ChronoSym: a new approach for fast and accurate SoC cosimulation

Iuliana Bacivarov* and Aimen Bouchhima

SLS Group, TIMA Laboratory, 46 Av. Félix Viallet, 38031 Grenoble, France
E-mail: Iuliana.Bacivarov@imag.fr E-mail: Aimen.Bouchhima@imag.fr
*Corresponding author

Sungjoo Yoo

CAE Center, SoC R&D lab. System LSI,
Semiconductor Business, Samsung Electronics, Soowon, Korea
E-mail: sungjoo.yoo@samsung.com

Ahmed A. Jerraya

SLS Group, TIMA Laboratory, 46 Av. Félix Viallet, 38031 Grenoble, France
E-mail: Ahmed.Jerraya@imag.fr

Abstract: The early validation of modern SoC is not anymore feasible using traditional cycle-accurate cosimulations. These are based on the concurrent execution between SW running on multiple Instruction Set Simulators and HW simulators. The challenge is then speeding-up the simulation, without sacrificing the accuracy of traditional methods. The key contribution of this paper is a novel fast and accurate HW/SW cosimulation approach, allowing to handle large SoCs, while reducing the design cycle. The key underlying concepts are timed native SW execution, combined with detailed HW/SW interaction models. This approach is implemented in ChronoSym tool and applied to two real SoC applications.

Keywords: SoC; HW/SW cosimulation; HAL simulation model; timed native execution; timing accuracy.

Reference to this paper should be made as follows: Bacivarov, I., Bouchhima, A., Yoo, S. and Jerraya, A.A. (2005) 'ChronoSym: a new approach for fast and accurate SoC cosimulation', *Int. J. Embedded Systems*, Vol. 1, Nos. 1/2, pp.103–111.

Biographical notes: Iuliana Bacivarov is currently a PhD student in microelectronics at TIMA Laboratory, National Polytechnic Institute of Grenoble, France. She received the Electronics Engineer degree from Polytechnic University of Bucharest, Romania in 2002 and the MS degree in Microelectronics from University Joseph Fourier, Grenoble, France in 2003. Her current research interests are networks-on-chip and performance evaluation and optimisation for multi-processor SoC design. She is also teaching electronics and computer sciences at ENSIEG (Ecole Nationale Supérieure d'Ingénieurs Electriciens de Grenoble) since 2003, and at ENSERG (Ecole Nationale Supérieure d'Electronique et de Radioélectricité de Grenoble) since 2005.

Aimen Bouchhima is currently a PhD student in microelectronics at TIMA Laboratory, France. He received his Eng. degree from the Polytechnic School of Tunisia in 2001 and the MS degree in Microelectronics from University Joseph Fourier, Grenoble, France in 2002. His current research interests include high level embedded software validation and HW/SW cosimulation in SoC design.

Sungjoo Yoo is currently working at Device Solution Network, Samsung Electronics in Soowon, Korea. He received PhD from Seoul National University in 2000. He worked as Senior Researcher at TIMA Laboratory from 2001 to 2004. His research interests include hardware/software cosimulation, hardware-dependent software, multi-processor system-on-chip, and network-on-chip.

Ahmed Jerraya is Research Director in System-Level Synthesis group at TIMA Laboratory. He received a PhD in computer sciences from the University of Grenoble in 1989. He is a Member of the IEEE, SIGDA and EDAA. He also served as general chair or program chair of several international Workshops and symposia. He organised several international courses including MPSoC (Multiprocessor System-on-Chip) School. He has published 227 papers in International Conferences and Journals and seven books. He received the Best Paper Award at the 1994 ED&TC for his work on Hardware/Software Cosimulation. His research interests include hardware/software interface design methods and MPSoCs.

1 Introduction

Modern SoC integrates on the same platform heterogeneous multi-processor systems together with off-the-shelf HW, in order to run large distributed application SW. The subsequent HW/SW codesign becomes increasingly complex and widely diverse.

In SoC design, about two thirds of the design time is spent in validation. The current SoC validation practice suggests the use of HW prototypes based on FPGAs or reconfigurable platforms. But this solution induces large delays in the design process and may violate the time-to-market constraints. Additionally, the early SoC validation is not anymore feasible using traditional cycle accurate HW/SW cosimulation approaches. These are based on running SW on multiple ISSs (Instruction Set Simulators) and executing them concurrently with HW simulation at transaction level or cycle-accurate level (http://www.mentor.com; http://www.coware.com; http://www.synopsys.com). The problem comes from the slow simulation speed of ISS. As well, for fast simulation, a behavioural high-level model (called virtual hardware in Rowson (1994) can be used. It executes the application SW on a simulation host. High simulation speed is usually attained. But ignoring the real HW architecture, it lacks the accuracy.

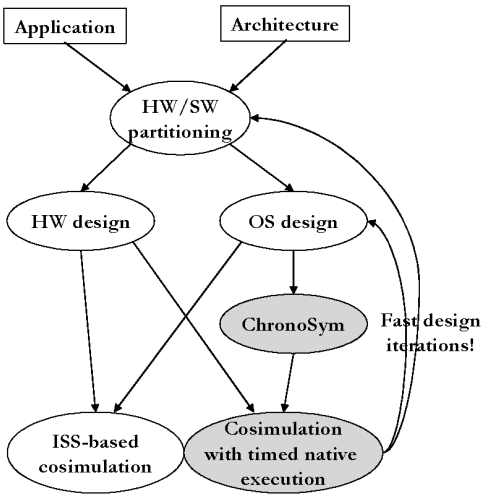
The challenge is then, to provide faster validation than traditional cosimulation methods, without sacrificing the accuracy. A fast validation will allow to explore the design space while guaranteeing time-to-market constraints. The validation accuracy will certify the correctness of design.

