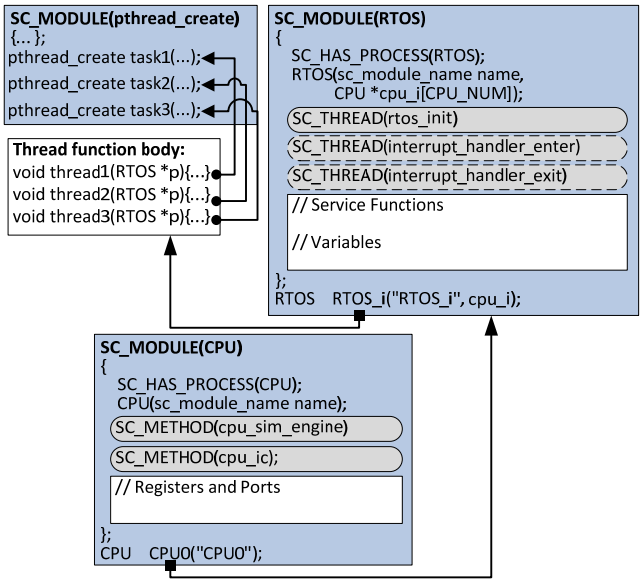


Figure 9. Simulation framework architecture

The RTOS model only contains one essential (i.e. *rtos_init()* to start simulation) and two optional (i.e. *interrupt_handler_enter()* and *interrupt_handler_exit()* in case of RTOS-assisted interrupt handling) SystemC processes. We implement other RTOS services as normal C++ functions. This is based on similar considerations to [15]: 1) such an RTOS model is closer to a real RTOS implementation that using procedure calls; 2) less SystemC processes imply less simulation engine switches, therefore the simulation performance is enhanced.

B. Simulation framework architecture

The mainframe of the simulator is constructed from three types of SystemC modules (namely C++ classes): the *pthread_create* module, the RTOS module, and the Live CPU module, which represents applications, the RTOS and the processor respectively (Figure 9). An object of a Live CPU module is used as an argument by an RTOS object, in order to let the RTOS use the CPU resource. As well, the RTOS object is employed by a *pthread_create* object, which means that a thread is within an RTOS. Normally, there are several *pthread_create* objects in simulation standing for several *pthreads*. In this way, applications, the RTOS and the processing element are connected in a straight manner. This modular architecture makes our simulation framework hierarchical, low coupling and especially extensible. For example, the RTOS bears the potential to simulate on a multi-processor platform by accepting several CPU objects.

V. EXPERIMENTAL RESULTS

In order to demonstrate the functional and timing accuracy of the proposed RTOS-centric simulation, we test it with an A/D data collection example (Figure 10). Three threads take charge of watching the keyboard, collecting A/D data and sending out results via the serial port. Tasks roughly have periods as 90ms, 100ms and 510ms. According to the RM algorithm, they are allocated descending priorities. A

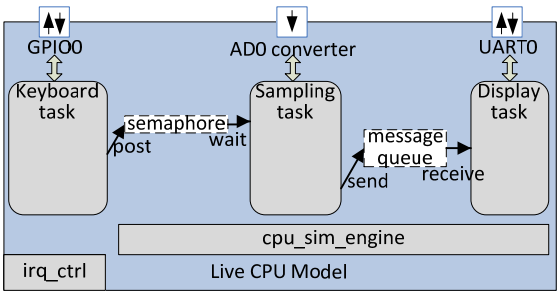
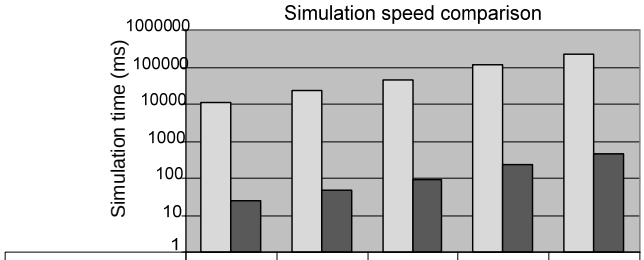


Figure 10. A/D data collection example

tick timer ISR is associated with a timer IRQ to drive tick scheduling with a 5ms tick length. A semaphore and a message queue provide synchronisation and communication services between threads. Same applications and the µC/OS-II RTOS are executed on the KEIL µVision ARM ISS, which is seen as a cycle-accurate reference model in the experiment. The target processor is configured as a 48MHz NXP LPC2378 processor. The RTOS services' timing overheads are measured based on the µC/OS-II with this ISS. All tests are executed on an x86 PC at 1.86GHz.

We examine the RTOS-centric simulator in three aspects: 1) the simulation performance compared with ISS simulation; 2) the functional correctness of the simulation by examining generated results; 3) the simulation timing accuracy of RTOS-centric simulation compared with ISS simulation.

