Efficient Parallel Transaction Level Simulation by Exploiting Temporal Decoupling

Rauf Salimi Khaligh and Martin Radetzki

Embedded Systems Engineering Group
Institute of Computer Architecture and Computer Engineering (ITI)
Universität Stuttgart
Pfaffenwaldring 47, D-70569 Stuttgart, Germany
{salimi,radetzki}@informatik.uni-stuttgart.de

Abstract. In recent years, transaction level modeling (TLM) has enabled designers to simulate complex embedded systems and SoCs, orders of magnitude faster than simulation at the RTL. The increasing complexity of the systems on one hand, and availability of low cost parallel processing resources on the other hand have motivated the development of parallel simulation environments for TLMs. The existing simulation environments used for parallel simulation of TLMs are intended for general discrete event models and do not take advantage of the specific properties of TLMs. The fine-grain synchronization and communication between simulators in these environments can become a major impediment to the efficiency of the simulation environment. In this work, we exploit the properties of temporally decoupled TLMs to increase the efficiency of parallel simulation. Our approach does not require a special simulation kernel. We have implemented a parallel TLM simulation framework based on the publicly available OSCI SystemC simulator. The framework is based on the communication interfaces proposed in the recent OSCI TLM 2 standard. Our experimental results show the reduced synchronization overhead and improved simulation performance.

Key words: Transaction-Level Modeling, Parallel Simulation, SystemC, Temporal Decoupling

1 Introduction

A transaction level model is a network of *modules* representing logical or physical entities. In TLM terminology, modules may be *active* or *passive*. Active modules contain processes, the fundamental unit of behavior and concurrency. Active modules perform computations and initiate communication (i.e. transactions) with other modules. Passive modules on the other hand do not contain processes and the functionality provided by them is executed by, and in the context of the processes of the active modules. Several factors contribute to the total time required for simulation of a given transaction level model: The **computation** performed by processes, the **communication** between modules

(e.g. via interface method calls), the **synchronization** between processes (e.g. *wait()-notify()*) and the **synchronization** of the processes with the simulation time (e.g. *wait(delay)*).

For parallel simulation of a TLM on N simulators, the set of all modules in the TLM is partitioned into disjoint sets, with each set being assigned to a single simulator. The total amount of processor time consumed by the computations in the modules can be considered the same in parallel and sequential simulations. The amount of time consumed by inter-module communication depends on whether the communicating modules are assigned to the same or different simulators. Inter-simulator communication can be orders of magnitude slower than communication between modules in the same simulator. The overhead of inter-simulator communication can be reduced by analysis of inter-module communication patterns and appropriate assignment of modules to simulators. For example, by assignment of modules with high communication requirements to the same simulator. This issue has not been in the scope of this work. We have focused on the overhead incurred by the synchronization between the simulators which is described in the following paragraphs.

Existing frameworks available for parallel simulation of TLMs are in intended for simulation of general, discrete event models such as signal-based, RTL models written in SystemC. In terminology of parallel discrete event simulation these simulation frameworks are all *conservative*. That is, they guarantee that the causality relationships are never violated. For example it is guaranteed that no process will ever be notified of an event which belongs to the past. To ensure this, the progress of individual simulators in simulation time must be synchronized globally.

Let $S_1, \ldots, S_N$ be N simulators and T be the current global simulation time. In summary, at T, in each simulator S_i all processes sensitive to the events belonging to T are executed. Each simulator then reports the time of its next event t_i to a global synchronization process and waits. After collecting all t_i, the synchronization process then determines the next global simulation time $T' = min\{t_1, \ldots, t_N\}$, such that the causality relationships are not violated (conservatism). It then broadcasts T' as the next global simulation time to all simulators and they proceed to T', executing any process sensitive to events at T'. This process is repeated until a global simulation termination condition is satisfied. Especially in timed TLMs, this will require frequent communication between the simulators and the resulting overhead will negatively affect the efficiency of the simulation.

