An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles

Emmanuel Viaud, François Pêcheux, Alain Greiner
Laboratoire LIP6/ASIM
Université Pierre et Marie Curie
{emmanuel.viaud, francois.pecheux, alain.greiner}@lip6.fr

Abstract

The paper presents an innovative simulation scheme to speed-up simulations of multi-clusters multi-processors SoCs at the TLM/T (Transaction Level Model with Time) abstraction level. The hardware components of the SoC architecture are written in standard SystemC. The goal is to describe the dynamic behavior of a given software application running on a given hardware architecture (including the dynamic contention in the interconnect and the cache effects), in order to provide the system designer with the same reliable timing information as a cycle accurate simulation, with a simulation speed similar to a TLM simulation. The key idea is to apply Parallel Discrete Event Simulation (PDES) techniques to a collection of communicating SystemC `SC_THREAD`*. Experimental results show a simulation speedup of a factor up to 50 versus a BCA simulation (Bus Cycle Accurate), for a timing error lower than* 10^{-3}.

1. Introduction

With the advent of more and more complex SoCs, and GALS architectures containing dozens of processor cores, the need for high-speed simulation is of paramount importance. Unfortunately, simulation speed and accuracy are two antagonistic goals. Up to recently, high speed simulation was synonymous to high level functional simulation whereas reliable performance evaluations could only be achieved at the BCA level. This lack for an intermediate level is at the origin of the creation of a new abstraction level known as TLM (Transaction Level Modeling) "which abstracts the pin-level communication in the physical model to the level of media access or individual protocol word/frame transactions"[7].

The pure TLM abstraction level is mainly used to validate the embedded (generally parallel and multi-threaded) application software. It is basically a functional validation, as these models do not contain any timing information. This abstraction level allows a simulation speedup of one or two order of magnitude versus the BCA simulation level.

In the TLM/T (TLM with Time) abstraction level, the system designer wants to answer questions such as whether the selected hardware architecture matches the application real time constraint. A reliable answer to this question requires taking into account the dynamic behavior of the hardware, including accurate modeling of the dynamic contention in the shared interconnect, or accurate modeling of the cache misses.

In the paper, TLM/T modeling of SoCs implicitly refers to a shared memory architecture, where initiators (or masters) send requests to addressable targets (or slaves), and obtain responses (data in case of read, or acknowledge in case of write). A transaction is actually a request/response couple. Due to the limited bandwidth of the classical system bus, new kinds of interconnects (Network on Chip) have been developed, that offer scalable bandwidth. In such micro-networks, requests and responses are generally handled by two separate dual sub-networks, in order to avoid deadlocks.

The paper demonstrates that it is possible to obtain the same level of timing information as the "exact" timing behavior described by the cycle accurate BCA simulation, but with a simulation speed similar to the speed of a TLM simulation, and a loss of accuracy less than 10^{-3}.

The major problem in the definition of this TLM/T abstraction level comes from the simulation speed versus accuracy trade-off: it is quite hard to distinguish *a priori* the mechanisms that must absolutely be taken into account and those that can be neglected at the first order.

As our goal is to describe the dynamic behavior of a given hardware platform executing a given software application, we must emphasize on the fact that the simulation TLM/T simulation models describe hardware components, not software tasks. Therefore, we must define a TLM/T model for each hardware component in the architecture: one TLM/T model for the general purpose processor (including the caches), one TLM/T model for the dig-

3-9810801-0-6/DATE06 © 2006 EDAA

ital signal processor, one TLM/T model for the embedded memory bank, one TLM/T model for each peripheral, and of course, one TLM/T model for the interconnect itself. Fortunately, most of these hardware components are generic, reusable IP cores, and therefore the modeling effort is justified.

The paper is composed of five sections. After a brief introduction, section 2 presents the relevant work in the TLM and TLM/T modeling domains. section 3 introduces the paradigm of the PDES applied to TLM/T modeling and details the modeling choices for the key components. section 4 presents some experimental simulation results for a generic multi-processors platform, in terms of simulation speed and accuracy. Finally, the section 5 comments the previous results and gives some perspective.