This paper proposes a fast and accurate method for HW/SW cosimulation. It is based on native SW execution to achieve simulation speed. We call the SW simulation model a *timed native execution model*. In order to achieve accuracy in SW simulation, we execute real OS natively on a simulation host and run timed SW simulation. The simulation allows for detailed HW/SW interactions, considering HW interrupt and I/O together with SW time advancement.

A tool namely ChronoSym was developed to implement the proposed simulation model. Figure 1 shows how ChronoSym is used in SoC design flow. Given an application and a target architecture, the application is partitioned into SW and HW. Then, the OS is designed to run multiple tasks on the processors and the HW design is performed to yield a RTL implementation. The OS can be designed by configuring existing configurable OSs or by developing application-specific ones. Conventional ISS-based cosimulation model can be built after OS and HW design as shown in the figure.

ChronoSym is applied to the SW code after the OS is designed. The input to ChronoSym is the code of OS and application SW. ChronoSym generates a timed native execution model. HW simulation can be performed using cycle-accurate, transaction level or behavioural level model of HW modules. Compared with ISS-based cosimulation, in HW/SW cosimulation, the timed native execution model replaces only the ISS while having the same HW simulation model.

Figure 1 ChronoSym in SoC design flow



The timed native execution model allows for fast and accurate cosimulation which will enable designers to perform fast design iterations in SoC design flow. As shown by the backward arrows from cosimulation step to OS design and HW/SW partitioning steps, the fast design iterations will allow designers to explore more design choices in OS design and HW/SW partitioning within the given time-to-market constraint.

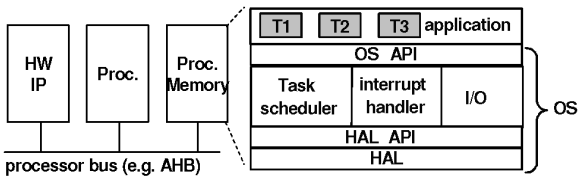
Two SoC applications, namely VDSL and McDrive, were used to prove the effectiveness of the presented model and the tool efficiency. The rest of the paper is organised as follows. Section 2 provides the background on HW/SW cosimulation, presenting different HW/SW cosimulation models, and explaining their accuracy and speed. Section 3 describes the proposed timed native execution model by detailing the developed tool, ChronoSym. Section 4 presents experimental results. Section 5 gives the conclusion.

2 Speed and accuracy of HW/SW cosimulation

2.1 HW/SW system

Figure 2 shows a specific SoC architecture made of processor, memory, HW IP and processor bus. The processor runs complex application SW, typically executing millions of instruction cycles. Furthermore, the SW is continuously enhanced with new features, such as parallel tasks, various input/outputs and interrupt handling. So, an operating system (OS) is needed.

Figure 2 HW/SW system

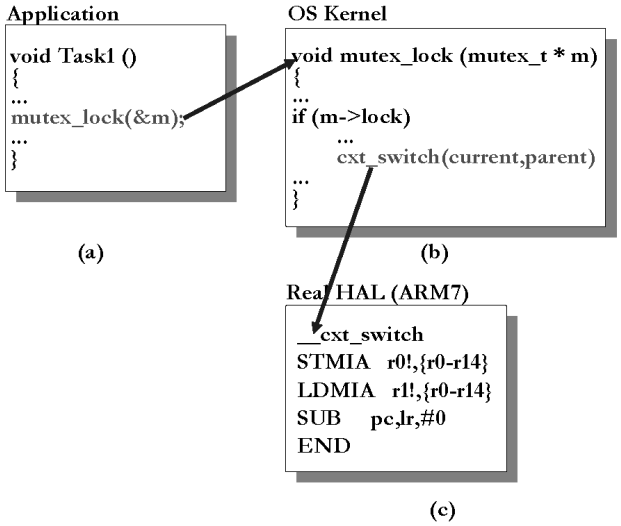


The OS provides application SW with OS services via OS application programming interface (API): e.g., scheduling policy, interrupt management and I/O. We divide the OS

code into two types: processor independent code and processor dependent code. We call the processor dependent code hardware abstraction layer (HAL). Examples of HAL functions are context switch, bus I/O, exception handlers, etc. The HAL functions are called by the processor independent part of OS via its application programming interface, HAL API.

Figure 3 illustrates OS and HAL API functions. Task 1 calls an OS API function, mutex_lock. This is implemented in OS Kernel illustrated in Figure 3(b). The OS function calls a HAL API function for context switch, cxt_switch. The OS kernel code can be processor-independent since HAL API functions hide all the processor dependency. HAL API functions have processor-dependent code. Figure 3(c) illustrates the assembly code of HAL API function, context switch for an ARM7 processor (ARM7 Technical Reference Manual, 2001).

Figure 3 Examples of OS and HAL API function

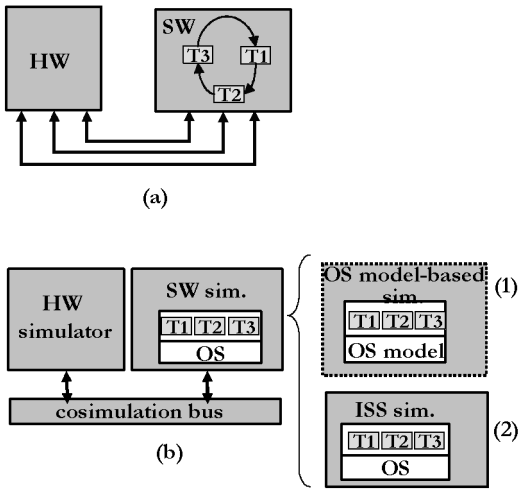


2.2 HW/SW cosimulation techniques

Existing HW/SW cosimulation techniques can be classified into three types: functional (often timed), OS model-based and ISS-based cosimulation. Figure 4(a) illustrates the functional (timed) cosimulation. Both HW and SW are specified as functional modules communicating through signals. In this case, the SW code is natively executed. In native execution, the SW code runs on a simulation host, without using a SW simulator (like an ISS). Obviously, this simulation will validate only the system behaviour. The SW tasks run in parallel, without an OS scheduler, on the simulation host environment, e.g., SystemC (http://www.systemc.org). The details of HW/SW interactions (e.g., I/O, interrupt) are not simulated. The HW and SW timing can be simulated by annotating HW and SW modules with their execution delays if such delay information is available.

