Combining Behavioural Real-time Software Modelling with the OSCI TLM-2.0 Communication Standard

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

***Abstract*—Transaction Level Modelling (TLM) is an emerging design approach to accelerate Electronic System Level (ESL) design. A virtual TLM prototype of an embedded system is an integration of computation and communication. Currently, TLM communication and hardware modelling has been well discussed and standardised. However, there still exist problems in the domain of TLM for software computation modelling and simulation. In this paper, we aim to propose some appropriate real-time software models from the perspective of TLM software modelling. They are compatible with current TLM modelling concepts and able to be combined with existing TLM communication models. In addition, we implement a software Processing Element (PE) model which effectively integrates mixed timing RTOS-centric software models, abstract processor hardware functions, and OSCI TLM-2.0 communication interfaces.**

I. INTRODUCTION

In the recent past, Transaction-Level Modelling (TLM) has been generally considered as a promising system-level modelling paradigm to improve productivity in the design of highly integrated embedded systems. TLM models are expected to serve as interoperable references across different design teams with different aims such as fast embedded systems architecture exploration, early embedded software (SW) development, and functional verification. Gaining both academic interest and industrial acceptance, SystemC language is the most established System-Level Design Language (SLDL) vehicle for TLM approaches today [1]. Based on the essential TLM principle "separation of *computation* from *communication*", TLM research can be divided into two aspects: the *computation* aspect and the *communication* aspect [2]. In this paper, we focus on the *software computation* modelling aspect, and we address the integration of *software computation* models and *communication* models in a TLM context.

From the hardware developer's point of view, TLM captures embedded systems' hardware architecture at a range of abstraction levels higher than the traditional Register Transfer Level, speeding up the simulation performance by orders of magnitude [1]. In TLM, an embedded system is broken down to a set of computation components comprising concurrent processes to implement application functions. Computation components communicate with each other by sending and receiving transaction requests through *ports* and *channels*. There exists a substantial body of TLM literature discussing various topics, e.g., abstraction levels [2] [3], coding standards (e.g., Open SystemC Initiative "OSCI" TLM [4]), and communication exploration [5] etc.

Embedded software development on a TLM model is not a new topic. In 2002, Pasricha considered the development of embedded software with SystemC TLM models [6] and subsequent efforts have been made to combine conventional cycle-accurate software simulation (e.g., using an Instruction Set Simulator (ISS)) with SystemC-based abstract TLM hardware and communication models [7] [8]. These approaches reflect a hardware and communication centric TLM research perspective, given that they do not innovate SW modelling and simulation, but utilise TLM techniques for modelling SW/HW interfaces and hardware components. However, a slow SW simulation speed and a high code completion requirement are two of the main characteristics of this kind of SW simulation method which may reduce the advantages of the TLM models. A key issue lies in exploring some more appropriate SW modelling and simulation methods in the context of TLM modelling in the early design phases.

In order to speed-up simulation performance and validate real-time embedded software early in a system-level design flow, researchers have considered SLDL-based behavioural software simulation. They usually integrate an abstract and generic RTOS model in order to supervise application SW simulation, which is known as RTOS modelling research [9] [10] [11]. Nevertheless, there still exist some problems in this area, which affect models' functionality, timing accuracy, and simulation performance. For example, current high-level RTOS modelling research does not consider integrating convenient TLM communication interfaces, nor extensibility for HW/SW co-simulation. In classical RTOS modelling approaches, multiple concurrent tasks together with an RTOS model constitute a software *processing element* (PE), and hardware abstraction is implicit in such a high-level RTOS-centric PE model.

This paper attempts to define appropriate **TLM real-time embedded software modelling styles** to accompany existing TLM coding styles. It advocates SW computation modelling as an integral part of overall TLM modelling research. It also proposes a **software PE model** that combines mixed timing **RTOS-centric software models** and **OSCI TLM-2.0** communication interfaces, where the former component bears accuracy and functional benefits compared with related RTOS models, and the latter is a de facto TLM communication modelling standard. As shown in Figure 1, this model

can be seen as a mixture of three parts: behavioural software simulation (the software modelling aspect), HW abstraction (the abstract hardware processor modelling aspect), and TLM interfaces (the SW/HW inter-module communication modelling aspect).

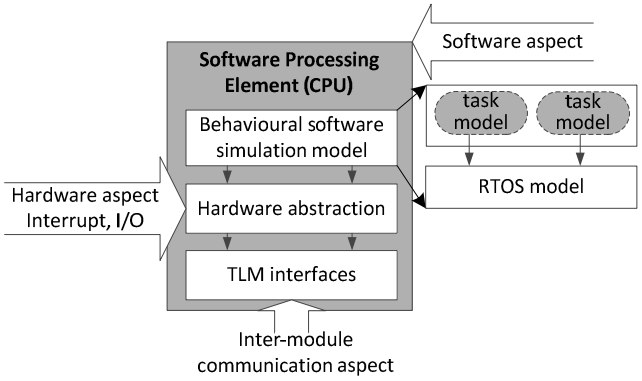

Figure 1. Software Processing Element Model

This paper is organised as follows. Section II reviews the related work. Section III defines software abstraction levels for TLM computation modelling. Section IV describes RTOS-centric real-time software models. Section V introduces the hardware aspect of the software PE model and discussed how to integrate the TLM interfaces. The model's capability and performance is demonstrated in experiments in Section VI and finally, conclusions are made in Section VII.

