Automatic Generation of Software TLM in Multiple Abstraction Layers for Efficient HW/SW Co-simulation

Meng-Huan Wu, Wen-Chuan Lee, Chen-Yu Chuang, and Ren-Song Tsay
Department of Computer Science
National Tsing Hua University, HsinChu, Taiwan
{mhwu, wclee, cychuang, rstsay}@cs.nthu.edu.tw

Abstract

This paper proposes a novel software Transaction-Level Modeling (TLM) approach for efficient HW/SW co-simulation. In HW/SW co-simulation, timing synchronization should be involved between the hardware and software simulations for keeping their concurrency. However, improperly handling timing synchronization either slows down the simulation speed or scarifies the simulation accuracy. Our approach performs timing synchronization only at the points of HW/SW interactions, so the accurate simulation result can be achieved efficiently. Furthermore, we define three abstraction levels of software TLM models based on the type of interactions captured. Given the target software, the software TLM models can be automatically generated in multiple abstraction layers. The experimental results show that our software TLM models attain 3 million instructions per second (MIPS) for low-level abstraction and go as high as 248 MIPS for higher level abstraction. Therefore, designers can have efficient co-simulation by selecting a proper layer according to the abstraction of corresponding hardware components.

1. Introduction

As the design complexity of SoC grows, HW/SW co-simulation becomes more and more crucial for early-stage system verification. To simplify the simulation efforts on RTL designs, the concept of Transaction-Level Modeling (TLM) [1] for hardware was introduced. By adopting higher abstraction modeling, hardware simulation can be greatly accelerated while key operational information is maintained at the same time. Nevertheless, software is an essential system component, and it also requires proper abstraction models to be compatible with hardware TLM models for efficient HW/SW co-simulation. In particular, industrial studies show that the complexity of embedded software is rising 140 percent per year, which is greater than that of hardware at 56 percent per year [6]. Obviously, abstraction for software is an urgent subject for investigation, but unfortunately very few publications have addressed this issue. The main contribution of this paper is to propose a systematic and practical approach for abstracting software TLM.

TLM is formally defined as a high-level approach to model digital systems where the communication among modules is separated from the functional units [1]. However, a conventional software abstraction approach emphasizes different granularities of timing annotation, i.e., *cycle level*, *instruction level*, *basic-block level*, *function level*, and so on. Such a timing granularity approach is inadequate to model the communication among hardware and software.

Based on our observation, to efficiently achieve accurate HW/SW co-simulation results, timing synchronization is required between the hardware and software simulations for keeping their interactions executed in order. The points where interactions *may* occur (named *interaction point*) can be different, dependent on the modeling abstraction. More interactions would introduce more synchronization efforts. According to the type of interaction points of interest, we propose three levels of software TLM. At each particular abstraction layer, the execution order of given interactions is ensured by our timing synchronization so that the accurate simulation can be guaranteed. Figure 1 summarizes the correlation between the abstraction degree and simulation speed for the three proposed models.

- **Instruction-Level Model (I-LM):** Since a processor interacts with other components via memory access, hardware I/O, or interrupts, which may occur at any instruction, each instruction should be considered an interaction point.
- **Data-Level Model (D-LM):** In fact, a program's execution result can be influenced by other components only via data input/output. In other words, a program's interactions go through data accesses, so each data access instruction is an interaction point.
- **Shared-Variable-Level Model (SV-LM):** The data of a program can be further classified into shared variables and local variables. Logically, a program interacts with others only through shared variables so that shared variable access instructions should be taken as interaction points.

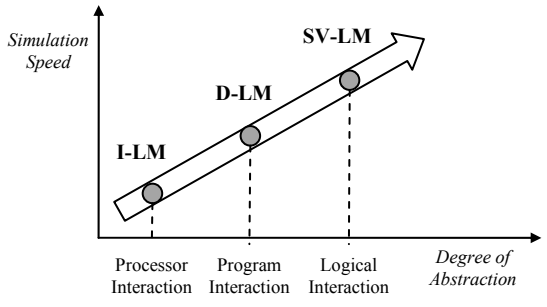


Figure 1: The correlation between the degree of abstraction and simulation speed for the three proposed software TLM models.

