

# A Mixed Timing System-level Embedded Software Modelling and Simulation Approach

Ke Yu, Neil C. Audsley

Dept. of Computer Science, University of York

York, UK

Email: {ke, neil}@cs.york.ac.uk

**Abstract**—System-level software modelling and simulation have become important techniques for real-time embedded system early design space exploration. However, the timing accuracy issues have not been solved well in current methods, which produce unrealistic results or large simulation overheads. In this paper, we propose a mixed timing modelling and simulation approach to decouple conventionally inter-dependent software timing modelling and simulation into two separate phases. This approach enables (1) mixed software timing information granularities and annotation methods at the modelling stage for performance and accuracy trade-off (2) good software preemption and hardware interrupt handling timing accuracy at the simulation stage without sacrificing simulation performance (3) varying system run-time status observability and simulation speed for efficiency trade-off. Experiments demonstrate that our approach has flexible simulation performance trade-offs and good simulation timing accuracy. The measured results indicate that hardware interruption and software preemption problems are also solved by our approach.

## I. INTRODUCTION

In recent years, Systems-on-Chip (SoC) has become the state-of-the-art platform for embedded systems, and provides a powerful computation capability for handling complicated real-time concurrent events. Due to ever-increasing complexity, embedded software design has emerged as “the most critical challenge of SoC productivity” [1].

System-level cycle-approximate modelling and simulation, which are based on System Level Design Languages (SLDL) (e.g., SystemC [2] and SpecC [3]), have been proposed as key enablers for validating software (SW) designs early in embedded systems design flow, when the hardware (HW) design is unfinished. Conventionally, there are two major directions of methods: abstract SW modelling and simulation [4] [5] [6] [7] and native SW modelling and simulation [8] [9] [10]. The former simulates coarse-grained timing task models with an abstract RTOS model at very high levels. The latter applies to real SW with relatively accurate time annotations and often uses a real RTOS.

However, there are still some challenges in current methods, which affect the simulation capability, accuracy and performance. An often referred weakness is the “annotation-dependent SW time advance approach” [5] [7] that results in problems in HW/SW synchronization. Referring to Figure 1(A), in a cycle-approximate SW simulation, a SW model executes its function codes on the host in zero target time. The SLDL uninterruptible “wait-for-

delay” time advance mechanism is usually used to simulate SW target-platform delay annotations. Once a “wait(delay)” function is invoked, SW time will be progressed by the value of “delay” without interruption. As a result, the SW model cannot be interrupted (namely preempted) during its delay period, when an interrupt event is raised by a HW model. The interrupt service routine (ISR) is able to start only if the current delay period is finished. We can observe the wrongly postponed  $t_{il}$  in Figure 1(A). Under such circumstances, both SW task switch and HW/SW synchronisation only happen at boundaries of delay annotations. The preemption latency and the interrupt latency are unrealistically restricted by the length of delay annotations. This SW time advance method makes it hard to model a preemptive real-time system or a real interrupt handling procedure. The intuitive but halfway solutions tackle this problem by using more “wait” statements with fine-grained “delays” to advance SW time[8] (Figure 1(B)), or by inserting some pseudo synchronization points [11], which frequently detect interrupts and exchange SW/HW simulators’ clock information. However, the accuracy is limitedly enhanced at the cost of modelling (more annotation and synchronization) and simulation overheads (frequent simulation engine context switch).

In this paper, we propose a mixed timing system-level SW modelling and simulation approach to address the above problem. The central idea is to decouple the conventionally

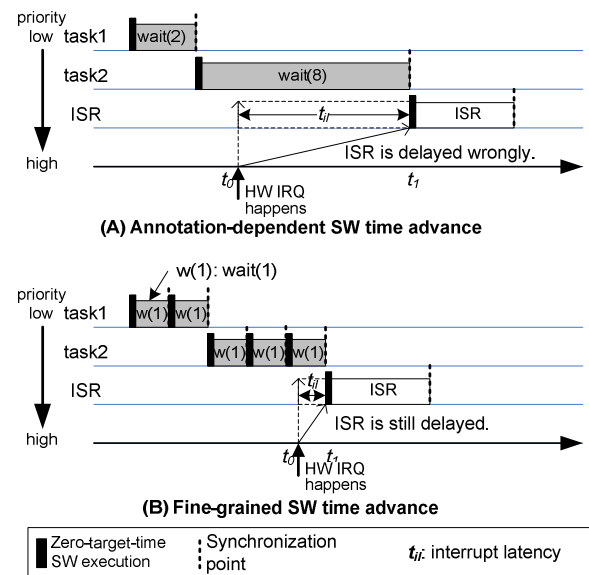


Figure 1. Conventional SW time advance methods

interdependent SW *timing modelling* and *timing simulation* jobs into two separate steps. It bears high flexibility in SW *modelling* and good timing accuracy in SW *simulation* with low simulation overheads. Specifically, the *timing modelling* step refers to annotating target platform execution cost (delays) for SW models; whilst the *timing simulation* step refers to managing (advance and stop) these delays at runtime. Differing from solutions above, in our approach, the SW simulation time advance is completely independent from modelling annotations. In addition, the low preemption latency and interrupt latency are achieved without losing simulation performance.