II. Related Work

OSCI TLM defines two popular abstraction levels for communication modelling based on criteria such as transmission methods and timing granularities, i.e., *Programmers View* (PV) and *Programmers View Timed* (PVT) [4]. Baklouti et al. directly apply the PV and PVT concepts to refine software communication [12]. As illustrated in Figure 2 (A), their method focuses on using TLM synchronous and asynchronous interfaces for abstract SW inter-module (among tasks, RTOS and drivers) communication services. Dömer defines TLM computation abstraction levels based on the concept of the separation of functionality and timing [13]. Four levels are identified in a modelling flow targeting

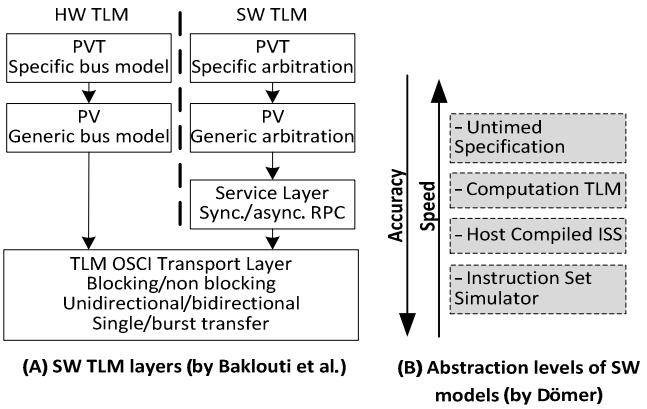

Figure 2. Related SW modelling abstraction level definitions

software on programmable processors (referring to Figure 2 (B)). This work does not specifically distinguish various TLM abstraction levels. In [14], Schirner et al. develop a high-level processor model to support software simulation as a complement to TLM communication modelling. The abstract processor model is modelled in a layered approach including five increasing feature levels, i.e., the application level, the task scheduling level, the firmware level, the processor TLM level, and the processor bus functional level. It enables incremental and flexible description of the software subsystem at different design stages, but RTOS modelling is not introduced in detail.

Some design flows have been proposed to support embedded software generation and synthesis from SystemC specification models and TLM models [15] [16] [17]. They reveal a system-level design point of view and make a valuable contribution to co-design and co-synthesis practices. However, an obstacle may reside in transforming SLDL-based specification models into RTOS-based software execution models. Similar to [12], the hardware-style TLM channel communication mechanism is not suitable for real-time software design. Moreover, it is known that the SystemC library lacks a priority assignment mechanism and preemptive scheduling, so the built-in SystemC scheduling kernel and synchronisation primitive channels are not applicable for real-time software modelling.

Building an RTOS model on top of a SLDL has been proposed to supply necessary dynamic real-time software services for behavioural software simulation. Nevertheless, research in this area has not been widely integrated with TLM communication modelling studies. Gerstlauer et al. introduce a SpecC-based abstract RTOS model in the TLM model refinement flow [18]. However, this RTOS model provides limited primitives for task management, and the RTOS timing overhead is not mentioned. Besides, it uses the imperfect annotation-dependent time advance method, so interrupt handling cannot be accurately modelled and the timing accuracy is limited by the minimal resolution of time annotations. Chung et al. describe a generic SystemC-based RTOS model oriented to MPSoC simulation [19]. Its generic RTOS and POSIX like API models support native application codes to execute with RTL/TLM HW models. However, its RTOS task machine model is simplistic and falls short of modelling real-time synchronisation mechanisms.

III. Defining TLM SW Computation Models

Since we are concerned about modelling real-time embedded software in the context of TLM research, before presenting any detailed modelling methods, it is necessary to clarify TLM SW computation models and the relationship between these SW models and existing TLM communication models.

In our previous work about system-level software modelling [20], we classified system-level embedded SW modelling works into two general types depending on their timing accuracy and usability: the coarse-grained timed **abstract SW model**, and the fine-grained timed **native-code SW model**. We proposed a SystemC-based mixed timing modelling and simulation approach that is able to support com-

pound abstract models and native-code models in one simulation for the sake of flexible trading off accuracy and performance issues. But in [20], we did not consider the connection with TLM modelling of communication, nor defined TLM software computation abstraction levels in detail.

Regarding the TLM communication modelling abstraction level definition, the latest OSCI TLM-2.0 [4] modelling standard is selected as the reference. It defines two coding styles for bus-based communication modelling, i.e., the Loosely-Timed style for PV models and the Approximately-Timed style for PVT models.