978-3-9810801-6-2/DATE10 © 2010 EDAA

Obviously, as the degree of abstraction rises, some irrelevant interactions are filtered out, so the number of interaction points decreases. With less synchronization overheads, a higher abstraction layer offers better simulation performance. Most importantly, it disregards detailed interactions and mulls over only those interactions of concerns to this layer. Thus, if a proper software TLM model is chosen, then both the desired accuracy and simulation performance of co-simulation can be accomplished.

In order to help designers efficiently create multiple abstraction layers of software TLM, we further devise an automatic generation method. Given target binary codes, the three proposed software TLM models can be automatically generated into corresponding SystemC [1] modules, which can easily be integrated with the hardware TLM models. Conventionally, an instruction-set simulator (ISS) is used for software simulation, but its maximum simulation speed is only few million instructions per second (MIPS). When cooperating with the hardware simulator, the timing synchronization between them would further slow down the simulation to less than one MIPS. On the contrary, our software TLM models (including synchronization overheads) attain 3 MIPS for I-LM and go as high as 248 MIPS for SV-LM as reported by the experimental results. This can demonstrate the effectiveness of our approach for HW/SW co-simulation.

The remainder of this paper is organized as follows. Section 2 discusses related work. Our timing synchronization is introduced in Section 3. Section 4 formally defines three abstraction layers for software TLM. The automatic generation is described in section 5. The experimental results are shown in section 6. Finally, our conclusion is in section 7.

2. Related Work

For HW/SW co-simulation, a conventional approach integrates an ISS and SystemC. To enable the communication between the two different simulators, Séméria and Ghosh [2] employed a bus functional model as a bridge. Similarly, Fummi et al. [3] adopted a wrapper to do so. However, the ISS is quite slow (few MIPS only), and the expensive communication cost further downgrades the simulation speed. In general, the performance of ISS-SystemC co-simulation is unsatisfactory. Even the high speed ISS like binary translation [7] would be significantly slowed down due to the heavy communication.

Rather than running on an ISS, the target source program is taken as a SystemC module to achieve native execution speed for software simulation. Then the two different simulations can be integrated smoothly, and the communication cost between them is greatly reduced. As an example, Schnerr et al. [4] proposed a timing back-annotation technique to produce a timed source program. Furthermore, Yoo et al. [5] employed *delay* functions onto the target source program for timing synchronization with SystemC. Nevertheless, this approach compiles target source codes by the host compiler, so target instructions are unavailable. It would make HW/SW interaction points unable to be accurately distinguished. Thus, the source-level software model is incapable of supporting HW/SW co-simulation comprehensively.

Obviously, the above mentioned approaches cannot meet the need of efficient and accurate timing synchronization in HW/SW co-simulation. Moreover, as far as we know, these approaches are insufficient to fully support the multiple abstraction layers for software models. This paper proposes a novel approach for multiple abstraction layers of co-simulation. To explicate the idea, the timing synchronization issue is first elaborated.

3. Timing Synchronization

To have accurate HW/SW co-simulation, it is necessary to simulate the *concurrency* of hardware and software so that *timing synchronization* is required between hardware and software for keeping their time consistent. To handle synchronization, a simulator like SystemC provides a timing synchronization function (i.e., *wait* function). When this function is called, a scheduler will be invoked to select a proper simulated component to execute. Therefore, the concurrency of simulated components can be cooperatively performed.

Ideally, timing synchronization should be performed at each cycle as shown in Figure 2(a). In each period of time, the simulations of hardware and software can be executed in order. However, the weighty synchronization overheads would significantly slow down the simulation, so how to