In our approach, synchronization between simulators is performed only at fixed, globally known points in the simulation time. This results in a simplified synchronization algorithm which can be implemented using efficient collective synchronization operations (e.g. barriers on SMP machines). We achieve this by exploiting a special, strict form of temporal decoupling (allowing the processes to run ahead of the simulation time). We will elaborate on this in the next sections.

This paper is organized as follows: In section 2 we give an overview of closely related work. Section 3 summarizes the main ideas behind our approach. In

section 4 details of a SystemC-based and OSCI TLM 2 inspired implementation are presented. Section 5 shows the results of our experiments and section 6 summarizes the results and provides some direction for future work.

2 Related Work

Transaction level modeling [17, 4, 3, 9] is an already established and increasingly popular, simulation-centric modeling paradigm for embedded systems and systems-on-chip and is enabled by modeling languages such as SystemC [11], SpecC [7] and SystemVerilog [1]. Except for some special cases such as cycle-driven TLM simulators (e.g. [2]), currently most TLMs are simulated using sequential discrete event simulators (DES) such as the publicly available SystemC simulator from OSCI. Some researchers see the performance of such general simulation kernels insufficient for many applications. For example, some propose alternative simulation kernels (e.g. heterogeneous simulation kernel [18]) while others address this issue at the modeling level (e.g. adaptive models [12]). In the recent OSCI TLM 2 standard [17] communication interfaces, modeling guidelines and techniques such as temporal decoupling are proposed. The OSCI TLM 2 standard targets sequential TLM simulation, and temporal decoupling is recommended for reduction of the overhead of context switches caused by the synchronization of the processes with the simulation time (*wait(time)*).

Currently most of the parallel simulation environments for transaction level simulation are based on SystemC. These simulators are meant for general SystemC models and hence deal with low level synchronization and communication constructs such as signals and evaluate/update channels. One of the first works in this area is [5], where simulation kernels are synchronized at the end of every delta cycle to ensure causality and the evaluate/update channels are synchronized at the end of every update phase to ensure correctness of the communication. This simulation environment requires a modified SystemC kernel. The high overhead of this fine-grain synchronization and communication has been addressed by the authors in their recent work [6]. Another environment for parallel SystemC simulation is introduced in [10]. The simulation time synchronization in this environment is similar to [5, 6], with the difference that no specialized simulation kernel is required. In [14], authors present a distributed SystemC simulation environment for simulations involving geographically distributed intellectual property. Their main focus has been on geographical distribution and not on the simulation performance.

The current research in parallel simulation of TLMs is based on the well-established parallel/distributed discrete event simulation (PDES/DDES) concepts (e.g. [15, 8]). In addition to conservative PDES, there exist *optimistic* approaches (e.g. TimeWarp [13]) where the causality conditions are allowed to be violated. In case of a violation, the simulation must be rolled back in time to ensure correct results. Implementation complexity, and memory and performance costs of such roll-back mechanisms for large system-level models have prohibited their use in parallel TLM simulation. The idea of temporal decoupling can

be traced back to early works in optimistic PDES/DDES. In [19] conservative PDES principles are used to increase the speed of transaction level simulation by avoiding synchronization with the SystemC simulation time as much as possible. The simulation itself however is purely sequential and is performed in a single simulation process. The recent version of the commercial distributed simulation environment Simics [20] claims to utilize temporal decoupling for simulation acceleration. This environment is based on a proprietary simulation kernel and a custom modeling language, with a possibility for integration with SystemC models. However, at the time of this writing, the details of the simulation kernel and SystemC integration are not published to the best of our knowledge.

3 Exploiting Temporal Decoupling for Efficient Parallel Simulation

Similar to [5, 6, 14], we use an application and model dependent number of sequential discrete event simulator processes in parallel. Each simulator is assigned a subset of the modules of a given TLM. The most significant difference of our approach with existing approaches is the synchronization of processes with the simulation time and synchronization of the simulators with each other.