The question is: is it possible to integrate our SW abstraction models to TLM modelling concepts for communication? Inspired by the idea of separating the functionality and timing issues of TLM models, we explore the answer by comparing characteristics of our mixed timing software modelling method and the OSCI communication modelling method as follows (see Figure 3):

- Both modelling approaches decompose a model's functionality into several basic entities, i.e., *tasks* (or finer-grained *functions*) for SW modelling in our approach, and *transactions* with corresponding transport functions for TLM communication modelling. If there is a further necessity for more accurate modelling, then a basic entity can be divided into some finer-grained entities, i.e., multiple *functions* inside a task or multiple *basic blocks* inside a function, and corresponding multiple *phases* during a transaction's transmission life.
- We define two comparable timing abstraction levels for models. The Coarse-Grained timed level and the Fine-Grained timed level for SW modelling are comparable to the Loosely-Timed (LT) style and the Approximately-Timed (AT) style for TLM communication. We propose that the coarse-grained timed level uses two time points to represent the execution cost of a task or a function, i.e., the beginning and the end of execution. The loosely-timed coding style also defines two timing points for each transaction to denote calling to and returning from the transmission respectively. Accordingly, the concept of our fine-grained timed level is also parallel to the OSCI Approximately-Timed communication coding style, because they both use multiple timing points inside a basic functional unit, namely, multiple annotations and synchronisation points.
- Besides, both the *untimed* timing level and the *cycle-accurate* level are not recommended in either our SW modelling or OSCI TLM-2.0 standard. This is because modelling real-time software and contemporary bus systems apparently needs a timing concept.

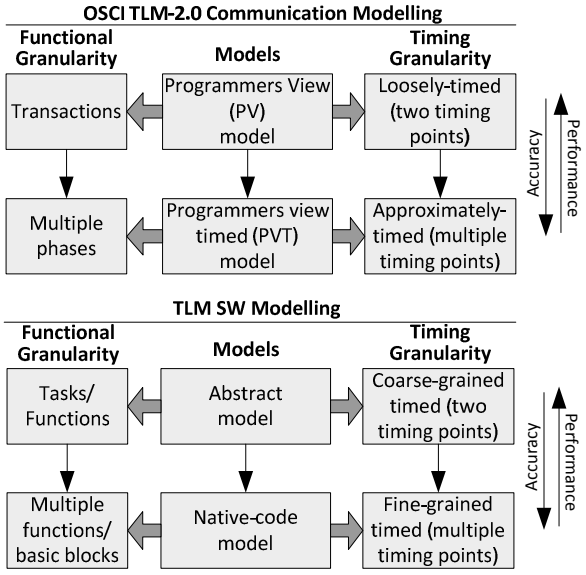

Figure 3. Comparing mixed timing SW models with OSCI TLM-2.0 models

Based on the above comparison, our mixed timing software modelling has some similarity to the OSCI TLM-2.0 communication modelling standard, that is, in terms of modelling concepts about timing granularity and functional granularity. Since they are both implemented in the SystemC simulation environment, they also carry similar changing trends in terms of modelling accuracy and simulation performance, meaning that models at a corresponding level are "harmonious" to each other without resulting in undesired extreme results in a co-simulation.

A basic assumption of our TLM SW computation research is that our models are applicable after initial HW/SW partitioning. Therefore, we address the need to include a generic and configurable RTOS model to facilitate real-time software simulation early. Note that we do not recommend using TLM communication techniques in software modelling, since they are not common methods in the conventional development of real-time software. This idea is contrary to the literature [12] that uses OSCI TLM communication services for joint HW and SW communication exploration.

Specifically, we define two TLM software models as below.

A. Abstract software model

The underlying assumption of abstract software models is that they are usually applied at the early design phases for real-time timing analysis and fast simulation. At this point, the target platform is undetermined and the SW code has not been implemented. Consequently, they do not contain much implementation code or only contain some functional specification code. Also, corresponding timing information of running code on a target platform cannot be measured with high precision for use in modelling.

SW applications are normally organised as a collection of SLDL-based abstract tasks associated with coarse-grained temporal properties, e.g., period, deadline, offset, and execution times. Periodic execution should be explicitly supported by a generic RTOS model that supplies basic real-time software services. The timing overhead of the RTOS model is considered by estimation.

We propose to use task-level (Figure 4(A)) and function-level (Figure 4 (B)) time estimates. These time estimates (annotations) can either be given as a fixed value that represents the Worst Case Execution Time (WCET) at the model

```
void task(){
  while(1){
  //No functional code
  //or limited code
  DELAY(value);//Anno.
  wait();//Sync. point
  }
}
```

```
void func()
{
  //No functional code
  //or limited code
  DELAY(value);//Anno.
  wait();//Sync. point
}
```

(A) Task-level time annotation **(B) Function-level time annotation**

Figure 4. Abstract software models

building stage, or be randomised between a lower bound and an upper bound, which represents the Best Case Execution Time (BCET) and the WCET respectively. The value is passed to the Live CPU Model [20] by the `DELAY()` function as the normal execution cost of a task or a function, i.e., without interruptions.