Figure 4(b) illustrates OS model-based and ISS-based cosimulation. HW/SW cosimulation consists in parallel running specific simulators that host SW and HW. Because the SW and HW simulators continuously exchange data, synchronisation is needed. This is usually performed by a cosimulation bus.

Figure 4 Three types of HW/SW cosimulation



In the case of OS model-based cosimulation, for SW simulation, application SW is executed natively on top of an OS model (http://www.windriver.com/products/html/vxsim.html; Desmet et al., 2000) Bradley and Xie, 2002; Gerstlauer et al., 2003; http://www.cadence.com/products/vcc.html). The OS model emulates the OS API including multi-tasking, interrupt handling and device drivers for I/O. The OS model-based cosimulation enables to model the sequential execution of multiple SW tasks running on the same processor. However, the real OS is not used, but only the OS API is emulated. Thus, the impact of OS on system run, e.g., OS runtime overhead, is not correctly simulated.

For instance, in the OS model of http://www.cadence.com/products/vcc.html, the delay of task scheduling can be modelled as an aggregated delay value in a function, e.g., the delay of earliest-deadline first scheduling = $a + b \times n \times \log n$ where n is the number of tasks and a and b are curve-fitting parameters. Such a delay function lacks in accuracy since it gives a coarse grain delay value.

In ISS-based cosimulation, for SW simulation, an ISS is executed for each target processor. The ISS runs on the simulation host while interpreting the SW assembly code. The ISS interprets the execution of instruction through pipeline stages, e.g., fetch/decode/execute steps of ARM7 processor. At the same time, it emulates the underlying processor HW (e.g., I/O and interrupt signals). The ISS-based cosimulation simulates the entire SW code, including the OS and application SW.

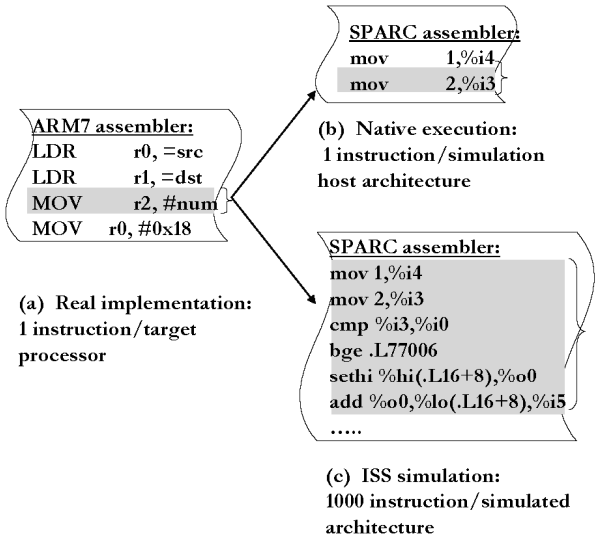
2.3 SW simulation speed

Obviously, the three cosimulation models have different simulation speed, especially in SW simulation. The maximum speed is obtained by the functional (timed) cosimulation since both SW and HW modules are executed natively on the simulation host. The ISS-based cosimulation gives the lowest speed since the ISS simulates the detailed processor

execution, interpreting each assembly line. OS model-based cosimulation maintains the high-speed simulation, since the application SW code is executed natively on the simulation host.

Usually, the native execution of SW is two or even three orders of magnitude faster than ISS simulation of SW. Figure 5 explains this difference. Figure 5(a) illustrates an assembly code of target processor (e.g., ARM7). Figure 5(b) and Figure 5(c) present the assembly code of simulation host machine (e.g., SPARC) for native execution and for ISS simulation, respectively. In Figure 5, shaded areas correspond to each other. For the simulation of a single assembly line of ARM7 processor, 'MOV r2, #num', the native execution runs 'mov 2, %i3' on a SPARC host machine. For the simulation of the same assembly line of ARM7 processor, the ISS runs 100–1000 assembly lines of the host machine as shown in the shaded area of Figure 5(c) since it simulate the processor functionality, e.g., pipeline stages such as fetch, decode, and execute. For further details of ISS simulation, refer to Zivojnovic and Meyr (1996).

Figure 5 Comparison of the numbers of instructions to simulate/execute a single assembly line of target processor