Furthermore, to assist SW modelling and simulation, we build an abstract transaction-level-modelling (TLM) HW model that includes the CPU, memory, bus, and peripheral modules. This is because although cycle-accurate HW models are not essential for system-level SW modelling, limited TLM HW modelling is still beneficial. Since some RTOS services' functional and timing modelling accuracy is hardware-dependent, such as the context switch service and interrupt handling. The existence of HW models makes the simulation more likely to resemble a full embedded system. Specifically, the CPU model (called the Live CPU Model hereafter) is essential in our mixed timing approach, because it plays a live role in managing SW time advance, just like a real CPU executing SW instructions.

Our modelling and simulation work is implemented with the SystemC SLDL [2] and TLM concept [11]. Since they both are advocated as promising industrial system-level design tools, their popularity ensures our approach will be applied by other embedded systems designers.

The paper is organized as follows. Section II surveys related work. Section III introduces the mixed timing modelling and simulation approach. Section IV describes the abstract HW model and in particular the Live CPU Model. Several experiments are shown in Section V to demonstrate the benefits of our approach. Section VI concludes this paper.

## II. RELATED WORK

A large body of research has been undertaken in the area of system-level real-time embedded SW modelling and simulation. In terms of different timing accuracy and usability, there are two main types: coarse-grained timing abstract SW modelling and simulation, and fine-grained timing native SW modelling and simulation. Although instruction set simulation (ISS) based SW simulation is a widely used cycle-accurate method, it is not quite suitable for early design phases due to its slow simulation speed and high requirement on SW completeness.

The coarse-grained timing abstract SW modelling and simulation focus on very early design phases, such as system specification, system analysis and SW/HW partitioning stages. This usually models SW applications as a collection of tasks with loose timing properties (e.g., period, deadline, worst-case execution time). An abstract and generic RTOS model is usually built to supply basic scheduling and task management services. The SW timing information is either annotated by estimation or randomised by some statistical

theories, e.g., the uniform distribution in [12] and the Gumbel probability density in [5]. The advantage of this method is the fast simulation speed, since applications and RTOS are highly abstract models. The drawback of this method is low timing accuracy (coarse time annotations for applications and inadequate consideration of RTOS timing overheads) and incomplete modelling capability (lack of SW/HW interaction modelling). The native SW modelling and simulation generally model application tasks with functional codes, and port a real RTOS instance. Its timing accuracy is improved, because SW times are annotated at a fine granularity (e.g., block level, source line level, and assembly line level). Its simulation speed is not comparable with abstract simulation, but is still faster than ISS, as reported in [10].

The ARTS project presents abstract SystemC-based SW modelling in [4] and uses it for high-level MPSoC design space exploration in [12]. It abstracts a real-time embedded system through three sub-models: the task graph model, the scheduler model, and the link communication model. It does not model task functionality and lacks RTOS overheads modelling. A similar method is introduced in [6], but this focuses on quickly evaluating different scheduling policies in simulation.

The UCI CECS group introduces SpecC-based abstract RTOS modelling for TLM modelling refinement flow in [7] [13]. It uses an annotation-dependent SW time advance approach. Thus its simulation timing accuracy is not satisfied. A remedy approach "Result Oriented Modelling (ROM)" is presented in [14], which can virtually interrupt the "wait-for-delay" statement by recording preemption information in a task control block. However, it still relies on the SLDL uninterruptible "wait-for-delay" function. So the preempted task may wake up at a wrong time point after its "delay" period is finished. This results in unexpected simulation engine context switch and consequential processing overheads. Our approach is comparable to it, but differs in the following ways: 1) we use a "wait-for-event" mechanism to ensure the preempted task only wakes up on receiving an event at a correct time point. A simulation speed-up, because of less processing overheads on incorrect wake-ups, can be expected; 2) the ROM combines all interrupt service routines' time delays, which have happened during an ongoing "delay" duration, in order to launch a new "delay". But our method processes each ISR's time delay at its respective arrival. Our simulation has the advantage of better similarity with the real execution.

In [15] [16], Posadas et al. propose a POSIX compliant SystemC-based SW modelling approach. Their method belongs to the aforementioned fine-grained annotation technique. Reference [9] presents a timed RTOS simulation tool for a proprietary Texas Instrument DSP/BIOS RTOS. It proposes an event time-stamp prediction method for interrupt modelling, which has the tight requirement that application tasks should report their future synchronization time to the RTOS kernel. Consequently, its usability is restricted. A synchronization time-point prediction method is presented in [17], which also needs to statically analyse SW codes to estimate the next synchronization point.

### III. MIXED TIMING MODELLING AND SIMULATION APPROACH

In a system-level SW simulation context, modelling means a process to describe functional and timing characteristics of target computing components in high level languages. The results are models for simulation purposes. Simulation refers to executing these models on the host PC, in order to validate and analyse functional and timing behaviour of an under-design system. Deep in the consensus of embedded system design is the notion that *timing accuracy* is a first-class factor for determining the accuracy of modelling and simulation. Our mixed timing approach divides this *timing accuracy* problem into two aspects: *the timing accuracy of modelling* and *the timing accuracy of simulation*. Multiple timing granularities and techniques can be applied to them respectively. Figure 2 illustrates the key concept of this approach by an example, in which varying time annotation granularities in modelling do not interfere with SW time advance responsiveness in simulation. This approach has the following specific features:

1) This approach can utilise multiple-grained SW timing information and annotation methods for various sub-models at the modelling stage. It allows the user to build mixed timing models with varying timing scenarios for modelling performance and accuracy trade-off. This idea is comparable to “variable timing synchronization granularities” in [8] and “mixed untimed and timed models” in [11]. But the approach in [8] aims to solve the HW/SW synchronization overhead problem by trading off multiple synchronization granularities. It increases simulation speed but loses synchronization accuracy or vice versa. The approach in [11] mainly applies to the HW modelling area.