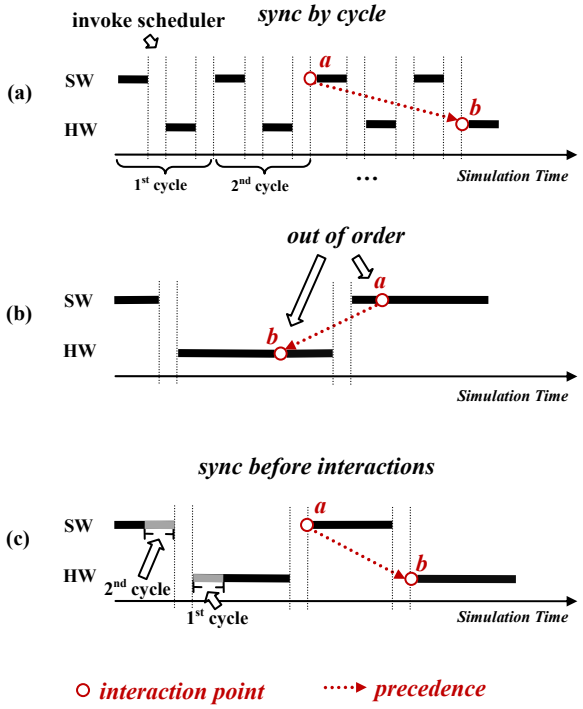


Figure 2: (a) timing synchronization by each cycle, (b) an improper granularity of timing synchronization, (c) timing synchronization before each interaction points.

reduce synchronization overheads becomes crucial.

Yet, if the granularity of timing synchronization is enlarged improperly, the simulation will be incorrect as illustrated in Figure 2(b). We know that any two different simulated components influence each other through mutual interactions (e.g., interaction *a* and *b* in Figure 2). Assume that the interaction *b* of the hardware obtains a value produced by the interaction *a* of the software in Figure 2(a). Then their out-of-order execution in Figure 2(b) would let *b* obtain the value not produced by *a*, which leads to incorrect simulation results.

To overcome this issue, we define the *precedence* of interactions based on their time ordering. By keeping the precedence, the influence from interactions must be maintained, and the correct simulation results are guaranteed thereby. To do so, we perform timing synchronization before each interaction point where an interaction may occur, as illustrated in Figure 2(c). In this way, part of the simulation in between interaction points may be executed out of order. For example, as depicted by the shaded regions in Figure 2(c), the second cycle of the software simulation, supposed to be executed later than the first cycle of the hardware simulation, is actually completed earlier. Nevertheless, since no other operations, except interactions, can directly influence (or be influenced by) other simulated components, the execution order of such operations makes no difference to the results.

In conclusion, a *transaction* can be defined as a sequence of operations, starting from an interaction, including succeeding operations, until the next interaction. Essentially, each transaction can be regarded as an atomic action, and the end of a transaction is a *sync point*. Because the number of interaction points is considerably smaller than the number of cycles, the synchronization overheads are greatly reduced. As a result, our synchronization scheme allows better simulation performance without sacrificing accuracy.

4. Software Abstraction

Based on the previous discussion, we know that the interaction point is the key to synchronization. Moreover, according to the software abstraction, there can be different types of HW/SW interactions, so the interaction points actually to be considered are also different. Therefore, we further propose three levels of software TLM corresponding to different abstraction layers of HW/SW interactions.

4.1 Instruction-Level Model (I-LM)

Considering a software component executes on a processor, interactions of a processor are introduced by memory accesses, hardware I/O accesses, and interrupts.

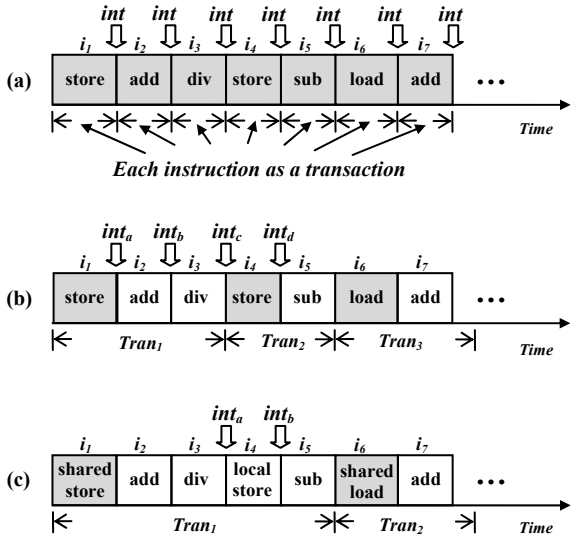
Memory accesses and hardware I/O accesses are the two ways for a processor to trigger an interaction. Typically, a memory access is via a memory load/store instruction. On the other hand, hardware I/O has two common mechanisms, *memory-mapped I/O* (MMIO) and *port-mapped I/O* (PMIO). For MMIO, hardware I/O accesses are also activated through memory load/store instructions to particular addresses (which are actually mapped onto hardware registers), whereas for PMIO, they are through I/O specified instructions instead.