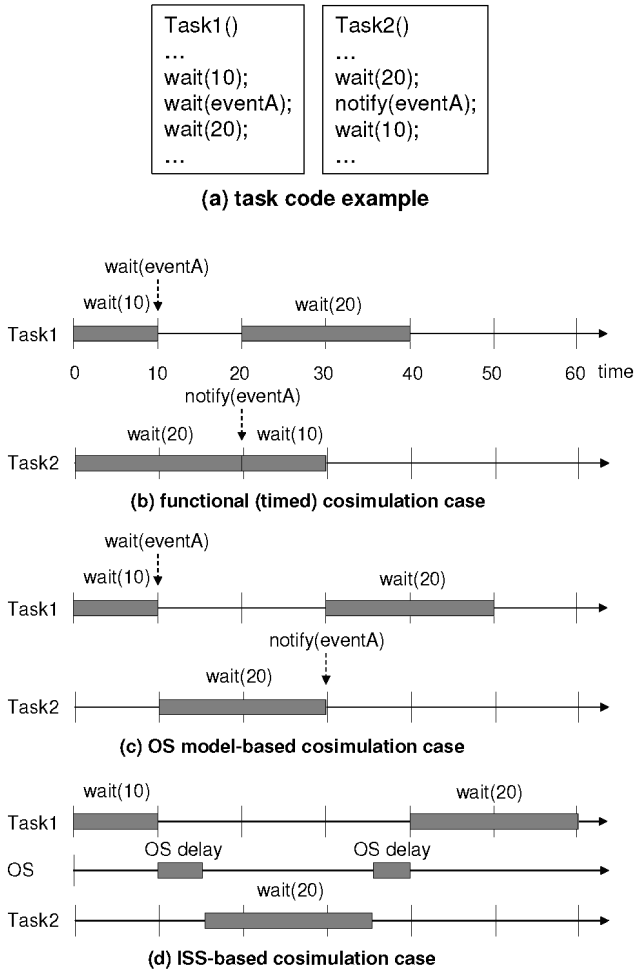
As the figure illustrates, for the native execution, the same order of magnitude of instructions will run on the host processor as on the target processor. With such a (roughly) one-to-one correspondence, a high speed is obtained in native execution. The comparison between Figures 5(b) and 5(c) shows that native execution may yield about 100–1000 times of speed up compared with ISS simulation.

2.4 HW/SW cosimulation timing accuracy

The three cosimulation models have different accuracy. ISS-based cosimulation gives the highest accuracy since the ISS gives instruction/cycle-accurate simulation. The functional (timed) cosimulation has the lowest accuracy since the system is described only by its behaviour, not taking into account the target processor (e.g., serialisation of multiple SW tasks) or detailed HW/SW interactions (e.g., interrupt). OS model-based cosimulation considers architectural details (e.g., the tasks serialisation by the OS model). Thus, when compared with functional (timed) cosimulation, it gives better accuracy. However, it does not give the same accuracy as the ISS since the real OS is not simulated.

Figure 6 illustrates timing accuracy of three types of cosimulation. Figure 6(a) gives an code example of two tasks, Taskl and Task2. Figure 6(b) shows a simulation trace of functional (timed) cosimulation of the two tasks. Shaded areas represent that the corresponding task runs. Assume that Taskl executes wait (10) and Task2 executes wait (20) at time 0. The two tasks will run concurrently in this case between time 0 and 10 since the task serialisation is not simulated. At time 10, Taskl stops its execution after calling wait (eventA). It waits for a notification on the event. Task2 continues to run by time 20. It calls notify (eventA) which gives an event notification to Taskl. Then, Taskl can resume its execution. Both tasks run concurrently again between time 20 and 30.

Figure 6 Timing accuracy in three types of cosimulation



In OS model-based cosimulation, task serialisation is simulated using the OS model. Assume that both the tasks run on the same target processor and that Taskl starts to execute at time 0 in Figure 6(c). It executes wait (10)[1] by time 10. During the period, Task2 does not run. Thus, task serialisation is simulated. At time 10, calling wait (eventA)

causes task context switch from Taskl to Task2. Then, Task2 executes wait (20) by time 30. During this period, Taskl does not run. At time 30, calling notify (eventA) causes another task context switch from Task2 to Taskl. Simulation continues in this way.

In ISS-based cosimulation, the real OS is simulated. Thus, the effects of OS overhead are simulated. In Figure 6(d), Taskl runs by time 10 as in the previous case. At time 10, wait (eventA) is called.[2] The ISS simulates the execution of wait (eventA) including task context switch from Taskl to Task2 on the target processor. Since the task context switch takes time in reality, the ISS simulation yields an execution delay during the execution of wait(eventA). Thus, Task2 is resumed after time 10 and the OS delay of context switch as shown in Figure 6(d). In the case of OS model-based cosimulation, this delay, i.e., OS overhead is not taken into account since the real OS is not simulated.

2.5 Proposed HW/SW cosimulation method

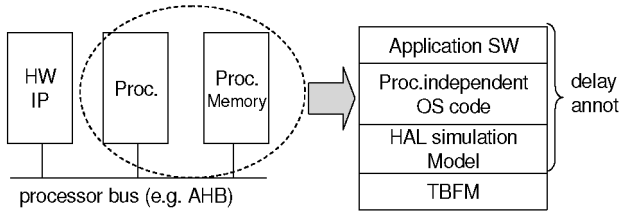
We present a method of fast and accurate cosimulation based on timed native execution model for SW simulation. Basically, we exploit native execution for SW simulation. Compared with conventional methods of SW simulation using native execution (functional simulation and OS model-based simulation), our contribution is that the real OS and detailed HW/SW interactions are simulated. Thus, the presented method yields better accuracy than those methods while exploiting the advantage of native execution, speed.

3 ChronoSym: a fast and accurate SW simulation technique

Figure 7(a) shows a specific SoC architecture that contains processor, memory, HW IP, and processor bus. The timed native execution model (shown in Figure 7(b)) is a simulation model of the processor and the SW code on the processor memory as illustrated in the figure. A timed native execution model consists of application SW, processor-independent OS code, simulation model of hardware abstraction layer, and timed bus functional model (TBFM).

The basic principle of timed native execution model is (1) to natively execute real SW code including real OS code on a simulation host and (2) to simulate both the timing of SW execution and HW/SW interactions.