2) It preserves high SW preemption and HW interrupt handling timing accuracy within a certain bound at the simulation stage. The simulation performance is also increased, compared to conventional simulation approaches. This is because our SW time advance method is annotation-independent and does not rely on uninterruptible SLDL “wait-for-delay” mechanism. In implementation, the Live CPU Model supervises SW simulation delays and monitors external HW interrupts at the same time. If excluding

possible interrupt disabled cases, such as critical sections, the Live CPU Model can preempt current SW execution (stopping its delay period in practice) as soon as an interrupt is caught, just like the real CPU execution.

3) It offers varying system simulation similarity and run-time information observability. By configuring the Live CPU Simulation Engine with different time advance modes, the users can make trade-offs between the simulation similarity, the information observability and the simulation performance. This is in contrast to the above-mentioned ROM approach which only maintains simulation correctness at state-changing boundaries and cannot show intermediate simulation results to the user.

In the following, our approach is described in detail with regard to *modelling* and *simulation* aspects, respectively.

#### A. Timing accuracy of modelling

In our approach, timing accuracy of modelling relates to various jobs of adding time delays for computing models accurately, including:

##### 1) Assigning sub-models with time annotations

Since we focus on the “timing” behaviour of real-time embedded SW, every SW sub-model that requires the target CPU computation resource needs to be annotated with corresponding target execution time information. In other words, CPU computation costs are represented in this way. Every HW model that can perform individual computation also needs to be annotated with delays. Every action inducing HW inter-module communication will also be annotated with corresponding communication delays.

##### 2) Using varying timing information sources

Akin to other system-level SW simulation, we utilise common profiling techniques “ISS-based measurement” and “RTOS benchmark”, to acquire accurate SW timing information. In the ISS-based approach, we firstly compile high-level source codes for a given processor architecture, and then run them on a cycle-accurate ISS (e.g., the KEIL ARM ISS) in order to measure their execution time cost. RTOS benchmarks can be obtained from vendor’s documents. Additionally, coarse-grained SW timing information can be generated from system specifications and time budget estimates.

##### 3) Applying multiple annotation granularities

Let us firstly recall two possible situations in early embedded systems design phases. Firstly, various applications, RTOS, and HW modules may have different development progress. This means that various components may have varying available function codes and timing information for use in modelling and simulation. Secondly, simulation users may focus on different sub-models in diverse circumstances. For example, computation modelling and communication modelling are two distinct working directions in TLM simulation. It is not only infeasible but also costly to build all sub-models with the same timing level. Hence, a mixed timing modelling method is more efficient for early design space exploration.

In fact, the “multiple timing granularities” method or the similar “multiple timing accuracy levels” concept has been discussed previously. In [18], three timing degrees for TLM

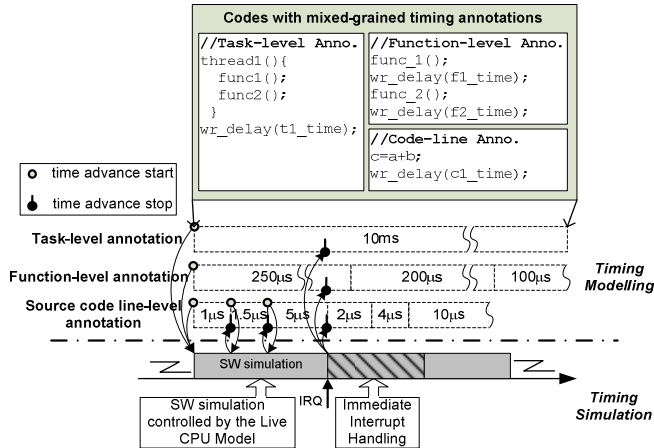


Figure 2. Separating timing accuracy issues of modelling and simulation

computation and communication models are defined: un-timed, approximate-timed, and cycle-timed. Reference [8] proposes multiple timing synchronization granularities: time period, source line-by-line, function-by-function, and instruction-level. Our approach defines several appropriate granularities for system-level SW modelling. For source-code-available SW applications, we apply source code line-level, function-level, and task-level annotations (Refer to Figure 2). For abstract SW models, we apply: function-level and task-level time budget estimates. For abstract RTOS modelling, since these “abstract and generic” sub-models are not the same as the implementation codes for a target system, it is meaningless to use an accurate profiling technique to produce fine-grained timing information. Therefore, we annotate each sub-model with a function-level delay, which represents the “simulated” RTOS service’s timing overheads.

#### 4) Annotating timing information on models

In our approach, the target timing information is annotated on models in two ways:

1) Static annotation is widely used in native SW simulation [10] [14]. The delay annotation is statically inserted after each SW segment (the segment length depending on the selected timing annotation granularity). It has the advantage of low simulation run-time overheads and high accuracy based on real measurement, but it requires huge preliminary profiling work.

2) For abstract SW models that are not suitable for profiling, we use the dynamic annotation approach. If a model contains several blocks (e.g., functions or loops), then it uses a block’s base execution time estimate “ $t_{i0}$ ” with its actual simulation count “ $n_i$ ” to calculate the SW model’s total simulation delay  $t_d$ :  $t_d = \sum_{blocks} t_{i0} * n_i$ . This dynamic

annotation approach is more practical than annotating a constant delay to every execution instance of a SW model regardless of its actual simulation trace.