2. Previous work

The exact place of TLM in the simulation level landscape is still not clear: TLM is not a single abstraction level, but regroups a rather large set of modeling practices. Cai and Gajski in [3] or Donlin in [6] propose a taxonomy. Cai and Gajski represent those levels as a 2D mesh in a plane oriented by the time accuracy sought for the simulation of the communication and of the computation. Pasricha [9] also presents some of the interests of the TLM methodology for both embedded software development and architecture exploration.

The most significant proposal for TLM standardization in our context is the one proposed by the Open SystemC Initiative in [12]. Unfortunately, this proposal is limited to TLM level and does not address timing issues at all.

[8] proposes a methodology for TLM simulations that bears some resemblance with our proposed method, most notably in the idea of *passive components* that are run only when needed. But nothing is specified about adding timing informations to the simulation.

The CCATB (Cycle Count Accurate at Transaction Boundaries) level presented in [10] is a method to fill the gap between functional level and BCA level. But this level of description is still rather close to the BCA level, and the expected speedup is limited (about a factor 2).

3. Applying PDES to TLM/T modeling

A possible approach to introduce the time in TLM, relies on `SC_THREAD` and the `wait` construct to add timing into the component models, and uses the SystemC internal clock as the SystemC clock. Our experience in the domain of parallel algorithms led us to model the parallel behaviors of the components in a completely different way.

3.1. Principles

Hardware components are still represented by threads executed in parallel but a local simulation clock is now assigned to each active component. These threads communicate by sending "packets" through statically defined, point to point communication channels. It is worth saying that, in a shared memory SoC architecture, there are only three types of packets, because all the communications between hardware components are related to read/write requests by the initiators, or interrupt requests by the peripherals:

- a *request packet* is sent by an initiator to the interconnect, that delivers the packet to the selected target ;
- a *response packet* is sent by a target to the interconnect, that delivers the packet to the requesting initiator ;
- an *interrupt packet* is directly sent by a peripheral (generally a target) requesting a service to an initiator component.

For a given hardware architecture, all the communication channels are statically defined.

The local simulation clocks of two communicating threads T_0 and T_1 are synchronized each time a packet is transmitted from T_0 to T_1. Therefore, there is no global simulation clock and the local clock of a component advances only when this component has received timing information on each of its inputs. The inputs are actually ports connected to the communication channels.

In this paradigm, an initiator such as a general purpose processor which wants to make a request to the memory (due to a cache miss for example), sends a compound packet piggy-backed with its local simulation time. When the interconnect component receives the packet, it exploits this timing information to compute its own new local time. Before a given component modifies its own local time, it has to verify that every input channel is filled with at least one packet to ensure that no event will occur with a date that belongs to the component past. This corresponds to the standard and well-known conservative approach described in the literature by Chandy, Misra and Bryant in [5] and [2].

To be complete, PDES also mentions optimistic algorithms but this approach involves saving and restoring the internal states of the different components. We do not use this optimistic approach, because it consumes too much time and memory to save and restore the complete contents of a standard SoC RAM or cache.

From a SystemC viewpoint, the PDES approach implies that at least every initiator component is modeled as a `SC_THREAD` that can perform read and write operations and be stopped with a `wait` statement and resumed at any time by other threads.

Furthermore, as time is propagated through packets passing between threads, the SystemC simulation clock is no

longer used. The scheduling of the different threads is performed at the delta-cycle level.

The three main difficulties in this approach are the following:

- modeling of the on-chip interconnect, especially taking into account the dynamic contention ;
- modeling of the programmable processors, including the cache controllers ;
- modeling of asynchronous events (interrupts).