In timed TLMs, timing information is annotated in the processes, for example using a *wait(time)* statement which synchronizes the process with the global simulation time. In sequential DES frameworks, this will result in a process context switch and has a negative effect on the simulation performance. Temporal decoupling is a technique recommended by the recent OSCI TLM 2 standard to reduce the effects of these context switches. In temporally decoupled models, processes have a local time which is allowed to "run ahead" of the global simulation time for a maximum amount of time called a *quantum*, after which the process must synchronize with the global simulation time. In the form proposed by OSCI TLM 2 and implemented in the accompanying library, the deviation between the local time of the process and the global simulation time can become larger than the quantum as the synchronization is left to the processes and no enforcing mechanism exists. This can be seen in the following pseudocode which shows timing annotation portions of a temporally decoupled process:

```
...
t_l = t_l + d_1
...
t_l = t_l + d_2
if t_l > q then
   wait(t_l)
end if
...
```

Here t_l represents the local time offset, q the time quantum and d_1 and d_2 arbitrary time intervals. The local time offset can be incremented beyond a quan-

tum boundary without synchronizing with the simulation time. At some point, the process decides to check the local offset and synchronize with the simulation time ($wait(t_l)$). Upon this synchronization, the local time offset becomes zero and the simulation time may or may not be on the quantum boundary. Our version of temporal decoupling is more strict in the sense that we enforce the processes to synchronize with the global simulation time exactly on *quantum boundaries*. That is, the local time offset of a process is never allowed to get larger than a quantum. For this, a specialized *wait* function is required which is to be used by all processes for timing annotations. The body of this function, which we call *decoupled_wait* is shown below. Here t_l is the local time offset of the process, q is the quantum, n_q is the number of complete quanta to elapse and o_q is the remainder offset.

function *decoupled_wait(d)*
begin
$n_q = \lfloor \frac{(t_l+d)}{q} \rfloor$
$o_q = (t_l + d) - n_q \times q$
if $n_q > 0$ **then**
  $wait(n_q \times q)$
  $t_l = 0$
end if
$t_l = t_l + o_q$
end

Using this function, the timing annotations shown in the previous example will require the following two calls:

...
$decoupled\_wait(d_1)$
...
$decoupled\_wait(d_2)$
...

Assuming that all processes in the model use this function for timing annotation, progress of simulation time in each simulator will be in multiples of the quantum q at all times. This can be exploited to simplify the synchronization of the simulators compared to existing methods (section 2).

There is no need for a central simulation synchronization process. Similarly, collection of local times from individual simulators and broadcasting back their minimum is not necessary. Figure 1 shows how the simulators can be synchronized using collective barrier synchronization. Each simulator executes the processes assigned to it for the duration of a quantum and then waits on a barrier shared by all simulators. After all simulators have reached the barrier, they either proceed to the next quantum boundary or terminate.

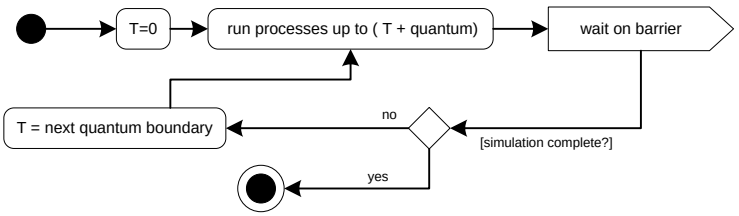

Fig. 1. Simulation progress in each sequential simulator

4 Implementation

As a proof of concept we have developed a parallel TLM simulation framework based on the ideas presented in section 3. The framework is intended for SystemC models and follows the communication interfaces introduced in the OSCI TLM 2 standard [17].