### B. Timing accuracy of simulation

Timing accuracy of simulation is reflected by accuracy of two basic actions: SW time advance and interrupt handling. Further, timing accuracy of SW time advance depends on two actions’ timing resolutions: the *SW delay progress resolution* and the *SW delay preemption resolution*. The former refers to the minimum step to progress a SW delay, and the latter refers to the latency to stop a delay. Interrupt handling timing accuracy is mainly revealed by the *interrupt latency*, which is the time from asserting a HW interrupt signal until beginning to a SW interrupt handler.

#### 1) Timing accuracy issues in simulation

In the SLDL based system-level simulation, every functional simulation model (including applications, RTOS and HW models) is executed on the host PC. The whole execution is managed by the background SLDL engine (it is the SystemC simulation engine in our context). This execution cannot represent any target timing behaviour (i.e., it is zero-target-time execution), since it is executed on the host. Consequently, the target execution delays need to be annotated on these functional simulation models. The Live CPU Model executes these simulation delays, in order to

mimic target execution timing overheads. Compared with a real CPU as an instruction execution engine, our Live CPU Model can be seen as a “time delay” execution engine.

After a SW model is dispatched by an RTOS scheduler and its functional codes are finished in zero-target-time, the SW delay information will be passed to the Live CPU Model. Then, the Live CPU Model begins to execute SW delay. Specific delay length depends on the information input. We call it the “variable-step” time advance method, because an actual time delay step is not fixed. It is worthwhile indicating that the concerned *SW delay progress resolution* is only restricted by SystemC’s resolution that has *1picosecond* default value. When the Live CPU Model decides to terminate current SW delay, it sends the termination event to the SW model immediately. Consequently, the latency to stop delay time, i.e., *SW delay preemption resolution*, is also zero-time. Because the Live CPU model senses external interrupt request when carrying out SW delay, interrupt handling can also begin immediately. The theoretical minimum *interrupt latency* is zero-time in simulation, and the worst-case *interrupt latency* is bounded by the longest interrupt disabled time. This accurate simulation mechanism is the same as the real CPU execution. The exact implementation will be introduced in the following section.

Simulation-related timing accuracy issues are thus relaxed from delay annotations in modelling. This is because delay annotations are “executed by the Live CPU Model” in an interruptible way rather than are directly used in un-preemptable “wait-for-delay” functions. Figure 2 depicts the separate timing modelling and simulation. An IRQ example shows the high HW/SW synchronization accuracy, no matter what timing annotation granularity is applied.

#### 2) Simulation similarity and speed trade-off

Reference [14] proposes that it is unnecessary to mimic intermediate states in simulation, and only essential to generate correct results at state-changing boundaries. High simulation performance is thus the primary goal. Our above mentioned variable-step time advance technique also generally subscribes to this point of view. For example, if a SW time delay is given as *10ms*, assuming the delay duration is not interrupted, the Live CPU Model will execute *10ms* time delay in one delay duration. The system simulation clock jumps from  $t$  to  $t+10ms$  in a single step along the time line. The decrement of the execution budget is not simulated step-by-step and hence cannot be observed by the simulation user. From the perspective of simulation result validity at specified synchronization points, there is no problem.

However, from the perspective of debugging real-time embedded SW and tracing system-wide variables status, the simulation users may not be satisfied by observing limited or outdated information at an observation point. We thus propose the “fixed-step” time advance method to update system-wide variables more frequently than to do it only at state-changing points. In the fixed-step mode, the Live CPU Model runs periodically to handle run-time changing variables, e.g., timers, time delay values, execution budgets etc. The Live CPU Model can blend “variable-step” and “fixed-step” methods in simulation in order to trade off simulation similarity with simulation speed.

#### IV. ABSTRACT TLM HARDWARE MODELLING

To undertake accurate system-level embedded SW modelling and simulation it is necessary to model and simulate the underlying hardware architecture. Because many RTOS services are hardware-dependent, such as context switch, interrupt service, and timer service, it could be difficult to model HW/SW interactions accurately without support from a hardware model on which SW are assumed to run. Moreover, one-sided SW modelling is against the system-level embedded systems HW/SW co-design principle. Many studies have suggested using transaction level models for high-level system modelling and simulation. We also build an embedded system abstract TLM HW model to assist mixed timing SW modelling and simulation. Figure 3 depicts its structure and the HW/SW interactions. It includes several types of HW modules: the clock generator module, the Live CPU module, the bus module, the memory module, the peripheral and device module. The HW resource (e.g., processor, bus and memory block) sharing and contention can be taken into account for accurate SW modelling and simulation. SystemC-TLM communication methods and an abstract bus offer intra- and inter-communication for the modules. The bus model is developed from a standard example in SystemC distribution [2]. It supports blocking and non-blocking communication between prioritised master modules (CPU) and slave modules (memory, devices, and peripheral). In this paper, we only focus on the Live CPU model, which is the core of our hardware modelling and vital for SW simulation.

We decompose the Live CPU Model into three essential components for SW simulation: 1) the Live CPU Simulation Engine taking charge of SW time advance 2) the Register Set assisting context switch and flags setting 3) the Interrupt Controller monitoring interrupts. By these sub-models, the CPU model is actively involved in high-level SW simulation.