Correspondingly, hardware components issue interrupts to interact with a processor (software). Since interrupts may occur at any processor instruction, to capture interrupts in correct order, timing synchronization has to be performed at each instruction. Therefore, for this model, each instruction is an interaction point and is also treated as a transaction, as depicted in Figure 3(a). Accordingly, it is named *instruction-level model* (I-LM). This fine-grained model intends to handle interrupts precisely, but the excessive synchronization effort would dominate the simulation performance. To accelerate the simulation, the interactions need to be considered in a higher abstraction layer.

4.2 Data-Level Model (D-LM)

For this model, data accesses, issued by either memory load/store instructions or I/O specific instructions, are considered interaction points. In terms of the program execution, only data accesses can directly influence the execution result. Hence, they are appropriate interaction points.

Interrupts from hardware are not treated as interactions points here since they do not change the program results immediately. Indeed, they may influence program execution, but their influence is implicitly through the data accesses issued by the interrupt service routine (ISR) or the other program they trigger. Therefore, as long as the order of data accesses is guaranteed, the program execution results must be correct. Accordingly, we define a *data-level model* (D-LM), in which timing synchronization is performed before each data access.



Note: the shaded instructions are interaction points.

Figure 3: Illustration of the relation between transactions and interrupts respectively in (a) I-LM, (b) D-LM, and (c) SV-LM.

To illustrate the effectiveness of this model, Figure 3(b) depicts an example of various interrupt timings to the program execution results. When a processor receives an interrupt, it will suspend the current program and invoke the corresponding ISR. The ISR can only influence (or be influenced by) the suspended program through its data access. Since the instructions within a transaction (such as i_2 (add) and i_3 (div)) access registers only, invocation of an ISR cannot affect their results. Consequently, when an interrupt that arrives at time point int_a or int_b is deferred to the end of $Tran_1$ (i.e., int_c), the execution result of the transaction remains the same. Meanwhile, the ISR will not be affected by instruction i_2 and i_3 as well, so the deferred handling does not affect the result of the ISR either. On the contrary, when an interrupt is handled in a different transaction, it may introduce erroneous simulation results. For instance, if both instruction i_4 (store) and the ISR of a particular interrupt access a same data address, then handling the interrupt at time point int_c or int_d would lead to different results. Accordingly, the handling of interrupts must be synchronized before data accesses.

In summary, D-LM has a higher abstraction layer on HW/SW interactions in contrast to I-LM. Since D-LM requires less synchronization efforts, it allows better simulation performance. Most importantly, D-LM can perform the same simulation results (including the interrupt effect) as I-LM does. Hence, it is a preferable model.

4.3 Shared-Variable-Level Model (SV-LM)

For this model, only data accesses to share variables are considered interaction points. In general, the data of a program can be further classified into *shared variables* and *local variables*. Logically, the value changes of local variables of a program do not affect the behaviors of others since programs interact with each other only through shared variables. Hence, focusing on shared variables, we define a *shared-variable-level model* (SV-LM), in which timing is synchronized before each shared variable access.

An example is shown in Figure 3(c), where $Tran_1$ and $Tran_2$ are separated by the shared access instruction i_6. If D-LM is applied, $Tran_1$ will be further divided into two transactions by the local access instruction i_4. Now with SV-LM, since i_4 accesses a local variable, whether an interrupt arrives before i_4 (int_a) or after i_4 (int_b) would make no difference to the program execution results.

Comparing with D-LM, SV-LM is even more efficient while it ensures the accuracy of logical interactions. However, this further abstraction of interactions ignores the local variable accesses. Although these accesses are logically irrelevant to others, they still share the same data bus with the shared variable accesses physically. Once the bus contention happens, the latency of the shared variable access can be affected by a local variable access. Consequently, SV-LM is proper to those who employ an ideal memory model without contention.