In order to compare the speed of RTOS-centric simulation with the standard ISS simulation, we let each simulator simulate 500ms, 1000ms, 2000ms, 5000ms and 10000ms target time. Even during the shortest 500ms simulation, three tasks can repeat at lease 55, 50 and 10 jobs. Not surprisingly, as a system-level SW simulator, RTOS-centric simulator gains much faster speed than ISS simulation. The Figure 11 reveals the impressed simulation performance of RTOS-centric simulation. Its simulation speed is about 500 times faster than the ISS simulator.



Target simulated time (ms)	500	1000	2000	5000	10000
ISS simulation time (ms)	11500	22960	47130	116130	225020
RTOS-centric simulation time(ms)	24	47	94	233	458

Figure 11. Simulation speed comparison

If a simulator performs correct functions, it should be able to generate similar simulation sequences and results at right time compared with real execution. In the experiment, we input same stimuli, e.g., keyboard signals and voltages, into both µVision ARM ISS and the RTOS-centric simulator. We observe A/D converting results, which are generated after multitasking interactions between applications and the RTOS. The Figure 12(Left) shows the screen shots of the ISS simulator's serial port output. The texts in the box show the time stamp of a corresponding A/D output. The Figure 12(Right) is a trace file of the RTOS-centric simulator and bold texts show time stamps of corresponding results. We can observe that the RTOS-centric simulator produces similar A/D results at very close times to the ISS simulator, which demonstrates its functional correctness.

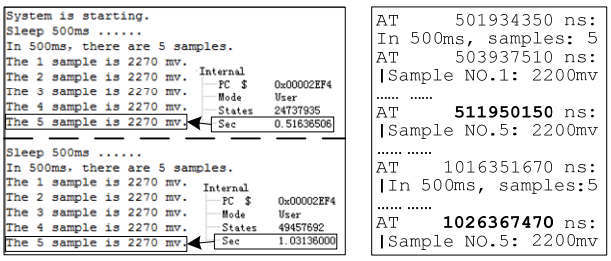


Figure 12. Simulation results comparison: ISS results (Left) and RTOS-centric simulator results (Right)

Conventionally, people examine the timing accuracy of simulation by comparing its simulated clock cycles of task completion with a more accurate standard simulator's. If both simulators run same applications and they consume similar numbers of cycles (or simulated target time) upon completion, then their timing accuracy is believed to be close. In order to make a more accurate comparison, we improve this approach by inserting more observation (comparison) points in the simulation flow, instead of only comparing total numbers. These observation points are selected to cover all three threads' codes. The Figure 13 shows timing accuracy comparison between the ISS simulator and the RTOS-centric simulator. The X-axis is 22 observation points in simulation flow and the Y-axis is the simulated target time at each observation point, which ranges from 0 to 600ms. In the figure, two simulator flows' curves are in close accordance, which reveals the good accuracy intuitively.

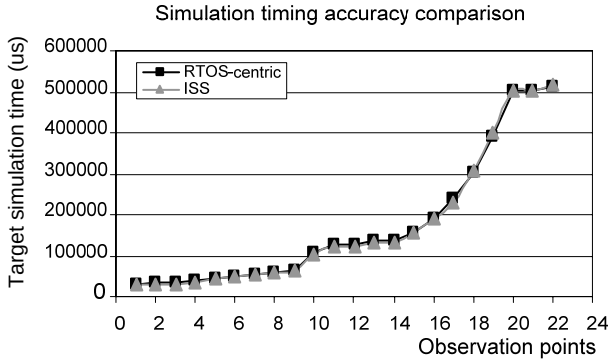


Figure 13. Simulation timing accuracy comparison

The Table II is the exact timing accuracy loss of the RTOS-centric simulation compared with ISS simulation at 22 comparison points. The accuracy loss is defined as:

$$loss = \frac{\left| Simulated\_time_{RTOS-centric} - Simulated\_time_{ISS} \right|}{Simulated\_time_{ISS}} \times 100\%$$

The table shows that the accuracy loss is marginal in this experiment. The main reason is the elaborate structure of the whole simulation framework. It is also because that delay information of both applications and RTOS services is carefully measured on the ISS before use in RTOS-centric simulation. If we cannot acquire accurate delay estimations,

then the timing accuracy of RTOS-centric simulation is necessarily affected.

TABLE II. ACCURACY LOSS OF THE RTOS-CENTRIC SIMULATION COMPARED WITH ISS

Comparison point	*#1*	*#2*	*#3*	*#4*	*#5*	*#6*
Accuracy loss	0.34%	0.34%	0.34%	0.68%	0.64%	0.61%
Comparison point	*#7*	*#8*	*#9*	*#10*	*#11*	*#12*
Accuracy loss	0.59%	0.57%	0.56%	4.44%	3.76%	3.76%
Comparison point	*#13*	*#14*	*#15*	*#16*	*#17*	*#18*
Accuracy loss	3.50%	3.51%	0.13%	0.03%	2.20%	1.49%
Comparison point	*#19*	*#20*	*#21*	*#22*		
Accuracy loss	2.52%	0.08%	0.09%	0.22%		

VI. CONCLUSION AND FUTURE WORK

This paper presents a generic and accurate system-level RTOS-centric embedded software simulation framework. The framework supports simulation of abstract application models, native application codes, the configurable RTOS model, the processing element mode, and various hardware models in a unified SystemC environment. It can help designers to evaluate both functional and timing effects of the projected real-time embedded software design fast and early. Its speedup and accuracy have been demonstrated by a comparison experiment with an ISS simulator.