Figure 7 Timed native execution model



(a) Processor sub-system (b) Timed native execution model

For the native execution of real SW code, the application SW code and processor-independent OS code are executed natively on the simulation host since they are portable on different processors. However, the processor-dependent code of OS, i.e., hardware abstraction layer (HAL) cannot be executed on a simulation host machine. Thus, we make a simulation model of HAL and execute it on the simulation host.

For the timed simulation of SW, we annotate the SW code with execution delays. Then, both SW and HW simulation are synchronised. The HW/SW interactions are handled in two ways. The SW to HW interaction, i.e., memory access is handled by the conventional BFM supported by TBFM. The HW to SW interaction, i.e., processor interrupt is handled in the timed simulation of SW.

TBFM contains the conventional BFM and a support function for the HW/SW synchronisation needed in the timed simulation of SW.

3.1 HAL simulation model

The HAL simulation model emulates the HAL API. In the following, we will explain how to build the simulation model for each type of HAL API functions using three types of HAL API functions: context switch, I/O, and exception handling.

3.1.1 Context switch

Figure 8(a) and 8(b) show an example of context switch code in ARM7 assembly and its simulation model on a UNIX machine, respectively. In the example, the task context and context switch operations are modelled using UNIX multithreading library data structure, ucontext_t (for ucp_list[] in Figure 8(b)) and functions, get/setcontext(). To be more specific, function getcontext() emulates 'STMIA' instruction in Figure 8(a). In reality, it saves the values of ARM7 processor registers to the stack of current task. In the simulation model, it saves the current context of SW simulation execution to a data structure, ucp_list[cur]. Function setcontext() emulates 'LDMIA' instruction in the original context switch code while restoring the saved context of SW simulation execution.

Figure 8 Context switch code and its HAL simulation model

```
__cxt_switch          ;r0, old stack pointer, r1, new stack pointer
STMIA  r0!,{r0-r14}     ; save the registers of current task
LDMIA  r1!,{r0-r14}     ; restore the registers of new task
SUB    pc,lr,#0         ; return
END
```

(a) Context switch: assembly code for ARM7 processor

```
void context_sw(int cur, int new)
{
        delay(34);
        getcontext(ucp_list[cur]); // UNIX multi-threading lib.
        setcontext(ucp_list[next]); // UNIX multi-threading lib.
        delay(3);
}
```

(b) Context switch: simulation model in UNIX

3.1.2 I/O

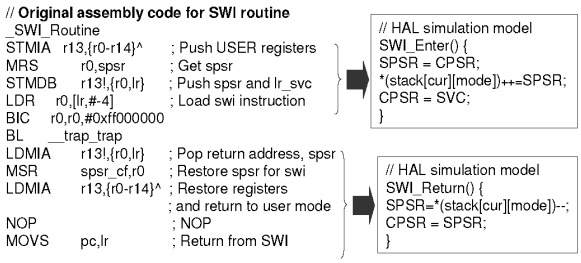
I/O functions represent memory read/write operations. To model the I/O functions, we use the conventional bus functional model which is included in the TBFM. The BFM transforms a memory access to events on processor interface signals, e.g., address/data bus, control signals.

3.1.3 Processor exception handler

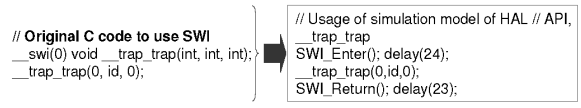
Processors can have several types of processor exception. For instance, an ARM7 processor has seven different types of exception: reset, undefined instruction, software interrupt (SWI), prefetch abort, data abort, IRQ, and FIQ (ARM7 Technical Reference Manual, 2001). We observe that, to build the simulation model of HAL, all the exceptions are not required to be modelled. For instance, the exception of undefined instruction is not related to the emulation of HAL API. In the case of ARM processor, five exceptions, SWI, prefetch abort, data abort, IRQ, and FIQ are related for the emulation of HAL API.

Figure 9 shows an example of modelling a HAL API function, SWI handler code, of ARM7 processor. The left-hand sides of Figure 9(a) and 9(b) show a SWI handler in assembly code _SWI_Routine and a C code section to call a generic SWI function called_trap_trap() with SWI number 0 (defined by _swi(0)).[3] When the function_trap_trap() is called (in Figure 9(b)), the processor execution jumps to the vector table element of SWI, then the SWI handler, _SWI_Routine (in Figure 9(a)) is executed.

Figure 9 Building a timed simulation model of HAL API



(a) Modeling a SWI routine



(b) Calling the simulation model of SWI routine

To model such exception handlers, our strategy is to model the minimal set of processor state and the related operations. In the case of ARM processor, the minimal set of processor state is made of processor mode registers (CPSR and SPSRs) that contain the information of processor state (ARM or THUMB), interrupt disables (IRQ and FIQ disables), and processor mode (among seven modes).[4] Processor mode registers need to be simulated to identify the current interrupt disable status (to determine whether a processor interrupt can be processed or not) and the current processor mode (to determine whether a memory access causes a protection violation or not).

The right-hand side of Figure 9(a) shows the modelling of the SWI handler. Two functions SWI_Enter() and SWI_Return() model the entry and return operations of the SWI routine. In the two functions, only the operations related to the mode registers (CPSR and SPSRs) are simulated. For instance, to simulate the operation that the SWI handler saves/retrieves SPSR to/from the stack of current task, we model the task stack with stack[cur] [mode] and put/get SPSR in/from the stack model.[5] The current process status register (CPSR) of new processor mode is also simulated while setting CPSR to SVC (supervisor mode).

The right-hand side of Figure 9(b) shows that the two functions are added before and after the SWI call. We model the other exception handlers of prefetch abort, data abort, FIQ, and IRQ in the same way.

In our work, we investigated the case of ARM processor to model the exception handling of processor. We will formalise the modelling to account for general cases where more complex exception handling may be found.