5. Automatic Generation

The three levels of software TLM have been decribed. Furthermore, in order to help designers efficiently create the software models in multiple abstraction layers, an automatic generation method is devised. We first establish the timed model. Then we identify the interaction points given different abstraction layers and annotate the timing synchronization function before them.

5.1 Input

Since we need the complete information of the target software to generate the software models in different abstraction layers, the *target binary codes* are ideal to be our input. Comparing with source codes, the binary codes contain the details about both the target instructions and the data layout in the target memory space, which allow us to identify the HW/SW interactions precisely. In addition, target binary codes are supposed to be available because the target processors are usually determined at co-simulation phase. Hence, the target binary can be easily generated by target cross compilers.

5.2 Timed Model

First, we generate the *functional model* of software by decompiling the target binary codes into C codes as illustrated in Figure 4, where each target instruction is translated into a corresponding C function. In addition, each basic block of the *control flow graph* (CFG) from the target binary codes is generated as a switch case. Then a *switch* statement is used to select proper blocks to execute based on the value of *program counter* (PC) during simulation. In this way, the generated model can execute the correct execution flow. Compared with an ISS which simulates an instruction by performing three steps, *fetching*, *decoding*, and *executing*, our method finishes fetching/decoding at the de-compilation stage. Only executing is required at the simulation stage so that the simulation performance is greatly improved. This is similar to static compiled simulation techniques [8][9].

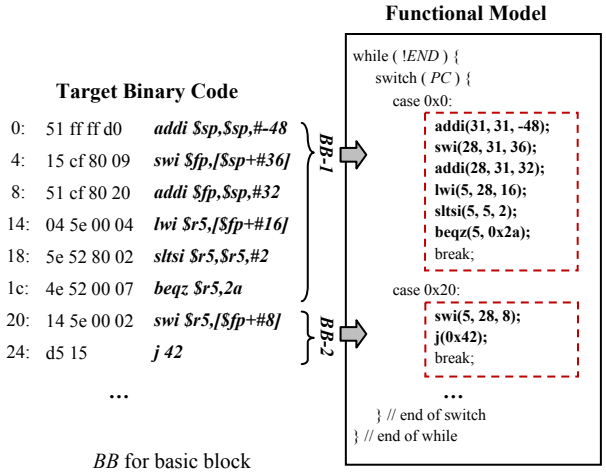


Figure 4: The de-compilation for the functional model.

The timed software model is needed for accurate timing synchronization. In order to have the *timed model*, we adopt the timing annotation technique [10], by which the timing information can be obtained without considerably downgrading the simulation performance. To do so, the essential execution time of each basic block is estimated in advance. Then during simulation, the overall execution time will be calculated by summing up the time of every executed block. Besides, some dynamic behaviors may also influence the execution time, such as cache hit/miss, branch prediction. To deal with these cases, we should further employ the corresponding correction codes to dynamically adjust the execution time. Consequently, the timed model is obtained.

5.3 Synchronization Model

In this stage, we identify sync points according to each different abstraction layer and respectively annotate them into the timed model.

5.3.1 I-LM & D-LM

For I-LM, the sync point is annotated in front of each instruction, whereas for D-LM, the sync point is annotated in front of each memory access instruction and each I/O specific instruction.

5.3.2 SV-LM

It is more complex to identify SV-LM's sync points since we are not always able to pre-determine whether a data access is for a shared variable. Thus, we mark those undetermined ones as *potential sync point*s. A potential sync point contains an extra procedure to check if the point is an actual sync point during simulation. For further discussion, we classify shared variables into two types: *SW/SW share* and *HW/SW share*.

SW/SW Share: Programs share data with each other via memory. The SW/SW shared variable access is issued by a memory access instruction. However, memory access instructions typically adopt the indirect address mode (i.e. memory address indicated by a register instead of an immediate). Consequently, the exact accessed address cannot be known until the instruction is ready for execution, so we use the check procedure to see if their accessed address belongs to the shared data segment. Then during simulation, the SW/SW shared variable access can be identified.