The framework provides a root module class *sc_ dist_ module* from which all modules in a TLM must be derived. This class has an attribute which identifies the simulator to which the module is assigned. This attribute is set upon construction and is specified as a constructor argument. This is used to simplify the task of assigning modules to simulators, and to avoid having different program sources for different simulators. The OSCI TLM 2 standard and the accompanying library are intended for sequential simulation and are not directly usable for parallel simulation. For example, the communication between OSCI TLM 2 modules is performed using *transport* methods which carry a payload, timing and phase information. To enable communication between modules in different simulation processes we have implemented a simple protocol based on a commercial Message Passing Interface (MPI [16]) implementation. The framework provides *stub* modules which handle the translation between the transport calls and MPI calls. For example, to communicate with a remote target *T1* (i.e. target residing in another simulator process), initiator *I1* calls the transport methods of the *initiator-side stub* of that target (*ISS*). This stub translates each transport call to a series of point-to-point MPI communication calls, which are received in the target simulator. Based on the information received in the target simulator, a *target-side stub* of the initiator (*TSS*) calls the transport method of the actual target (*T1*). The return values are transported back to *ISS* and finally to the initiator *I1*. Instantiation and interconnection of stubs are completely hidden from the user by a *binder* component, which handles the details in elaboration time whenever initiator and target sockets are bound.

The root module class *sc_ dist_ module* also provides the *decoupled_ wait* functionality discussed in section 3 in form of an overridden *wait* function. Additionally, it handles the local time offset of the processes. Another motivation for having the root module class was simplifying the migration of existing OSCI TLM 2 models to our parallel simulation framework. Ideally, the migration would require modifying a model to inherit from *sc_ dist_ module* instead of *sc_ module*.

The communication between the simulators required for synchronization is also implemented using MPI, based on collective barrier synchronization.

It should be pointed out that in our framework we use the standard OSCI SystemC simulation kernel and library without modification. In each simulator process, a single SystemC simulation kernel is run, which is controlled according to the simulation progress control presented in section 3. The simulation progress control is implemented as a loop in the user-level *sc_main()* function, and is based on the documented, user-level functions of the OSCI SystemC kernel [11]. For example to proceed in simulation time for a quantum Q, *sc_start(Q)* is called from the simulation control loop.

5 Experimental Results

To evaluate our framework and the proposed parallel simulation approach we have performed several experiments. It should be noted that application and model-dependent factors can greatly affect the degree of speedup achieved when using parallel simulation. The communication pattern between modules assigned to different simulators and the degree of concurrency of computations in the modules are two examples. These application and model specific issues have not been in the scope of this work.

All following experiments were performed on a Linux-based, quad-core Intel SMP simulation host. In the first experiment, we compared the performance of the barrier-based synchronization of the simulators which we proposed in section 3, with the performance of synchronization based on the collection of local times and broadcast of the next global time (section 1). Each simulator was assigned a single active module, which only performed *wait(t)* in a loop. The experiment was performed with 2, 3 and 4 simulators on 2, 3 and 4 cores respectively, with each core running a single simulator. The reduction in the synchronization overhead using our proposed synchronization algorithm was approximately 52% for 2 simulators and 35% for 3 and 4 simulators.

In the second experiment our objective was to determine the maximum achievable speedup in our framework when using 2, 3 and 4 cores. We simulated perfectly parallelizable models with theoretical speed-up limits of 2, 3 and 4 respectively. Each model consisted of a number of identical active modules, which repeatedly performed computations and communicated the result to a passive module. Each such computation and communication sequence was annotated by a (decoupled-)*wait()* function. To account for the worst case, all simulations were performed with the smallest possible time quantum. The models were simulated with different computation to communication ratios and the results are shown in figure 2. Here, each unit of computation load corresponds to computation requiring roughly 5 microseconds on our simulation host.