In the following, we introduce our SW time advance mechanism based on the use of Live CPU Simulation Engine by referring to Figure 4:

Step (A): a SW code block (no matter if it is an application task, a function, a code line or an RTOS service) executes in zero-target-time at time  $t_0$ . From the perspective of OS scheduling, the SW code block is at the running state,

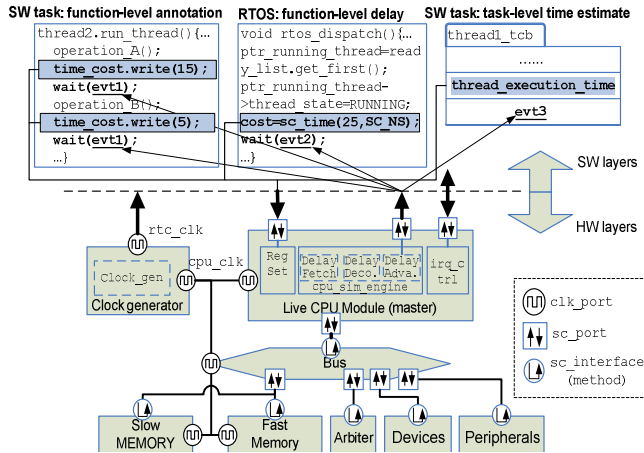


Figure 3. TLM hardware model for SW simulation

i.e., in occupation of the CPU.

Step (B): After it is finished, the SW code block's delay context (delay information) is loaded into Live CPU Model's registers for a preparation of time advance. At the same time, the SW code block keeps waiting for its exclusive SystemC event (*sc\_event*) that will be sent by the Live CPU Simulation Engine in the future. This event represents the "address of code block to run" in simulation and is stored in the *program counter (PC)* of the Live CPU Model, just like the use of *PC* in a real CPU.

Step(C): Inspired by the instruction execution mechanism (fetching, decoding, and performing) of a real CPU, our Live CPU Simulation Engine also takes corresponding steps to execute a SW delay: 1) delay information fetching, 2) delay information decoding, and 3) delay advancing. Depending on different time annotation methods, the delay information input can be fetched from various sources, including simulation framework global variables, a SW task control block, or the return value of a delay writing function. Afterwards the engine decodes acquired delay information into standard-form data  $t$ , i.e., a double float number with  $\mu s$  unit. The decoded result  $t$  is stored in the *delay register (DR)* of the Live CPU Model, which resembles the *instruction register* in a real CPU.

Step (D): Finally, the "delay advancing" starts at time  $t_0$ . If the Live CPU Simulation Engine works in a pure variable-step mode, it plans to consume the delay in a single advance step and release the event in *PC* at time  $t_0+t$  by SystemC event timed notification mechanism. If the Live CPU Simulation Engine is in a fixed-step mode, it runs periodically to decrement and update the delay value in *DR* until it is exhausted; then it releases the event in *PC*. If the Live CPU Simulation Engine works in both variable-step and fixed-step modes, it will do both.

Step (E): If there is no interruption or preemption during this  $t$  time delay. Thus, at time  $t_0+t$ , the value in the *delay register* is exhausted and the projected event is released as planned. The waiting SW code block gains CPU again to continue execution. As well, an interrupt may happen during the ongoing delay duration, such as the IRQ event at  $t_1$  in Figure 4. Given that interrupts are not disabled, the Interrupt Controller Model catches this IRQ immediately and invokes

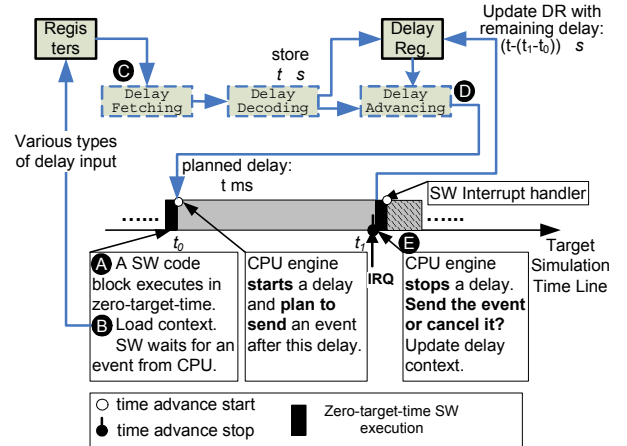


Figure 4. SW simulation controlled by the Live CPU simulation engine

the Live CPU Simulation Engine to handle this IRQ. The Engine firstly cancels the previously planned SystemC event; then updates the *delay register* by calculating the remaining value:  $t - (t_i - t_0)$ ; finally saves the value into SW context control block. After that, a SW interrupt handler is called to continue the IRQ handling process. This SW interrupt handler executes its functions and gets time advance service by repeating the above process. In this way, both SW time advance and HW interruption is simulated accurately.

In conventional HW/SW co-simulation, individual clocks are used for different HW and SW simulators respectively. They need to compare the HW and SW simulators' local clocks for global time synchronisation, which bring undesirable simulation overheads. However, there is only one unified HW/SW clock in our simulation and this is synchronous to the SystemC simulation clock. The SW models do not need a local clock for timed advance. The Live CPU Simulation Engine uniformly and serially carries out SW time advance jobs. This approach offers good timing accuracy in simulation, whilst it brings two additional advantages: firstly, the Live CPU Engine's execution mechanism is similar to a real CPU and is thus straightforward to understand; secondly, it generates a CPU simulation trace similar to a real execution.