HW/SW Share: The shared variables between hardware and software are located in either memory or hardware registers. For those in memory, the space they store is usually pre-defined. Similarly, a check procedure can be applied to distinguish them. On the other hand, when the shared variables are in hardware registers, I/O specified instructions used for PMIO are pre-determinable. As for MMIO, normal memory access instructions are used, so a check procedure is needed again.

Based on the above principles, the shared variable access can be correctly identified, so the sync points of SV-LM are determined. Accordingly, the three proposed abstraction software TLM models are generated.

5.4 Limitations

Our automation method inherits the restrictions of the static compilation technique i.e., the target program must be *run-time static* and have no *indirect jump*. If the program goes against them, an extra interpretive simulator can be used to handle such special cases. Fortunately, since these cases are only a small portion of the program, the simulation performance is not greatly affected.

6. Experimental Results

To evaluate the proposed software TLM, we did two experiments. The first demonstrates the simulation speed of our software TLM models; the second tests a real case of HW/SW co-simulation. The setup of both experiments is as follows: the testing machine is equipped with Intel Xeon 3.4 GHz quad-core and 2GB ram. Our target processor adopts Andes instruction-set architecture [12]. The generated models are simulated on the SystemC 2.2.0 kernel.

6.1 Performance of Software TLM

The first experiment evaluates the simulation speed of the three different software TLM models. A *wait* function is annotated for the SystemC scheduler to do timing synchronization. Without any synchronization, the ideal speed of generated software TLM models attains hundreds of MIPS. Table 1 makes a comparison with a typical ISS [13] which is also without synchronization. The speed of the ISS is only about hundreds of KIPS to few MIPS as reported. This is because an ISS has to perform *fetching*, *decoding*, and *executing* for each time of the execution of one instruction while we only perform once of fetching/decoding for a same instruction no matter how many times it is executed. Since there are a lot of loops within common programs, the time spent on fetching/decoding is greatly reduced in our model. Therefore, we can outperform the ISS by two to three orders of magnitude in simulation speed.

Table 1. The ideal speed comparison with ISS

Our software TLM	ISS [13]
136 ~ 248 MIPS	200+KIPS ~ 4+ MIPS

The simulation speeds of our software TLM in different abstraction layers are shown in Figure 5. Here we tested them by five different benchmarks. FFT, LU, and RADIX are parallel programs from SPLASH-2 benchmarks [10] while Micro-benchmark and Fibonacci are sequential programs. The simulation performance of I-LM dominated by the synchronization overheads is consistently at around 3 MIPS despite different benchmarks. As for D-LM, synchronization is just required at data accesses, so it can accelerate up to 6 ~ 13 MIPS. Moreover, since SV-LM only synchronizes at shared variable accesses, the simulation speed is further raised to 37 ~ 248 MIPS. Especially for the

sequential benchmarks (without shared-variable access), the simulation speed can be as high as the ideal one.

It is obvious that by decreasing the considered interaction points, higher abstraction would greatly improve the simulation performance.

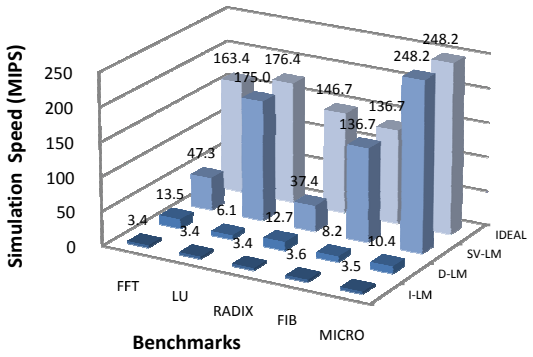


Figure 5: Simulation speeds in different software abstraction models.

6.2 HW/SW Co-simulation

In this experiment, we use a real case to test HW/SW co-simulation. As shown in Figure 6, our software TLM model, embedded into a processor module with a MMU and a cache, is co-simulated with a hardware JPEG encoder in the form of the timed TLM model. The target software, which runs a driver for the JPEG encoder, moves the encoded images from the encoder and then decodes the images. To comprehensively demonstrate the behaviors of our software TLM, we compare two common scenarios, *polling* and *interrupt-driven*, for HW/SW interactions. Table 2 shows that in both modes, a software TLM model in a higher abstraction layer has fewer transactions, and the simulation time is shorter.