A synchronisation timing point (e.g., calling a `wait()`) must be explicitly defined after a `DELAY()` function, in order to yield control of the co-operative SLDL simulation kernel and thus allow for multi-tasking real-time software simulation. Calling and returning from the `wait()` function marks the beginning and end timing points of a model's execution duration in the target timeline. An abstract software model is pre-emptive at any implicit timing point during its `DELAY()` time interval in order to process any asynchronous interrupt events in our modelling. The pre-emption modelling technique also relies on the Live CPU Model.

B. Native-code software model

Embedded software (including both applications and RTOS) is organised (wrapped) into several concurrent SLDL processes, and these processes can be further divided into atomic code segments. These processes natively execute in a SLDL simulation on the host computer. Application software model processes are supervised by the RTOS model, which may mimic the specific services of a real RTOS and is endued with corresponding timing delay information.

Timing accuracy becomes a major concern in native-code software modelling and simulation. The desired target timing behaviour cannot be directly represented in native-code software execution. Therefore software execution costs (time delays) of the target platform need to be either statically estimated by a WCET analysis or dynamically measured in an ISS simulation approach, before they are annotated to corresponding code fragments of SLDL modelling processes. Fine-grained function-level annotations (Figure 5 (A)) and basic block level annotations (Figure 5 (B)) can be applied in this kind of SW TLM model. They reveal different timing

```
void func1()
{
  code_segment1;
  code_segment2;
  code_segment3;
  DELAY(value);//Anno.
  wait();//Sync. point
}
```

(A) Function-level time annotation

```
void func1(){
  DELAY(t_b1);//Anno.
  wait();//Sync. point
  int temp = 0;                 Block 1
  if(condition){
      temp++;                   Block 2
      DELAY(t_b2);
      wait();
  }
}
```

(B) Basic block-level time annotation

Figure 5. Native-code software models

accuracy levels depending on choices taken by simulation users. A fine-grained time annotation can improve timing accuracy in case there are data-dependent loops in the code, but may also decrease simulation speed. Note that in Figure 5 (B), a basic block's annotation may inhabit two possible places, i.e., *before* a source code block and *after* a source code block. In modelling, the intention of this difference is to "glue" the code block with its annotation statement as close as possible.

Multiple synchronisation points are inserted in models. Their respective behaviour is the same as the previously mentioned abstract software models.

IV. RTOS MODELLING FOR TLM COMPUTATION

We propose using a unified generic and configurable RTOS model for transaction level software computation modelling. This model is largely based on our previous work in [21]. Figure 6 shows an overview of the structure of the software model that includes multiple application software and RTOS modelling layers.

This model explicitly supports simulating hybrid TLM SW computation models. Application software is mapped onto individual thread models, and process models if necessary. In simulation time, the RTOS model can determine whether a thread is an abstract model or a native-code model, and dispatch different functions to serve it.

It provides better functions than some previous RTOS models by providing generic, POSIX-like [22], and µC/OS-II [23] API services. The RTOS kernel model incorporates the following services: thread management, pre-emptive scheduling services, synchronisation primitives, time services and interrupt handling services.

The timing accuracy of the RTOS model has two aspects. Firstly, the static modelling timing accuracy is flexible, meaning that the model's programmer can decide whether and how to annotate temporal information to each RTOS service function. Secondly, the dynamic simulation timing accuracy is constant and sufficiently accurate, meaning that some important RTOS timing properties can be accurately reflected in simulation time, such as the interrupt latency and interrupt response time. This is because we implement both

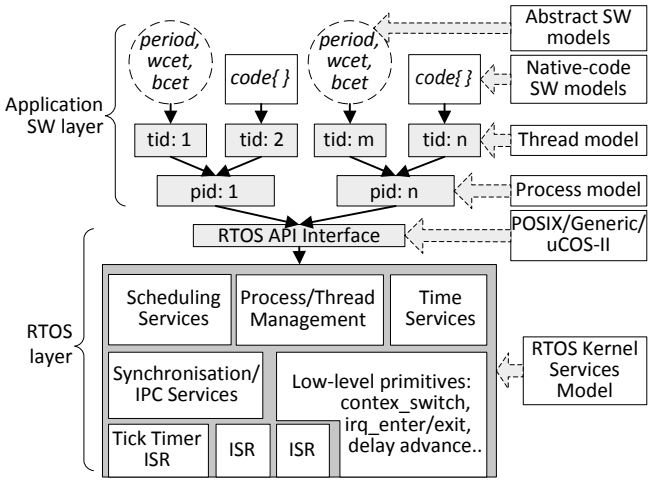

Figure 6. RTOS modelling for TLM SW computation

the pre-emptive scheduling mechanism (using the Live CPU Model) and complete (i.e., nested, prioritised, and maskable) interrupt services in two layered interrupt handling schemes.

V. SOFTWARE PE MODEL WITH OSCI TLM INTERFACES

Referring to the target software possessing element model in Figure 1, this section addresses modelling the hardware abstraction part and the TLM communication part.