3.2 Simulation of SW timing and processor interrupt

3.2.1 Delay annotation

Figure 10 shows the flow of delay estimation and annotation for processor-independent code. First, the source code (in C) is compiled with the target compiler (e.g., armcc or gcc with ARM7 as the target processor). The execution delay of each assembly instruction is calculated using the data sheet of target processor supplied by the processor vendor (ARM7 Technical Reference Manual, 2001). Then, the correspondence between the source code and the assembly code is found. For instance, in Figure 10, the source code line, $i = 4$ corresponds to a MOV assembly instruction, two LDR and a STR. Function delay() is annotated to the corresponding source code line. In the example of Figure 10, for each assembly instruction, function delay() is used. For the efficiency of simulation speed, delay() functions (corresponding to source code lines) might be merged into a single one while trading off with timing accuracy.

We annotate the simulation model of HAL as well as processor-independent code with delays. Currently, we annotate the HAL simulation model manually although the delay annotation of processor-independent code is automated.

The accuracy of delay estimation is crucial to the accuracy of HW/SW cosimulation with timed native execution models. In Section 4, we will give comments on this issue after presenting the experimental results.

3.2.2 Function delay()

Function delay() has two roles of synchronising HW and SW simulation times and simulating processor interrupt. Figure 11 illustrates how function delay() plays both roles. Assume that function delay(10) is called from the delay annotated SW code shown in Figure 10. Figure 11(a) shows a case when there is no processor interrupt during the period of 10 time ticks. In that case, function delay(10)

returns after the entire period of 10 ticks elapses. The SW simulation time advances by 10 time ticks as the result of executing function delay(10).

Figure 10 Automated delay annotation

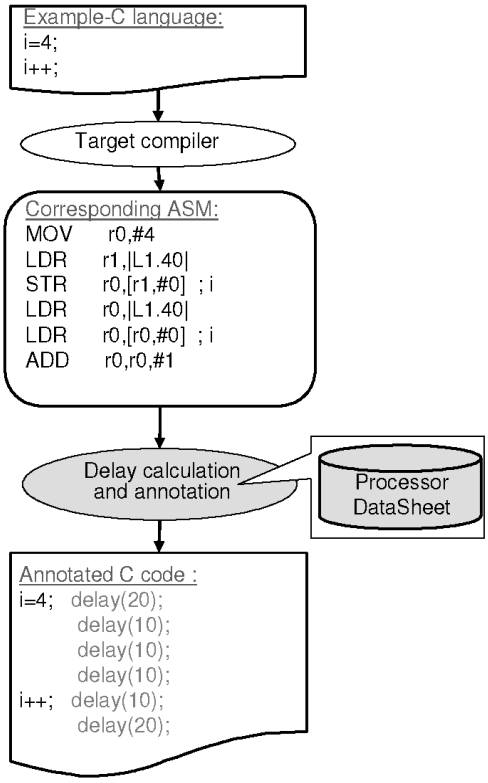
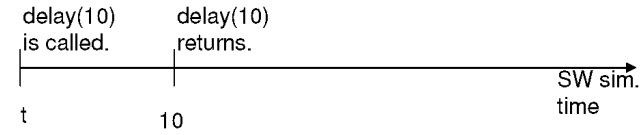
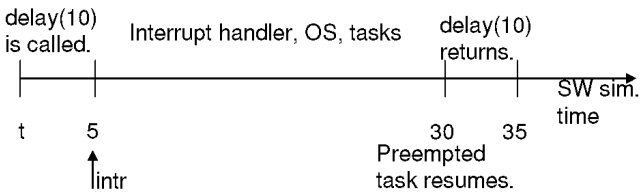


Figure 11 Advance of simulation time



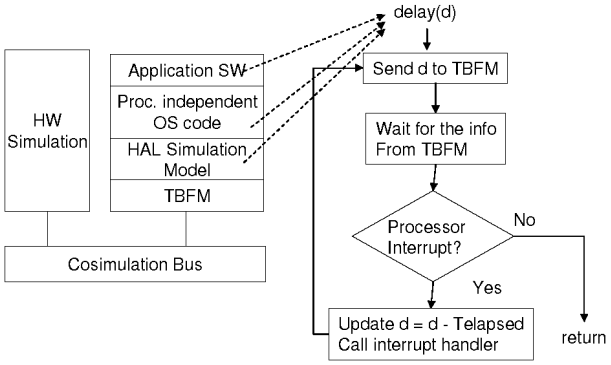
(a) No interrupt case



(b) Interrupt case

Figure 11(b) shows a case when there is a processor interrupt at time 5 during the time period of 10 time ticks. In this case, when the interrupt arrives, in reality, the execution of current task is pre-empted and a processor interrupt handler is invoked. To simulate this, function delay() calls the interrupt handler model (IRQ or FIQ handler model as described in Figure 9). The interrupt handler can even call OS functions (e.g., lock release) to enable task context switch. Figure 11(b) illustrates that the execution of interrupt handler, OS function(s) and other task(s) than the pre-empted one, takes 25 time ticks. After the interrupt handler (and/or the execution of other tasks) is finished, the execution of original task which called function delay(10) resumes at time 30 as shown in the figure. In this case, after the period of remaining delay, i.e., 5 time ticks, function delay(10) returns.

Figure 12 Function delay()



3.2.3 Function delay() and TBFM

Function delay() works with TBFM to advance SW simulation time and to simulate processor interrupt. Figure 12 shows the details of function delay() and TBFM.