3.2. Interconnect Modeling

Modeling the effects of the dynamic contention is the main obvious problem. The main source of contention comes from access conflicts on the targets: when several initiators address simultaneously the same target, the requests must be sequentialized. This contention is intrinsic, and can be observed with any interconnect micro-architecture. There is, of course, other sources for contention. In a given interconnect, such as a multi-stage micro-network, one can observe internal contention: two request packets sent by two different initiators addressing two different targets can be conflicting for an internal switch in the multi-stage micro-network. This kind of contention is very specific to each micro-network and in all existing Network on Chip, the dynamic behavior is mainly related to the destination contention. For the sake of simplicity and genericity, the internal contention effects are neglected. With this hypothesis, it is not necessary to describe precisely the detailed protocol of a specific interconnect to reproduce the dynamic contention.

As presented in figure 1 a generic hardware interconnect has been defined. It behaves as two independent crossbars (one for the requests and one for the responses), with output FIFOs to emulate the buffering available inside the network. It is possible to adjust the depth of the FIFOs to fit rather accurately the dynamic behavior (especially the saturation threshold of any specific micro-network).

Another important feature is the following: if we consider separated subnetworks for requests and responses (this is a common feature to have distinct hardware resources to avoid deadlocks), the two sub-networks are not symmetrical. As a given initiator will expect a response from a unique target (if we consider that complex, multi-threaded processors can be modeled by several threads), there is no destination conflicts in the response sub-network.

Each sub-network is composed of two kinds of nodes. In figure 1, circles represent simple C++ functions while squares represent SystemC `SC_THREAD`s, with a local time *t*. The first kind of node at the input corresponds to *routing* blocks and performs the routing of the packets to the addressed target. There is one such block for each initiator component connected to the interconnect. The *arbitration* blocks are responsible for arbitration in case of conflicts. Calling M the number of initiators and S the number of targets linked to the interconnect, an interconnect is made of $M+S$ routing blocks (for the request network) and $S+M$ arbitration blocks (for the response network). The different blocks are linked by FIFOs that store the packets.

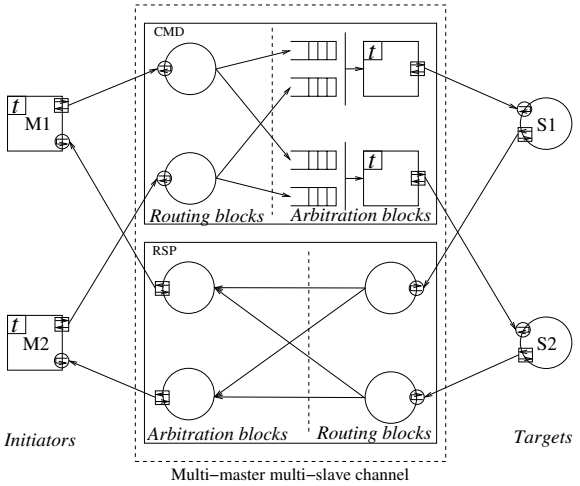

Figure 1. Structure of the interconnect

Let's detail the arbitration policy in the request subnetwork. As in most physical interconnects, the general arbitration policy is Round-Robin, in order to avoid starvation. But, to conform to the PDES algorithm, the timing information (timestamp) associated to each request packet must be taken into account. Every time a packet is stored in a FIFO, the corresponding arbitration block is awaken, and the the SELECT-FIFO() algorithm described below is executed. If there is no packet available in a given FIFO, the arbitration block asks the corresponding initiator its current local time and pushes a "false" packet in the corresponding FIFO. This is similar to "solicited null-messages", in the PDES general terminology, and prevents deadlocks to occur.