In a real embedded system, software runs in a CPU subsystem. In our high-level software modelling approach, this CPU subsystem is abstracted and encapsulated into a model, namely the Live CPU Model. It provides abstract yet essential hardware controlling functions (e.g., interrupt controller, clock information) to upper level software. Especially, it supports the pre-emptive TLM software timed simulation through the Live CPU Simulation Engine by a *wait-for-event* method [20].

For the aim of modelling and simulating a whole embedded system such as a System-on-chip in TLM design, it is necessary to extend our software processing element model with external SW-to-HW, and HW-to-HW communication capabilities. The OSCI TLM-2.0 standard is chosen due to its popularity. This standard consists of two aspects: coding styles about abstraction levels and core interfaces for transporting transactions. We have discussed the former topic in Section III, whilst the latter will now be considered.

The TLM-2.0 core interfaces define three basic "players" for inter-module communication, i.e., *initiators*, *targets*, and *interconnect components*, with *transactions* passed among them. An initiator can create new transaction objects and start communication by calling standard transport methods, whereas a target only receives transactions. An interconnect component acts as a router or an arbiter by which to relay transactions in communication.

According to the above definitions, our software processing element seems a natural *initiator*. An important problem is: where are the initiator functions placed in our models? In order to make the structure of the whole embedded system simulation framework concise and extensible, we maintain a low coupling relationship among the application software model, the RTOS model, and the Live CPU Model. In programming implementation, an object of a Live CPU Model is passed to constructors of the RTOS object and application thread objects as an argument, similar to that between the RTOS object and thread objects (see Figure 7). Consequently, as the root model, the Live CPU Model is the most suitable module to implement initiator's TLM communication primitives. Besides, this is a straightforward way in terms that, in a real design, the CPU controls communications with other hardware components.

Since low-level hardware architecture modelling and complex communication explorations are not the focus of this paper, for simplicity and generality, we set a typical SoC topology as the referenced model (referring to Figure 7), which includes:

- Initiator modules: A Live CPU Model is set for RTOS-based software simulation. An optional hardware IP module can be included for customised computation. Both blocking (the LT socket) and non-blocking (the AT socket) transport interfaces are supported in the Live CPU Model. The HW IP module is connected to the interrupt controller (*irq_ctrl*) in the Live CPU Model by a channel in order to trigger the software processing element in case of an interrupt event.
- Target modules: These could be memory components or peripheral devices. Both LT and AT styles can be applied. In case there is more than one memory module, this topology can represent an embedded system with application data partitioning. The small-size memory module has a fast access speed whilst the big-size memory module is slow.
- Combined initiator/target module: The Direct Memory Access (DMA) controller allows directly moving data between memory locations and devices without intensive handling from the Live CPU Model. It plays roles as both an initiator (for reading and writing data) and a target (being programmed by a DMA requester). The structure and methods used by the DMA controller are illustrated in Figure 8. Three main methods in the model implement a typical DMA mechanism. The `b_transport()` method, which is inherited from the standard TLM class, listens to the target socket and waits for configuration information from requestors. Upon receipt, the configuration information (i.e., source address, destination address, and size of transfer) is saved in pro-

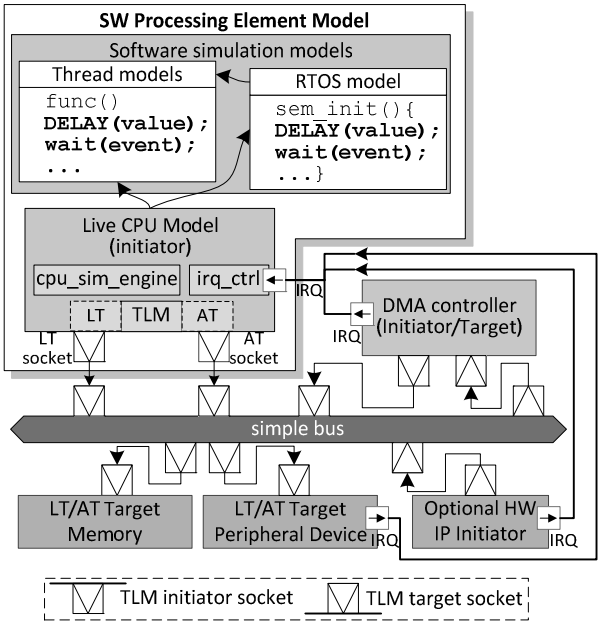

Figure 7. Extending SW PE Model with TLM Communication Model

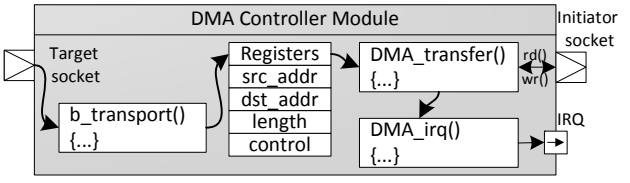

Figure 8. Structure of the DMA controller

grammable registers of the DMA controller. Then the `DMA_transfer()` function begins to read data from source locations and write them to destinations. When the entire DMA transfer is finished, the `DMA_irq()` function will interrupt the Live CPU Model.