The third set of experiments were performed to measure the effectiveness of temporal decoupling. Figure 3 shows the effect of temporal decoupling on the simulation speed of a perfectly parallelizable model consisting of 2 active modules

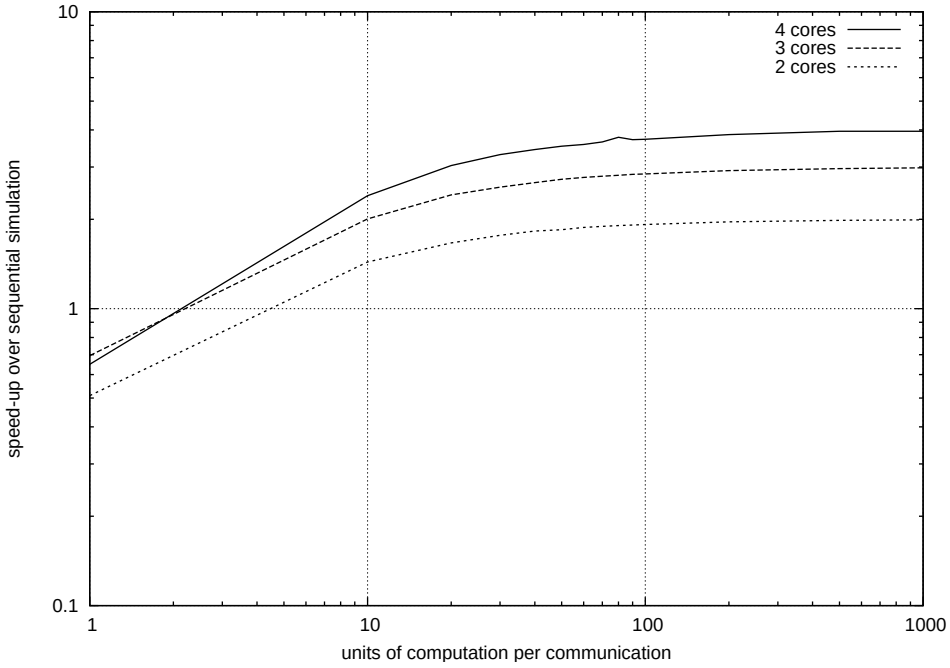

Fig. 2. Achievable speedup under ideal conditions

and a single passive module, simulated on 3 cores. The quantum size was varied between the 1 nanosecond (the minimum possible) and 10 nanoseconds.

6 Conclusion

We have shown that by taking advantage of a certain form of temporal decoupling we are able to reduce the overhead of synchronization between simulators, resulting in a more efficient parallel simulation for a subclass of transaction level models. With a suitable quantum size and with increasing computation-to-communication and synchronization ratio, the theoretical maximum speedup of N can be approached in a simulation on N cores. The current version of the framework can be easily modified to run on clusters of SMP hosts. Optimization of the framework for clusters, automatic analysis of the models for efficient module-simulator assignment and simulation frameworks for massively parallel MPSoCs and NoC-based systems are our planned future work in this direction.

References

[1] Accellera Organization, Inc. *SystemVerilog 3.1a Language Reference Manual*, May 2004.