SELECT-FIFO()

```
1   goes through the M linked FIFOs
2   T_min = lowest packet timestamp in the FIFOs
3   T_ab = arbitration block time
4   if T_min < T_ab
5     then if FIFO[T_min] has a real packet
6             then select FIFO[T_min]
7                  if other packet(s) in the FIFOs
8                    then wait for a delta-cycle
9                    else wait for another packet
10            else wait for a delta-cycle
11    else if no FIFO has T_i < T_ab
```

```
12          then select FIFO[T_min]
13              if other packet(s) in the FIFOs
14                  then wait for a delta-cycle
15                  else wait for another packet
16          else wait for a delta-cycle
```

After the packet to be transmitted has been selected, its time is updated. If the time of the packet T_{min} is lower than the time of the arbitration block T_{ab}, it means that a contention previously occurred: the time of the packet is then updated to the time of the arbitration block. On the other hand, if T_{min} is higher than T_{ab}, it means that the target was idle in the recent past and therefore T_{ab} is set to the time of the packet. Finally, the packet is sent to the target through a function call. In order to take into account the target intrinsic latency, this function call returns a latency value which is used to update T_{ab}. Thus, the arbitration block time represents the local time of its linked target.

3.3. Components modeling

As described in figure 1, the internal structure of the TLM/T model depends on the role of the modeled component: it can be an initiator, a target or both (like a DMA controller that is both a target (to be configured by the host processor), and an initiator (to access the memory)).

An initiator component is modeled as a `SC_THREAD` and has its own local clock while a target component is modeled as a simple C++ function. Moreover, a target only reacts to request packets. When it receives a packet, it uses the piggy-backed time as the starting time of its task, then performs its work, sends back a response packet to the initiator and returns the updated time to the caller.

The structure of an initiator component is depicted in figure 2. An initiator component has a local time T_{thread}. It performs its computational task, updating accordingly its local time and generates request packets each time it must access the memory or a peripheral.

In case of a programmable processor, the behavior is merely described as an execution loop that keeps executing the binary code stored in the instruction cache, as an Instruction Set Simulator. In case of blocking events, such as instruction or data miss, or uncached read, a request packet is sent (with the local timestamp), the corresponding `SC_THREAD` is unscheduled and waits for the response packet.

A problem may arise if an initiator component works autonomously for a very long time, without sending any request packets (for example, the cache is large enough to contain the requested data or instructions). In this case, the initiator `SC_THREAD` is never unscheduled. The local clock can progress without limitations, and prevents the other threads (representing other initiators, or arbitration blocks) to do so. To prevent such a problem, a single parameter, global to the simulation has been defined: the *lookahead* time. The lookahead time acts as an upper time barrier for the initiator components and specifies how far an internal execution loop can run without any external synchronization. Past this upper time barrier, an initiator component unschedules itself.

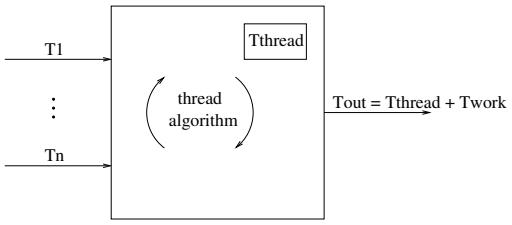

Figure 2. Structure of an initiator component

A master/slave component has to implement both aspects, and thus has a function to receive the packets which will launch the working thread according to the received requests.

3.4. Handling interrupts

The conservative scheduling of request and response packets in the interconnect ensures that no causality violation (a packet with a timestamp smaller than the local time arrives) may happen. But the interrupts are asynchronous events and must be handled differently. Of course, a purely pessimistic approach would avoid any causality problem by preventing the component to increase its local time as long as it is sure it cannot receive any interrupt in its past. Unfortunately, that method is not applicable as it inevitably creates a deadlock in the simulation. On the other hand, a classical optimistic approach which would restore the state of a component back to the interrupt time is much too expensive to be actually implemented.

The proposed answer is to handle an interrupt by polling. The interrupt packet is registered when it arrives (with its timestamp), and taken into account by the processor as soon as the processor time is larger or equal than the interrupt time. A pending interrupt request is polled at each turn of the initiator execution loop. The resulting inaccuracy remains an inherent drawback of the method but can be minimized by the lookahead parameter which limits the maximum time leap of each component.