- Interconnection: The OSCI TLM-2.0 AT-style *simple_bus* model is selected as the interconnection architecture. It supports both LT and AT models and is extensible by adding more initiators and targets.

In terms of the TLM communication protocol, we apply the *generic payload* recommended by the OSCI TLM library so as to achieve the interoperability of memory-mapped bus models. It provides typical characteristics of memory-mapped bus protocols, for instance command, address, data, single word transfer, and burst transfer, etc [4].

Figure 9 shows three code parts for how to implement TLM interfaces in models and how an application task uses a TLM function. In the Live CPU Model (code (A)), both a LT-style communication sub-component and an AT-style sub-component with associated sockets are instantiated. Code part (B) presents the `write` and `read` methods of the LT sub-component, which inherits the standard `SimpleLTInitiator1` class in the TLM-2.0 library. The AT sub-component implements similar functions, which are omitted for brevity. Referring to the code part (C), an application task "`task_write`" calls TLM communication functions through a pointer to the Live CPU Model.

```
SC_MODULE live_cpu_model{
  ...
  initiator_socket_type PE_LT_socket1;
  TLM_LT_COMPONENT *LT_initiator1;
  ...
  tlm_initiator_socket<32> PE_AT_socket1;
  TLM_AT_COMPONENT *AT_initiator1;
  ...
};
```

(A) Code of TLM components in the Live CPU Model

```
class TLM_LT_COMPONENT:public SimpleLTInitiator1
{ ...
  int LT_write(unsigned uiId, unsigned uiData);
  int LT_read(unsigned uiId, unsigned uiData);
  ...
};
```

(B) Code of the LT TLM communication sub-component

```
void task_write(RTOS *rtos_i_ptr, CPU *cpu_i){
  ...
  cpu_i->LT_initiator1->LT_write(0, array[i]);
  //or
  cpu_i->AT_initiator1->AT_write(0, array[i]);
  ...
}
```

(C) Code of an application task calling TLM functions

Figure 9. Implementation of TLM interfaces in models

VI. EXPERIMENTS

In this section, we present some case studies, in order to demonstrate the performance and capability of our integrated real-time software and TLM communication modelling and simulation approach.

A. Performance Study of TLM Models

The basic benchmark model includes two software threads, an optional RTOS model, two optional memory modules (one is loosely-timed, and the other is approximately-timed), and the before-mentioned *simple bus* TLM model. One software thread implements an *insert-sort* algorithm to process an array and then writes the result to a memory module through the TLM bus, whilst another software thread reads data from that memory module. The RTOS model mimics the µC/OS-II RTOS and is annotated with relevant executing performance on a 48MHz ARM7 processor.

We configure and run the model in six scenarios as follows:

Scenario 1 (Pure SystemC + LT TLM): This is a pure SystemC TLM model. Software threads are implemented as SystemC `SC_THREADS` and use the internal SystemC kernel scheduler but without priorities and pre-emption. Coarse-grained time annotation and the LT style are used for software and TLM communication models, respectively. They can represent the behaviour of a generic functional SystemC simulation.

Scenario 2 (Pure SystemC + AT TLM): The only difference from Scenario 1 is that AT TLM communication is used in this test. It supports more timing phases in a transaction than the LT TLM model.

Scenario 3 (Abstract SW + LT TLM): Our RTOS model and the LT style TLM model are integrated for software simulation in this case. Two abstract (namely coarse-grained timed) software threads are controlled by the RTOS model and utilise RTOS synchronisation and timing services.

Scenario 4 (Abstract SW + AT TLM): Being different from Scenario 3, the AT TLM communication method is used in this case.

Scenario 5 (Native-code SW + LT TLM): In this case, software threads are annotated with fine-grained time delays, whose number is about 1000 times more than the abstract model. Other properties are the same as in Scenario 2.

Scenario 6 (Native-code SW + AT TLM): This case includes both fine-grained timed software model and the AT TLM communication model.

The model of each scenario is executed ten times so as to obtain an average result. In each run, a thread repeats about ten jobs, with two thousand transactions being transferred on the bus. All experiments run on a 1.86GHz x86 PC.

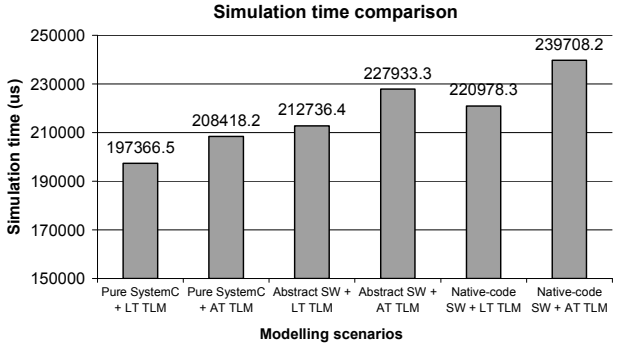

Figure 10. Simulation results

The obtained results are shown in Figure 10. Not surprisingly, pure functional SystemC models achieve the fastest simulation speed due to simplicity.