In addition to above-mentioned *program counter* and *delay register*, the Live CPU Model contains other conceptual general-purpose registers as well as some special-purpose status registers to assist SW simulation. These registers stores SW delay information and system status information. When a context switch is invoked, current CPU general-purpose registers' contents (storing SW delay information) are saved in the pre-empted task control block (TCB); and the newly dispatched task's TCB context is loaded into these registers. By default, we configure and name the register set by partial reference to ARM processor register scheme [19]. Simulation users can tailor the register set according to different presumed CPU models.

It is acknowledged that the *interrupt latency*, *interrupt response time*, and *interrupt recovery time* are some keen-interested timing properties of a real-time embedded system. The Interrupt Controller Model provides a good foundation to model the usual HW/SW cooperative interrupt handling mechanism, which usually has three bottom-up layers: the HW interrupt controller, the RTOS interrupt handler, and ISRs. The Interrupt Controller Model always watches several *sc\_ports*, which are connected with IRQ lines (some SystemC signal channels). In order to deal with multiple simultaneous interrupts from various devices and bound the *interrupt latency*, the interrupt controller can prioritise, mask or disable interrupt sources by setting corresponding register bits. When a hardware device raises an IRQ by asserting a signal on its interrupt request line, the Interrupt Controller Model can catch the signal immediately and calls the Live CPU Simulation Engine to stop the current delay process. At the same time, it sets the *PC* with an event belonged to a SW interrupt handler. Subsequently, depending on a specific interrupt handling scheme, either a vectored ISR or an RTOS kernel-level interrupt handler function will begin.

## V. EXPERIMENTAL RESULTS

In this section, we take several experiments to demonstrate the benefits of our mixed timing RTOS modelling and simulation approach. All simulations are performed with SystemC v2.2 on a 2.2GHz PC. Host simulation times are measured by Windows high-resolution ( $\mu s$  level) *QueryPerformanceCounter()* function which can retrieve the hardware cycle counter.

### A. Simulation Engine Modes Comparison

To demonstrate the flexible simulation performance trade-off ability and the good SW time advance accuracy of the mixed timing method, we evaluate multiple time advance methods supported by our method. The focus is to evaluate the relationship among simulation speed, simulation observability, and SW time advance accuracy under different configurations of the Live CPU Simulation Engine. The applications consist of eight tasks (four periodic tasks and four sporadic tasks) with task-level delay annotations. We apply a time-driven and event-driven combined scheduling mechanism with the round-robin algorithm. Eight interrupt sources are included in the simulation and fires randomly to trigger sporadic tasks. The simulation runs *1000ms* target time that allows a task to repeat at least 20 jobs.

Live CPU Simulation Engine Mode A: This uses a fixed-step time advance approach, which runs every microsecond to advance  $1\mu s$  SW time. It is similar to the fin-grained time period synchronization approach in [8]. This can achieve  $1\mu s$  SW time advance resolution.

Live CPU Simulation Engine Mode B: This is a dual-grained fixed-step time advance approach. When a SW annotation slice is greater than *1ms*, the engine runs every millisecond to progress SW time by a fixed value *1ms*. Once the annotation slice falls below *1ms*, then the engine runs every microsecond to advance remaining SW time by a fixed value  $1\mu s$ . It can achieve  $1\mu s$  SW time advance resolution.

Live CPU Simulation Engine Mode C: The Engine uses a mixed fixed-step and variable-step time advance mode. It progresses a full delay annotation slice in an interruptible variable-length step, and it runs every millisecond to update simulation status information. The SW time advance resolution is only restricted by the timing resolution of SystemC simulation engine.

Live CPU Simulation Engine Mode D: This is a pure variable-step time advance mode. The engine progresses the SW annotation slice in an interruptible variable-length step. The SW time advance resolution is only restricted by the timing resolution of SystemC simulation engine.

In order to compare the simulation speed of our proposed approach with the simulation speed of a conventional ISS, the same applications are run using KEIL  $\mu$ Vision ARM ISS for duration of *1000ms*. The target processor is set as a NXP LPC2378 running at 40MHZ. We port a  $\mu C/OS-II$  RTOS on this ISS to manage tasks.

The obtained results are shown in Figure 5. Compared with the ISS simulation, unsurprisingly, our approach obtains drastic performance improvement. The results also indicate that the fixed-step time advance approach generally costs

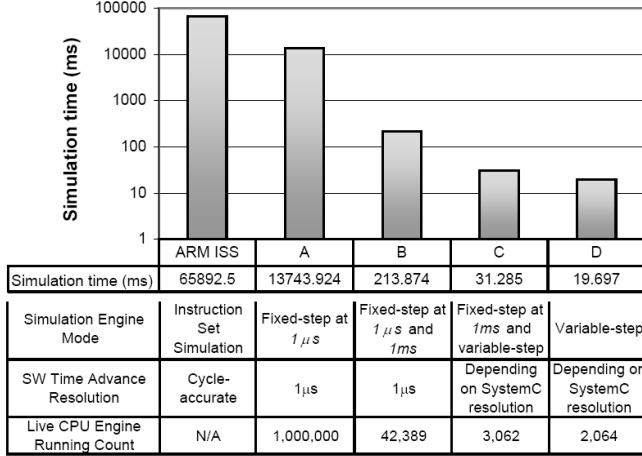


Figure 5. Comparison of different simulation engines much more simulation time than variable-step approach: Engine D achieves a considerable speed-up (up to 676 times) to that achieved by the fine-grained ( $1\mu s$ ) fixed-step Engine A, when they both have a  $1\mu s$  time SW time advance resolution. This is because the fixed-step approach progresses simulation time and updates status variables more frequently than the variable-step approach: the running count of Engine A is 484 times of Engine D's.