4. Experimental results

To validate this simulation paradigm, we defined a generic multi-processors architecture, and executed the

same software application on two different simulation platforms (BCA & TLM/T) describing the same hardware architecture.

As shown in figure 3, the simulated hardware contains a generic, VCI/OCP [1] compliant interconnect, a variable number of processors (including data and instruction caches), two embedded memory banks (containing respectively the code and the data), a parameterized timer that can generate N independent interrupts, and a variable number of TTY terminals that can display messages produced by the embedded software by means of `printf`-like calls. Each processor is connected to its private interrupt line coming from the timer.

We developed two simulation environments for this architecture:

- The reference BCA simulation platform is build with the cycle-accurate, bit-accurate simulation models provided by the SoCLib project [13] ;
- The TLM/T simulation platform has been build by interconnecting the corresponding TLM/T components as described above.

All processors run the same software task: it waits for an interrupt, and displays a message on the TTY, indicating the IRQ index and date. The timer period is initialized by the embedded software, with a different value for each processor (depending on the processor identifier). In both cases the embedded software binary code is loaded in the embedded ROM before starting the simulation.

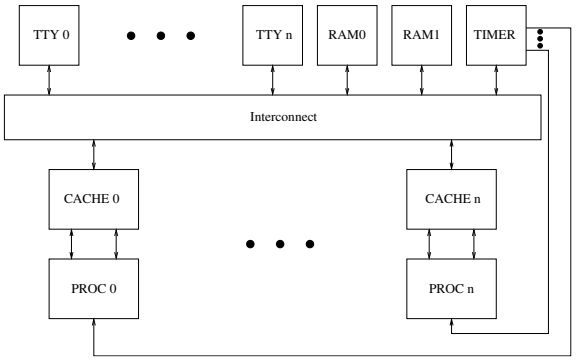

Figure 3. The generic platform

The comparison of the results is made on two points:

- the simulation speed
- the accuracy of the results

All the measurements were performed on a Linux workstation equipped with a Pentium IV processor running at 2.6 GHz with a L2 cache of 512 kb and 256 Mb of SDRAM. The SystemC version is the 2.1 beta 11. Each simulation has run at least one million cycles.

The table 1 gives the simulation speed for several architectures, from 9 to 39 processors. As can be seen on figure 4 the speedup factor of the TLM/T is comprised between 10 and 60, and increases with the size of the system.

	9 procs	15 procs	25 procs	39 procs
BCA	11600	4400	2100	1150
TLM/T	163000	101000	91000	74000

Table 1. Simulation speed for both platforms in cycle/s

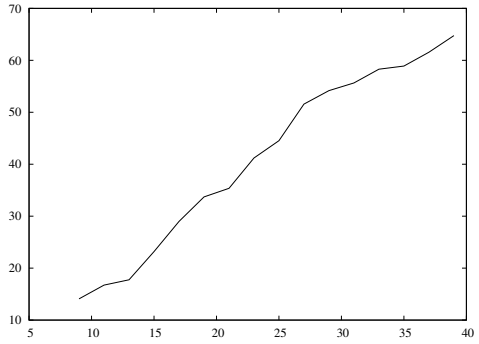

Figure 4. Speedup factor against the number of processors

Using the same method as presented in [4], the accuracy of the TLM/T simulation is defined as the maximum timing error between the two timestamps of the same identifiable event in both BCA and TLM/T simulations. In this platform, the interrupt times generated by the timer are good candidates for such events. To evaluate the loss of timing accuracy introduced by the TLM/T modeling we compared the interrupt service time (as the value displayed on the TTY), for both simulation and measured the maximal discrepancy (in number of cycles) for the same event in both simulations. The figure 5 shows the evolution of the maximal timing error against the number of processors in the platform. The value is always less than 800 cycles for a total duration of 1 000 000 cycles.