However, it is more interesting to us that our RTOS-based software simulation only incurs a less than 10% overhead (comparing simulation runtimes of the comparable scenario pair 1, 3 and pair 2, 4), given that it provides essential real-time software services. We notice that, in a similar-purpose study [24], introducing an OS simulation model to functional level SystemC models incurs about a 15% speed overhead, which is at the same order of magnitude as ours.

Besides, as expected, native-code software models and AT TLM models always give a worse performance than corresponding abstract and LT TLM models. Yet, when different levels of software models and TLM models are mixed, it is not a straightforward process to predict the behaviour of their simulation speed. This is since either annotation statements in software models or TLM transaction phases may become the dominant factor in the simulation performance.

B. DMA-based I/O Simulation

DMA is an essential component to reduce CPU Input/Output (I/O) workload in modern computers. It can especially improve CPU performance in some System-on-Chips applications where I/O functions have a high data bandwidth. In a typical ARM-based SoC, I/O functions are implemented by a combination of memory-mapped addressable peripheral registers and interrupt inputs [27]. Our models have good enough capability (i.e., DMA enabled memory transfer and full-functional interrupt handling) to model the two mechanisms.

This experiment implements the RSA cryptography algorithm. The first software task encrypts randomly-generated messages and saves them in the memory. The task then uses a semaphore to synchronise the second task, which will begin to program the DMA controller to transfer ciphered messages and secret keys to a hardware peripheral device as the decipher. After the hardware device decrypts messages, it asks the DMA controller to transfer them back to the memory. The DMA controller raises an interrupt request when it finishes working. Both interrupt-based CPU I/O and DMA transfers are included in the experiment. We expect to observe successfully recovered messages after several transfers across the TLM bus and low frequency of I/O related interrupts by using the DMA method.

Figure 11 shows some parts of the simulation log, which are organised in blocks corresponding to various distinctions of our models. The original messages encrypted in the first software task (in the 1st block) are successfully decrypted by the hardware device (in the 3rd block). Both CPU and the peripheral device can initiate DMA transfers (in 2nd and 4th blocks respectively). Finally, in the last log block, the DMA controller interrupts the CPU to tell its finish, and the interrupt controller (in the Live CPU Model) recognises the IRQ number and notifies the RTOS model for software interrupt handling.

Figure 12 illustrates the simulation timeline. DMA transfers relax the software system from frequent and time-consuming context switches, where the interrupt source only triggers once in each system cycle. This not only means that the running speed (total cycles to finish a specific job) of a I/O intensive model can be improved by utilising DMA, but

```
//SW task: task_encrypt_data() encrypts:
AT  1472170 ns:   |Message to be ciphered: 9614
                  |Ciphered message: 3307
AT  2272170 ns:   |Message to be ciphered: 1454
                  |Ciphered message: 35894
AT  3072170 ns:   |Message to be ciphered: 5878
                  |Ciphered message: 2726
                  ▼
//SW task: task_dma_transfer() uses DMA to
//transfer encryption from memory to HW device
AT  3887545 ns:CPU::CPU0.LT_initiator2:
                   Send write request
                   A = 0x20000000, D = 0x1
AT  3887595 ns:DMA::DMAC:
                   DMA is programmed:
                   A = 0x0         D = 0x1
                   Receive DMA control request.
                   ▼
//HW peripheral device RSA_IP decrypts:
AT  3990020 ns:   RSA_IP::RSA1:
                  Deciphered message =9614
AT  4090020 ns:   RSA_IP::RSA1:
                  Deciphered message =1454
AT  4190020 ns:   RSA_IP::RSA1:
                  Deciphered message =5878
                  ▼
//HW peripheral device uses DMA to transfer
//decrypted data to memory
AT  4190070 ns:DMA::DMAC
                   DMA is called by a device.
AT  4190070 ns:DMA::DMAC: reads:
                   A = 0x10000008
AT  4190255 ns:DMA::DMAC
                   Receives OK response.
                   D = 0x258e
AT  4190255 ns:DMA::DMAC: writes:
                   A = 0x0         D = 0x258e
                   ▼
//DMA controller interrupts the CPU after it
//finishes transferring
AT  4191180 ns:DMA::DMAC
                   DMA transfer finishes.
AT  4191180 ns:<IRQ_SOURCE: irq= DMAC
                 Interrupt happens.
//CPU IRQ_controller acknowledges and processes
//it
AT  4191180 ns:CPU IC::IRQ 6 is raised.
AT  4191180 ns:CPU::IC: ICSR != 0.
                        Call IRQ Handler.
```

Figure 11. Execution flow of the experiment

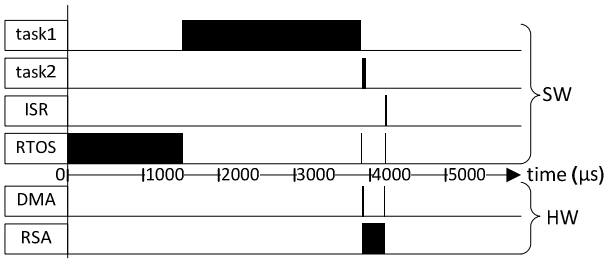

Figure 12. Simulation timeline

also infers the possibility to unitise the CPU more efficiently by implementing some more software functions during DMA transfer durations.