The Engines B and C are two compromise methods when compared with the fine-grained fixed-step Engine A and the pure variable-step Engine D. The measured simulation results show that the Engine C can still save more simulation time than Engine B (6.8 times speed-up). For its moderate simulation overhead and system information tracing ability, Engine C's combined fixed-step and variable-step mode is thus acclaimed in our simulation approach.

We also note that there is still 37% simulation time difference between the Engine C and Engine D. This is because the former runs about 1000 more times (once a millisecond for a  $1000ms$  duration) in its fixed-step mode; and this higher Engine running count directly slows down simulation performance. In order to reveal the correlation between the step length and the simulation speed in fixed-step mode, we test three other step granularities:  $2ms$ ,  $5ms$  and  $10ms$ , which means the Live CPU Simulation Engine

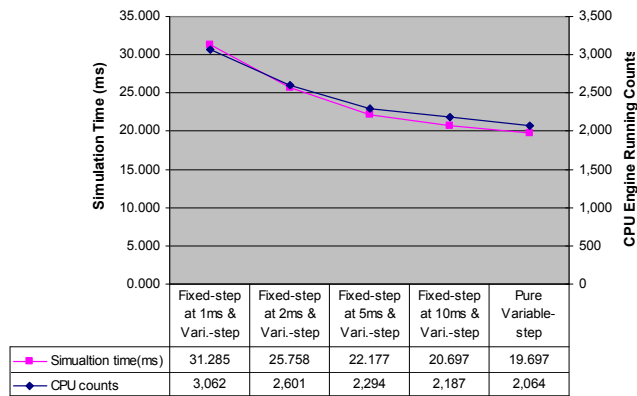


Figure 6. Comparison of simulation speed with varying fixed-step length

runs every  $2ms$ ,  $5ms$ , and  $10ms$  respectively. The Figure 6 shows that the simulation time and engine running count are steadily decreasing whilst the fixed-step granularity is growing. This characteristic can be used to tune the Live CPU Simulation Engine and thus to optimise the simulation performance and simulation observability in different situations.

In summary, the highly configurable Live CPU Simulation Engine leaves adequate space for simulation users to trade off simulation time and accuracy, depending on their differing intents.

### B. Interrupt Handling Simulation

In order to evaluate the preemption and interrupt handling ability of our approach, we make an interrupt experiment, which supports a timely, nested, prioritised, and maskable interrupt handling mechanism with low and bounded interrupt latency. It includes five IRQs (IRQ1-5) and five associated ISRs (ISR1-5), which are assigned ascending priorities (higher than any applications' priority).

The Figure 7 shows a time-line segment of this experiment. Initially, at  $t=7011\mu s$ , IRQ2 and IRQ3 happen simultaneously. Since the Live CPU model controls SW time advance and monitors IRQ lines, ongoing SW time advance can be stopped as soon as possible, i.e., the IRQ is handled immediately. The theoretical *interrupt latency* ( $t_{il}$ ) is zero in simulation. Note that, to make our model more practical, we may also assign  $t_{il}$  with some tiny delays to mimic the case that a CPU needs to finish current instruction execution before it handles the interrupt. Because IRQ3's priority is higher than IRQ2's, then the interrupt controller ignores IRQ2 and begins to service IRQ3. Afterwards, RTOS services (*context switch* and *interrupt handler enter*) and ISR3 execute sequentially. At  $t=7022\mu s$ , a higher-priority IRQ4 happens and it then gets nested service by pre-empting ISR3. The lower-priority IRQ1 that fires during ISR4 execution is ignored by the interrupt controller, as it has been masked by the preceding interrupt handler. After the finish of ISR4, the lower-priority IRQs that have taken placed are handled successively.

In order to quantify timing accuracy of interrupt handling, we measure the *interrupt latency* of these five IRQs. Each IRQ randomly fires 500 times in 10 seconds

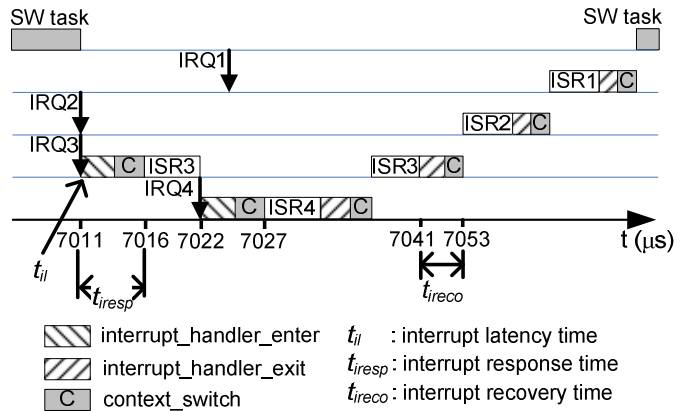


Figure 7. Interrupt handling experiment

simulation. The focus is to evaluate the experimental *interrupt latency* with the theoretical results. In order to eliminate varying RTOS critical section's interference on *interrupt latency*, we assume interrupts are always enabled and the Live CPU Simulation Engine can stop SW execution as soon as a higher-priority interrupt happens. Therefore, at any simulation time point, the highest-priority IRQ's *interrupt latency* should always be zero, and all IRQs are only able to be postponed by higher-priority ISRs. Therefore, the theoretical maximum (worst-case) *interrupt latency* of an IRQ can be computed as the sum of all higher-priority ISR time cost:  $t_{il\_max} = \sum_{high\_prio} t_{ISR}$ .