Table 2. Co-simulation result comparison

	Polling		Interrupt-driven	
Model	**Trans. Count**	**Sim. Time (s)**	**Trans. Count**	**Sim. Time (s)**
I-LM	**14,254,971**	**4.309**	**4,240,657**	**1.352**
D-LM	**6,256,378**	**2.089**	**1,965,228**	**0.685**
SV-LM	**1,381,914**	**0.720**	**1,381,765**	**0.584**

Compared with the interrupt-driven mode, more than three times the amount of instructions is simulated in the polling mode since it has to do busy waiting until the completion of the hardware encoder. Consequently, the polling mode requires more simulation time than the interrupt-driven mode. With our approach, the software models can be automatically generated and be easily integrated into a co-simulation environment.

7. Conclusion

In this paper, we have proposed automatic software TLM modeling in multiple abstraction layers for efficient HW/SW co-simulation. The performance of co-simulation is determined by the software modeling, the hardware

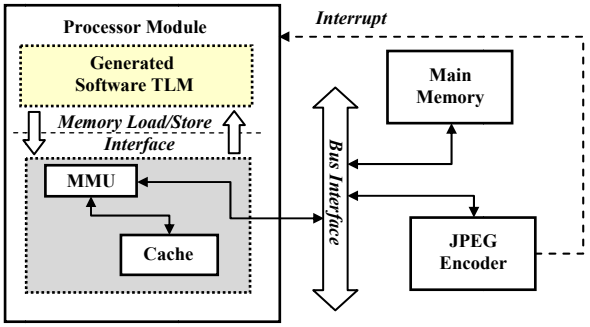


Figure 6: The co-simulation of a generated software TLM and a hardware JPEG encoder.

modeling, and the synchronization between the two. This work contributes to the efficiency of software modeling and synchronization. As the experiments show, the generated software TLM allows high speed simulation. The full support of multiple abstraction layers provides the possibility of better performance by reducing synchronization overheads while maintains desirable accuracy.

8. Acknowledgements

This work was supported by National Science Council (Grant No. NSC96-2628-E-007-144-MY3) and the specification of Andes ISA was provided by Andes Technology.

REFERENCES

[1] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer Academic Publishers, 2002.

[2] L. Séméria and A. Ghosh, "Methodology for hardware/ software co-verification in c/c++," in *ASP-DAC '00*. pp. 405-408, 2000.

[3] F. Fummi, S. Martini, G. Perbellini, and M. Poncino, "Native iss-systemc integration for the co-simulation of multi-processor soc," in *DATE '04*. pp. 564-569, 2004.

[4] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," in *DAC '08*. pp. 290-295, 2008.

[5] S. Yoo et al., "Building fast and accurate sw simulation models based on hardware abstraction layer and simulation environment abstraction layer," in *DATE '03*. pp. 550-555, 2003.

[6] P. Magarshack and P. G. Paulin, "System-on-chip beyond the nanometer wall," in *DAC '03*. pp. 419-424, 2003.

[7] M. Wu, C. Fu, P. Wang, and R. Tsay, "An effective synchronization approach for fast and accurate multi-core instruction-set simulation," in *EMSOFT '09*, pp. 197-204, 2009.

[8] J. Zhu and D. Gajski, "A retargetable, ultra-fast instruction set simulator," in *DATE '99*. pp. 62-69, 1999.

[9] M. Burtscher and I. Ganusov, "Automatic synthesis of high-speed processor simulators," in *MICRO '04*. pp. 55-66, 2004

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *ISCA '95*. pp. 24-36, 1995.

[11] J. Schnerr, O. Bringmann, and W. Rosenstiel, "Cycle accurate binary translation for simulation acceleration in rapid prototyping of socs," in *DATE '05*. pp. 792-797, 2005.

[12] Andes, available at www.andestech.com

[13] SimpleScalar, available at www.simplescalar.com