When function delay(*d*) is called, the function sends the delay value d to TBFM. TBFM advances the SW simulation time waiting on events on processor interrupt signals (e.g., nIRQ, nFIQ signals of ARM7 processor). When there is no processor interrupt during the period of *d* time ticks, TBFM advances the current SW simulation time by *d* time ticks and gives to function delay() the current SW simulation time and the status of processor interrupt (no interrupt, in this case). When there is a processor interrupt during the period of d time ticks, TBFM stops advancing the SW simulation time at the moment when the interrupt arrives. Then, it sends to function delay() the current SW simulation time and the status of processor interrupt (interrupt triggered in this case).

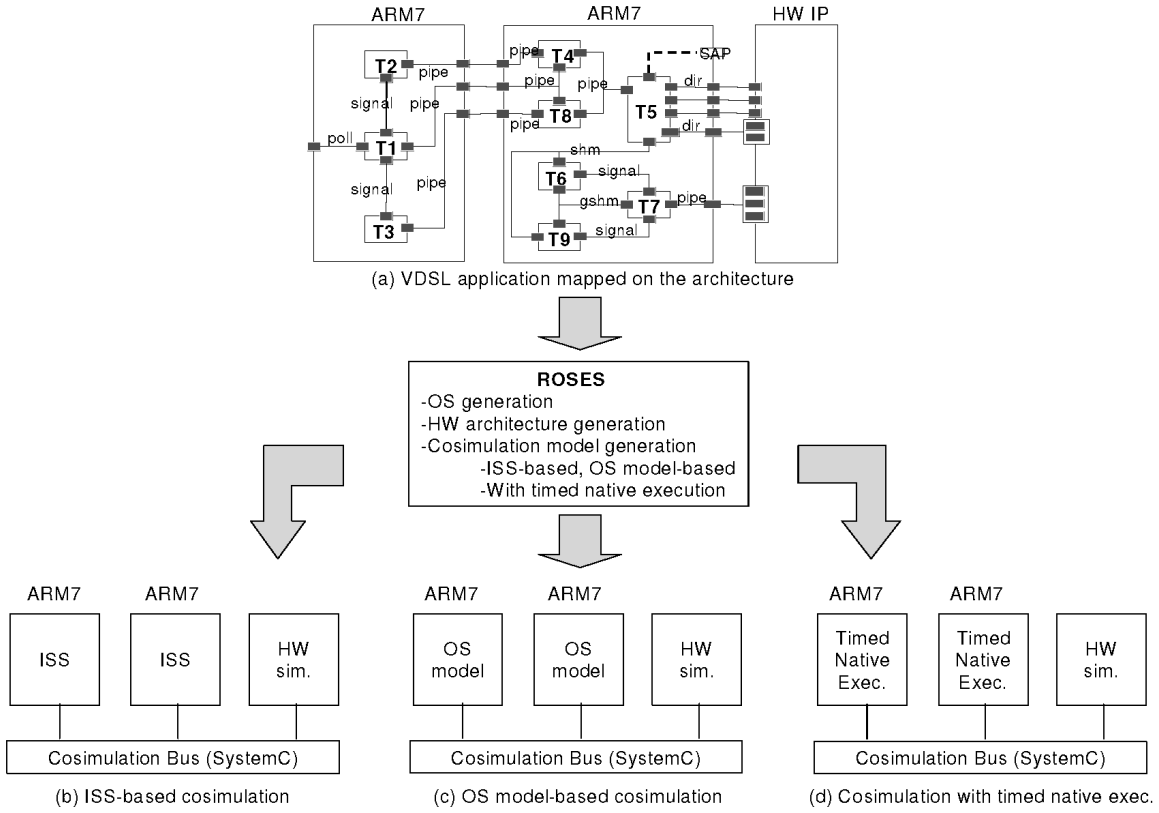
4 Experiments

4.1 VDSL and McDrive systems: design and cosimulation models

We used an industrial example namely VDSL (Diaz-Nava and Okvist, 2001) and our own example called McDrive in our experiment. We designed the VDSL system on a multiprocessor SoC (MPSoC) architecture which includes two ARM7 processors and one HW IP. Figure 13(a) shows the VDSL application mapped on the MPSoC architecture. As shown in the figure, three SW tasks are mapped on one ARM7 processor and six SW tasks on the other processor. SW tasks communicate with each other via OS primitives such as pipe, signal, and shared memory. We designed the McDrive system with one ARM7 processor and three HW IPs.

We used a SoC design tool called ROSES (Cesario et al., 2002) to design RTL implementations from the applications mapped on the architectures. To be specific, the tool generates OSs (for two ARM7 processors of VDSL system and for one ARM7 processor of McDrive system) and HW glue logics to build a complete RTL implementation. The tool gives also HW/SW cosimulation models based on ISSs and OS models as shown in Figures 13(b) and 13(c), respectively. We integrated ChronoSym into ROSES and generated the HW/SW cosimulation model with timed native execution models as illustrated in Figure 13(d).

Figure 13 VDSL system: design and cosimulation models



4.2 Experimental results

We ran the three cases of cosimulation for the VDSL and McDrive systems. In all the cases, the HW simulation runs at RTL in SystemC. All the SW simulation models (ISSs, OS models, and timed native execution models) run on separate UNIX processes. Between HW and SW simulation, we used UNIX shared memory for inter-process communication.

Tables 1 and 2 give the runtime and accuracy of three types of cosimulation for the McDrive and VDSL systems, respectively. In terms of simulation speed, compared with ISS-based cosimulation, OS model-based cosimulation and the proposed method give 13 and 10.8 times speedup, respectively in the case of McDrive system. The proposed method gives 2.1 times speedup compared with ISS-based cosimulation in the case of VDSL system.