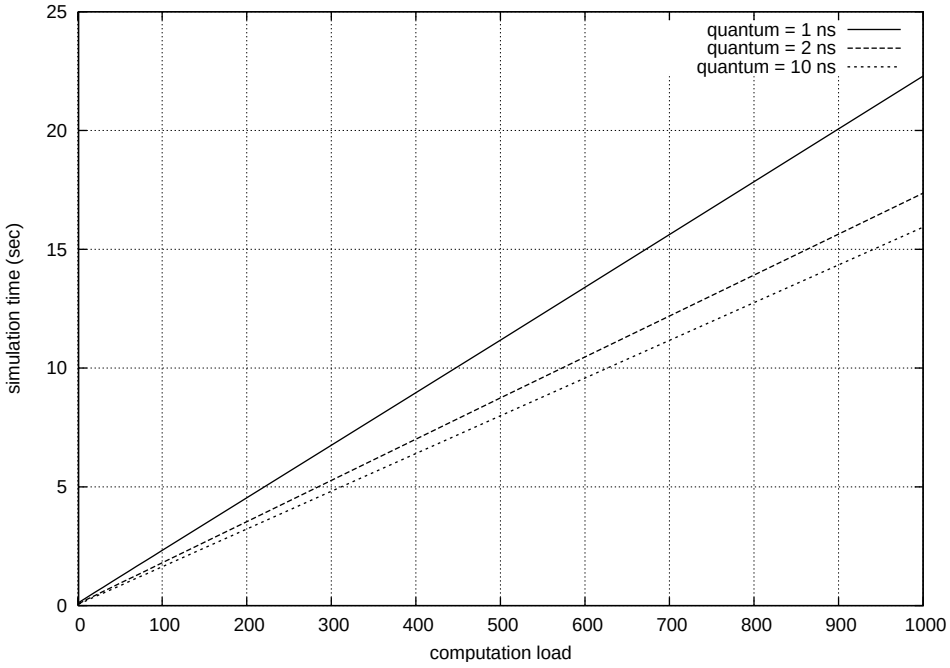

Fig. 3. Effect of temporal decoupling

[2] ARM Limited. *Cycle-Accurate Simulation Interface (CASI) Specification, version 1.1.0*, June 2006.

[3] Mark Burton, James Aldisy, Robert Guenzel, and Wolfgang Klingauf. Transaction Level Modelling: A Reflection on What TLM is and How TLMs May be Classified. In *Proceedings of the Forum on Specification and Design Languages (FDL '07)*, September 2007.

[4] Lukai Cai and Daniel Gajski. Transaction Level Modeling: An Overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, October 2003.

[5] Bastien Chopard, Philippe Combes, and Julien Zory. A Conservative Approach to SystemC Parallelization. In *Proceedings of the International Conference on Computational Science (ICCS 2006)*, May 2006.

[6] Philippe Combes, Eddy Caron, Frédéric Desprez, Bastien Chopard, and Julien Zory. Relaxing Synchronization in a Parallel SystemC Kernel. In *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications (ISPA '08)*, December 2008.

[7] Rainer Doemer, Andreas Gerstlauer, and Daniel Gajski. *The SpecC Language Reference Manual, Version 2.0.* SpecC Technology Open Consortium (www.specc.org), December 2002.

[8] Richard M. Fujimoto. Parallel and distributed simulation. In *Proceedings of the 31st Winter simulation conference (WSC '99)*, 1999.

[9] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., 2006.
[10] Kai Huang, Iuliana Bacivarov, Fabian Hugelshofer, and Lothar Thiele. Scalably distributed SystemC simulation for embedded applications. In *Proceedings of the International Symposium on Industrial Embedded Systems (SIES'2008)*, June 2008.
[11] IEEE Computer Society. *Standard SystemC Language Reference Manual, Standard 1666-2005*, March 2006.
[12] Rauf Salimi Khaligh and Martin Radetzki. Adaptive Interconnect Models for Transaction-Level Simulation. *LNEE 36, Languages for Embedded Systems and their Applications*, 2009.
[13] Yi-Bing Lin and Edward D. Lazowska. A study of time warp rollback mechanisms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1991.
[14] Samy Meftali, Anouar Dziri, Luc Charest, Philippe Marquet, and J. Deskeyser. SOAP Based Distributed Simulation Environment for SoC Design. In *Proceedings of the Forum on Specification and Design Languages (FDL '05)*, September 2005.
[15] Jayadev Misra. Distributed Discrete-Event Simulation. *Computing Surveys*, March 1986.
[16] MPI Forum. MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/.
[17] Open SystemC Initiative (OSCI) TLM Working Group (www.systemc.org). *Transaction Level Modeling Standard 2 (OSCI TLM 2)*, June 2008.
[18] Hiren D. Patel and Sandeep 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 onVLSI (GLSVLSI '04)*, April 2004.
[19] Emmanuel Viaud, François Pêcheux, and Alain Greiner. An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, March 2006.
[20] Virtutech. Virtutech Simics. http://www.virtutech.com/.