TABLE I. compares the measured maximum *interrupt latency* with calculated theoretical results. As expected, in most cases, when the happening IRQ is the highest-priority, it is serviced without any delay (i.e., zero-time latency). In case it is delayed by some other higher-priority ISRs, its maximum *interrupt latency* does not exceed the theoretical worst-case value either.

TABLE I. COMPARISON OF THEORETICAL AND EXPERIMENTAL INTERRUPT LATENCY

	Zero-time IRQ Latency Times	Delayed IRQ Latency Times	ISR time cost ( $\mu s$ )	Theoretical Max IRQ Latency ( $\mu s$ )	Measured Max IRQ Latency ( $\mu s$ )
IRQ5	500	0	500	0	0
IRQ4	441	59	10	500	494
IRQ3	440	60	10	510	488
IRQ2	448	52	10	520	502
IRQ1	444	56	10	530	488

## VI. CONCLUSION AND FUTURE WORK

In this paper, we have presented a mixed timing system-level cycle-approximate embedded software modelling and simulation approach. By isolating the *timing modelling* problem from the *timing simulation* problem, it can not only integrate multiple-level timing models for the sake of accuracy and efficiency at the *modelling* stage, but can also achieve a good SW time advance and interrupt handling accuracy at the *simulation* stage. The hardware TLM model presents a clear and essential high-level HW architecture abstraction for assisting SW simulation. Especially, the HW/SW synchronization problem is tackled by the Live CPU model, which incurs much less overhead than conventional fine-grained annotation and synchronization approach. Additionally, the Live CPU model supports multiple execution modes, which could trade off the speed of simulation with the observability of simulation process. Through comparison with  $\mu$ Vision ARM ISS simulator, the proposed approach improves simulation performance up to three orders of magnitude.

The remaining issue of our research is to implement the mixed timing modelling and simulation approach in more complex embedded SW simulation that includes a full functional RTOS model. We also intend to apply it for

multiprocessor systems. Favourably, the modular hardware TLM models have provided such potential.

## REFERENCES

- [1] A. A. Jerraya, S. Yoo, D. Verkest, and N. Wehn, *Embedded Software for SoC*: Kluwer Academic Publishers, 2004.
- [2] OSCI, "SystemC." <http://www.systemc.org/>.
- [3] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, 1st ed: Kluwer Academic Pub, 2000.
- [4] J. Madsen and M. Gonzalez, "Abstract RTOS Modelling in SystemC," in *NORCHIP Conference*, 2002.
- [5] P. Hastono, S. Klaus, and S. A. Huss, "Real-Time Operating System Services for Realistic SystemC Simulation Models of Embedded Systems," presented at The International Forum on Specification & Design Languages (FDL'04) Lille, France, 2004.
- [6] 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 *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping (RSP'04)*: IEEE Computer Society, 2004.
- [7] A. Gerstlauer, H. Yu, and D. D. Gajski, "RTOS Modeling for System Level Design," in *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*: IEEE Computer Society, 2003.
- [8] I. Bacivarov, S. Yoo, and A. A. Jerraya, "Timed HW-SW cosimulation using native execution of OS and application SW," presented at High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International, 2002.
- [9] Z. He, A. Mok, and C. Peng, "Timed RTOS Modeling for Embedded System Design," in *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*, 2005, pp. 448.
- [10] I. Bacivarov, A. Bouchhima, S. Yoo, and A. A. Jerraya, "ChronoSym: a new approach for fast and accurate SoC cosimulation," *International Journal of Embedded Systems* vol. 1, pp. 103 - 111 2005.
- [11] F. Ghenassia, *Transaction Level Modeling with SystemC: TLM Concepts and Application for Embedded Systems*: Springer, 2005.
- [12] 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, The Morgan Kaufmann Series in Systems on Silicon*, A. A. Jerraya and W. Wolf, Eds. San Francisco, CA: Morgan Kaufmann, 2005, pp. 284-311.
- [13] H. Yu, A. Gerstlauer, and D. Gajski, "RTOS Scheduling in Transaction Level Models," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. Newport Beach, CA, USA: ACM Press, 2003.
- [14] G. Schirmer 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.
- [15] 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.
- [16] H. Posadas, E. Villar, F. Blasco, R. D. Ds, P. Tecnológico, and S. Paterna, "Real-Time Operating System modeling in SystemC for HW/SW co-simulation," presented at Proceedings of Conference on Design of Circuits and Integrated Systems, IST, Lisbon, 2005.
- [17] J. Jung, S. Yoo, and K. Choi, "Fast cycle-approximate MPSoC simulation based on synchronization time-point prediction," *Design Automation for Embedded Systems*, vol. 11, pp. 223-247, 2007.
- [18] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. Newport Beach, CA, USA: ACM Press, 2003.
- [19] A. N. Sloss, D. Symes, and C. Wright, *ARM System Developer's Guide: Designing and Optimizing System Software*: Morgan Kaufmann, 2004.