Note that, in both cases of McDrive and VDSL systems, native simulations (OS model-based cosimulation and the presented method) give significant simulation speedups. However, they do not reach a theoretical maximum speedup (100–1000 times) explained in Figure 5. It is because (1) the HW simulation model at cycle-accurate RTL is fixed in all the three types of HW/SW cosimulation and (2) SW and HW simulation synchronise with each other via expensive UNIX shared memory. In the case of McDrive system, the HW simulation takes 48.9% of total simulation runtime and the synchronisation takes 31.8%. In the case of VDSL system, the HW simulation and synchronisation take 71.2% and 21.5%, respectively. We expect the simulation speedup will increase when the HW simulation is performed at higher abstraction levels (e.g., transaction level) than cycle-accurate RTL.

Table 1 Comparison of cosimulation runtime and accuracy (error) of McDrive system

	ISS-based cosimulation	*OS model – cosimulation*	*Cosim. with timed native execution model*
Simulation runtime (speedup)	6.5 sec	0.5 sec (13 times)	0.6 sec (10.8 times)
Simulated cycles (error)	73,800 cycles	50,600 cycles (31.4%)	61,100 cycles (17.2%)

Table 2 Comparison of cosimulation runtime and accuracy (error) of VDSL system

	ISS-based cosimulation	*OS model – cosimulation*	*Cosim. with timed native execution model*
Simulation runtime (speedup)	32sec	12 sec (2.7 times)	15 sec (2.1 times)
Simulated cycles (error)	1,486,000 cycles	1,026,400 cycles (30.9%)	1,227,600 cycles (17.4%)

In terms of timing accuracy, we assume that ISS-based cosimulation gives no error. Accordingly, the timing error is calculated as follows:

$$\text{error} = \frac{\left\| \text{noCycl}_{\text{ISS}} - \text{noCycl}_{\text{OS model/timed native}} \right\|}{\text{noCycl}_{\text{ISS}}} \times 100\% \qquad (1)$$

where $\text{noCycl}_{\text{ISS}}$ is the number of simulation cycles obtained in ISS-based cosimulation, $\text{noCycl}_{\text{os model/timed native}}$ is the number of simulation cycles obtained for the cosimulation with OS model/timed native execution model.

In the cosimulation with timed native execution models, we obtained 17.2% (McDrive system case) and 17.4% error (VDSL system case) with respect to the ISS-based cosimulation. It is much smaller than the error of OS model-based cosimulation, 31.4% (McDrive system case) and 30.9% (VDSL system case), respectively. This accuracy improvement results from simulating the real OS and HW/SW interactions in the timed native execution model while OS model-based cosimulation emulates only the OS API, not the real OS execution.

Compared with ISS-based cosimulation, the source of error in the timed native execution model consists mainly in delay estimation. In our experiment, we take a simple delay estimation method explained in Section 3.2. If the delay estimation error is reduced by advanced delay estimation techniques as presented in Lajolo et al. (1999) and http://www.cadence.com/products/vcc.html, we expect further improvement in cosimulation accuracy.

5 Conclusion

This paper presents a method of HW-SW cosimulation using a timed native execution of real OS and application SW for SW simulation. The basic principle of timed native execution is (1) to natively execute real SW code (including real OS code) on a simulation host and (2) to simulate both the timing of SW execution and HW/SW interactions. ChronoSym tool was built to implement this method. The experiments prove that timed native execution enables 2.1 to 10.8 times speed-up in HW/SW cosimulation compared with ISS-based cosimulation with only about 17% timing error.

As future developments, in terms of simulation accuracy, we will integrate advanced delay estimation techniques to ChronoSym tool. We will also apply ChronoSym to more complex systems.

References

ARM7 Technical Reference Manual (REV 4) (2001) ARM Limited, 17th April, http://www.arm.com.

Bradley, M. and Xie, K. (2002) 'Hardware/software co-verification with RTOS application code', http://www.techonline.com/community/techtopic/21082.

Cesario, W. *et al.* (2002) 'Component-based design approach for multicore SoCs', *Proc. DAC*, pp.789–794.

Desmet, D., Verkest, D. and De Man, H. (2000) 'Operating system based software generation for systems-on-chip', *Proc. DAC*, 05th–09th June, Los Angeles, California, USA, pp.396–401.

Diaz-Nava, M. and Okvist, G.S. (2001) 'The zipper prototype: a complete and flexible VDSL multi-carrier solution', *ST Microelectronics Journal Special Issue xDSL*, September.

Gerstlauer, A., Yu, H. and Gajski, D. (2003) 'RTOS modelling for system level design', *Proc. DATE*, March, pp.10130–10135.

Lajolo, M., Lazarescu, M. and Sangiovanni-Vincentelli, A. (1999) 'A compilation-based software estimation scheme for hardware/software co-simulation', *Proc. CODES*, pp.85–89.

Rowson, J. (1994) 'Hardware/software co-simulation', *In Proceedings of the Design Automation Conference (DAC)*, June, pp.439, 440.

Zivojnovic, V. and Meyr, H. (1996) 'Compiled HW/SW co-simulation', *Proc. DAC*, pp.690–695.

Notes

[1]In this case, wait() is considered as an OS API function.

[2]In this case, wait() is also considered as an OS API function.

[3]In the case of ARM processor, the directive _swi() is used when a function is declared as a software interrupt service routine.

[4]The condition code flags (N, Z, C, V) are not simulated by the HAL simulation model.

[5]Cur and mode represent the index of current task and the current processor mode (mode = CPSR, mode).

[6]The McDrive example is a vending machine used in a drive-in restaurant.

Websites

ConvergenSC, http://www.coware.com.

Seamless CVE, http://www.mentor.com.

System Studio, http://www.synopsys.com.

SystemC, http://www.systemc.org.

Virtual Component Codesign, Cadence Design Systems Inc. http://www.cadence.com/products/vcc.html.

VxSim, Windriver Systems Inc., http://www.windriver.com/products/html/vxsim.html.