C. Discussion on Simulation Accuracy

Regarding accuracy, the problem is twofold: the RTOS-centric software part and the TLM communication part. The functional and timing accuracy of the former has been demonstrated in previous work [21]. In a case study, the RTOS-centric software simulation can generate correct functional results with less than 5% timing accuracy loss and a 500 times faster speed when compared with an ARM ISS simulator.

On the other hand, the functional accuracy of the TLM communication is guaranteed by inheriting the OSCI TLM-2.0 standard. Because of the highly abstract feature of our conceptual communication platform, the timing accuracy of communication modelling cannot be easily judged. Nevertheless, some academic research as well as some industrial tools have successfully used OSCI TLM as a sound base for in-depth communication modelling [12] [25] [26].

VII. Conclusion and Future Work

In this paper, we have defined two real-time embedded software abstraction levels for software computation modelling in the TLM context. They carry comparable modelling concepts, functional granularity and timing granularity to the existing OSCI TLM-2.0 communication standard. Hence, they make a contribution to the interoperability and standardisation of SystemC-based TLM simulation. At the same time, we have implemented a software processing element model that integrates mixed timing RTOS-centric software modelling, abstract processor hardware modelling and OSCI TLM-2.0 communication interfaces. It is the essential software component in an extensible SoC platform model. We also demonstrate the simulation performance and modeling capability of our modelling in several experiments.

Future work is to extend our work by integrating more complex and implementation-near SoC communication architectures.

References

[1] F. Ghenassia, Transaction Level Modeling with SystemC: TLM Concepts and Application for Embedded Systems: Springer, 2005.

[2] 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.

[3] A. Donlin, "Transaction Level Modeling: Flows and Use Models," in Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2004, pp. 75-80.

[4] OSCI TLM Work Group, "OSCI TLM-2.0 Language Reference Manual (Software Version: TLM 2.0.1)," available at http://www.systemc.org/, 2009.

[5] G. Schirner and R. Dömer, "Fast and Accurate Transaction Level Models Using Result Oriented Modeling," in Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design, 2006, p. 368.

[6] S. Pasricha, "Transaction level modeling of SoC with SystemC 2.0," in Synopsys User Group Conference (SNUG), 2002.

[7] 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.

[8] 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.

[9] J. Madsen and M. Gonzalez, "Abstract RTOS Modelling in SystemC," in 20th IEEE NORCHIP Conference, 2002, pp. 43-49.

[10] 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.

[11] 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.

[12] M. Baklouti, A. Benzina, A. Bouchhima, and F. Petrot, "Extending transaction level modeling for embedded software design and validation," in Design & Technology of Integrated Systems in Nanoscale Era, 2007. DTIS. International Conference on, 2007, pp. 64-69.

[13] R. Dömer, "Transaction Level Modeling of Computation," Technical Report, Center for Embedded Computer Systems, University of California, Irvine., available at 2006.

[14] G. Schirner, A. Gerstlauer, and R. Domer, "Abstract, Multifaceted Modeling of Embedded Processors for System Level Design," Proceedings of the 2007 conference on Asia South Pacific design automation, pp. 384-389, 2007.

[15] M. Krause, O. Bringmann, and W. Rosenstiel, "Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating systems," Design Automation for Embedded Systems, vol. 10, pp. 229-251, 2005.

[16] A. C. Nacul, M. Lajolo, and T. Givargis, "Interface-Centric Abstraction Level for Rapid Hardware/Software Integration " in FDL'05 - Forum on Specification and Design Languages Lausanne, Switzerland, 2005, pp. 329-340.

[17] H. Yu, R. Dömer, and D. Gajski, "Embedded software generation from system level design languages," in Proceedings of the 2004 conference on Asia South Pacific design automation: electronic design and solution fair, 2004, pp. 463-468.

[18] 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.

[19] M.-K. Chung, S. Yang, S.-H. Lee, and C.-M. Kyung, "System-Level HW/SW Co-Simulation Framework for Multiprocessor and Multithread SoC," in VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on, 2005, pp. 177-179.

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

[21] K. Yu and N. Audsley, "A Generic and Accurate RTOS-centric Embedded System Modelling and Simulation Framework," in The Fifth UK Embedded Forum 2009 (UKEF '09), Leicester, UK, 2009.

[22] "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., available at 2004.

[23] J. J. Labrosse, Microc/OS-II: The Real-Time Kernel: Cmp, 2002.

[24] 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.

[25] CoWare, "CoWare Commits Support for SystemC TLM-2.0 Standard," http://www.coware.com/news/press664.htm.

[26] ARM, "ARM's IP and OSCI TLM 2.0," http://www.nascug.org/events/8th/nascug_8_paper_5.pdf.

[27] S. Furber, ARM system-on-chip architecture: Addison-Wesley Professional, 2000.