Future work is towards a multiprocessor RTOS model, in which we will integrate multiple Live CPU Models and multi-level scheduling policies.

REFERENCES

[1] G. Schirner and R. Domer, "Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling," Design, Automation and Test in Europe, 2008. DATE'08, pp. 122-127, 2008.

[2] M. Krause and O. Bringmann, "Combination of Instruction Set Simulation and Abstract RTOS Model Execution for Fast and Accurate Target Software Evaluation," in 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis: ACM New York, NY, USA, 2008, pp. 143-148.

[3] "IEEE Std 1003.13-2003, IEEE Standard for Information Technology-Standardized Application Environment Profile (AEP)-POSIX Realtime and Embedded Application Support," The Institute of Electrical and Electronics Engineers.2004.

[4] K. Yu and N. C. Audsley, "A Mixed Timing System-level Embedded Software Modelling and Simulation Approach," in International Conference on Embedded Software and Systems 2009, (ICESS '09), in press, 2009.

[5] J. Madsen and M. Gonzalez, "Abstract RTOS Modelling in SystemC," in NORCHIP Conference, 2002.

[6] P. Hastono, S. Klaus, and S. A. Huss, "Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded Systems," in The International Forum on Specification & Design Languages (FDL'04) Lille, France, 2004, pp. 380-391. .

[7] F. Hessel, V. M. d. Rosa, I. M. Reis, R. Planner, C. A. M. Marcon, and A. A. Susin, "Abstract RTOS Modeling for Embedded Systems," in 15th IEEE International Workshop on Rapid System Prototyping (RSP'04), 2004, pp. 210-216.

[8] H. Posadas, J. Ádamez, P. Sánchez, E. Villar, and F. Blasco, "POSIX Modeling in SystemC," in 2006 conference on Asia South Pacific design automation Yokohama, Japan: ACM Press, 2006.

[9] Z. He, A. Mok, and C. Peng, "Timed RTOS Modeling for Embedded System Design," in 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05), 2005, p. 448.

[10] Y. Yi, D. Kim, and S. Ha, "Fast and Time-Accurate Cosimulation with OS Scheduler Modeling," Design Automation for Embedded Systems, vol. 8, pp. 211-228, June 2003.

[11] I. Bacivarov, S. Yoo, and A. A. Jerraya, "Timed HW-SW cosimulation using native execution of OS and application SW," in 7th IEEE International High-Level Design Validation and Test Workshop, 2002, pp. 51-56.

[12] A. Gerstlauer, H. Yu, and D. D. Gajski, "RTOS Modeling for System Level Design," in Conference on Design, Automation and Test in Europe - Volume 1: IEEE Computer Society, 2003.

[13] H. Yu, A. Gerstlauer, and D. Gajski, "RTOS Scheduling in Transaction Level Models," in 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis Newport Beach, CA, USA: ACM Press, 2003.

[14] J. Madsen, K. Virk, and M. J. Gonzalez, "A SystemC-based Abstract Real-Time Operating System for Multiprocessor Systems-on-Chips," in Multiprocessor Systems-on-Chips, A. A. Jerraya and W. Wolf, Eds. San Francisco, CA: Morgan Kaufmann, 2005, pp. 284-311.

[15] R. L. Moigne, O. Pasquier, and J. P. Calvez, "A Generic RTOS Model for Real-time Systems Simulation with SystemC," in Conference on Design, automation and test in Europe - Volume 3: IEEE Computer Society, 2004.

[16] A. Bouchhima, S. Yoo, and A. Jerraya, "Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model," in Asia and South Pacific Design Automation Conference 2004 (ASP-DAC'04), 2004, pp. 469-474.

[17] J. Chevalier, O. Benny, M. Rondonneau, G. Bois, E. M. Aboulhamid, and F.-R. Boyer, "Space: A Hardware/Software SystemC Modeling Platform Including an RTOS," in Languages for System Specification: Selected Contributions on UML, SystemC, System Verilog, Mixed-Signal Systems, and Property Specification from FDL'03: Kluwer Academic Publishers, 2004, pp. 91-104.

[18] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," The Journal of VLSI Signal Processing, vol. 41, pp. 169-182, 2005.

[19] F. Hessel, V. M. Da Rosa, C. E. Reif, C. Marcon, and T. G. S. Dos Santos, "Scheduling refinement in abstract RTOS models," ACM Transactions on Embedded Computing Systems (TECS), vol. 5, pp. 342-354, 2006.

[20] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder, "RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model," Design Automation for Embedded Systems, vol. 10, pp. 209-227, 2005.

[21] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications: Springer, 1997.

[22] J. W. S. Liu, Real-Time Systems: Prentice Hall, 2000.