The promising results obtained so far can be explained by three factors. The first is directly linked to the fact that the TLM/T components are intrinsically lighter than their BCA counterpart. The access to a target is done through Interface Method Call which is much faster than the corresponding method in BCA. In TLMT, data movements are

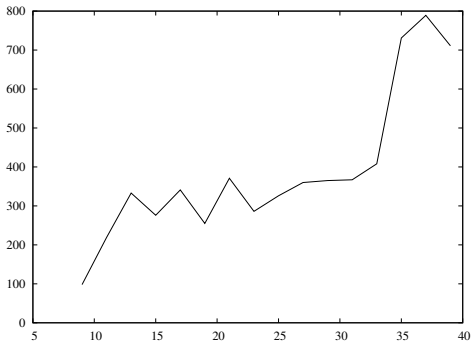

Figure 5. Timing error against the number of processors

represented by calls to functions like `memcpy` and pointer assignments while in BCA and traditional SystemC simulation, it requires the use of a discrete event simulation kernel: event list, delayed assignment of signals and re-triggering of sensitive processes.

The second factor is linked to the granularity of the data transfers. In the TLM/T simulation, the flow control is performed at the packet level (one slot in the interconnect FIFOs is exactly one packet), while the flow-control in the BCA simulation is at the word/cell level. In case of a cache line miss, a single request or response packet contains 8 words. This coarse grain flow control modeling introduces an acceptable timing inaccuracy, but improves the simulation speed.

Finally, because each initiator runs autonomously with its own local time, there is no need for a global synchronization scheme at each system cycle. Processors progress by leaps of several instructions, read complete cache lines atomically and therefore participate in the simulation only when they have to do so.

5. Conclusion

We demonstrated in this paper that the TLM/T modeling approach is able to provide the system designer with the same reliable timing information as a BCA simulation, with a loss of accuracy which is lower than 10^{-3}, at a simulation speed similar to the TLM functional simulation. This is a very promising result, as the system level simulation remains the main tool used to perform design space exploration.

Of course, systematic performance evaluation have still to be done on other multi-processors architectures, running more realistic software applications, but we are currently investigating the introduction of this simulation scheme into existing design methodologies and components through a collaboration with STMicroelectronics.

An open question is the modeling of multi-threaded processors, when the processor can issue several requests without waiting for response (i) before issuing request (i+1). Finally, we are working on a multi-level simulation tool that can dynamically switch between TLM/T and BCA simulation modes ("logical zoom").

Another interesting research direction would be to apply the presented methodology to a new SystemC kernel such as [11].

References

[1] *Virtual Component Interface Standard*. http://www.vsi.org.

[2] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, 1977.

[3] 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*, pages 19–24. ACM Press, 2003.

[4] F. Carbognani, C. K. Lennard, C. N. Ip, A. Cochrane, and P. Bates. Qualifying precision of abstract systemc models using the systemc verification standard. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 20088, Washington, DC, USA, 2003. IEEE Computer Society.

[5] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Softw. Eng*, 5(5):440–452, 1979.

[6] A. Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80. ACM Press, 2004.

[7] A. Gerstlauer, D. Shin, R. Dömer, and D. Gajski. Systel-level communication modeling for network-on-chip synthesis. In *ASP-DAC 2005*, page 45. IEEE Computer Society, 2005.

[8] Z. Kadi and P. Klein. Efficient passive-TLM and transaction management. In *First North American SystemC User's Group Conference*, 2004.

[9] S. Pasricha. Transaction Level Modeling of SoC with SystemC. In *Synopsys User Group Conference*, 2002.

[10] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *Proceedings of the 41st annual conference on Design automation*, pages 113–118, 2004.

[11] H. D. Patel and S. K. Shukla. Towards a heterogeneous simulation kernel for system level models: a SystemC kernel for synchronous data flow models. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 248–253. ACM Press, 2004.

[12] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez. Transaction Level Modeling in SystemC. http://www.systemc.org, January 2005.

[13] SoCLIB. A modelisation & simulation plat-form for system on chip, 2003. http://soclib.lip6.fr.