Time-Decoupled Parallel SystemC Simulation

Jan Henrik Weinstock*, Christoph Schumacher*, Rainer Leupers*, Gerd Ascheid* and Laura Tosoratto†
*Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany
†Istituto Nazionale di Fisica Nucleare - Sezione di Roma, Rome, Italy
Email: jan.weinstock@ice.rwth-aachen.de

***Abstract*—With increasing system size and complexity, designers of embedded systems face the challenge of efficiently simulating these systems in order to enable target specific software development and design space exploration as early as possible. Today's multicore workstations offer enormous computational power, but traditional simulation engines like the *OSCI* SystemC kernel only operate on a single thread, thereby leaving a lot of computational potential unused.**

Most modern embedded system designs include multiple processors. This work proposes *SCope*, a SystemC kernel that aims at exploiting the inherent parallelism of such systems by simulating the processors on different threads. A lookahead mechanism is employed to reduce the required synchronization between the simulation threads, thereby further increasing simulation speed.

The virtual prototype of the European FP7 project EURETILE system simulator is used as demonstrator for the proposed work, showing a speedup of 4.01× on a four core host system compared to sequential simulation.

I. Introduction

Virtual platforms have proven to be a highly valuable tool in the design process of embedded systems. They allow testing and debugging of device dependent software before a hardware prototype is available and enable design space exploration at early development stages.

Since it became an IEEE standard in 2006, SystemC [1] has emerged as the de-facto industry standard for the modeling of virtual platforms. It is essentially a discrete event simulator implemented as a C++ library with a set of classes and macros that aid in quick virtual platform development.

Modern embedded systems are getting more and more complex. State of the art platforms like the Qualcomm Snapdragon Series [2] usually feature multicore processors with up to four cores and multiple levels of memory. As more complex components need to be simulated simultaneously, the speed of these simulators degrades, threatening the viability of virtual platforms as development tools.

One approach to mitigate this problem is to parallelize the simulation, i.e., distributing the load of simulating multiple processor cores as well as other components among different host threads. Modern workstations are usually equipped with multicore processors making them an ideal host for such a parallel simulation.

A parallel simulator is typically composed of a set of sequential simulations, each one running on its own thread and controlling its own local simulation time. Parallel simulators have to make sure that they behave identically to a sequential version, i.e., it must be guaranteed that all simulation events happen in exactly the same order. Violations of this order are commonly referred to as *causality errors* [3], [4]. Establishing the correct order of events among multiple simulation threads is therefore the paramount objective.

A. Contributions

This paper presents a conservative approach to parallelize the SystemC kernel, which employs a *lookahead mechanism* to efficiently synchronize the event order amongst multiple threads. A prototype simulation kernel called *SCope* has been created from scratch according to the IEEE 1666-2005 SystemC standard. It is capable of running industry-scale virtual platforms like the EURETILE simulator [5] in a multithreaded environment. The key contributions are as follows:

- *Simulation partitioning:* manual and semi-automatic partitioning strategies are outlined to distribute simulation load among multiple threads.
- *Parallel SystemC kernel:* a parallel implementation of the SystemC specification is presented that allows simulation processes to run in a time-decoupled fashion to efficiently exploit parallelism of the modeled system.
- *Parallel TLM-2.0 API:* extensions to the OSCI TLM-2.0 API are introduced that allow race-free and temporal correct communication between different simulation threads and seamless integration with existing models.
- *Case Study:* the prototype simulation kernel *SCope* is tested using the EURETILE embedded system simulator and benchmark results versus sequential and synchronous simulation engines are presented.

The remainder of this paper is structured as follows: First, Section II gives a brief overview about related research efforts in the field of parallel simulation of embedded systems. Section III covers the conservative lookahead synchronization mechanism used for this work. The design of the parallel SystemC kernel is illustrated in Section IV and simulation performance is analyzed in Section V. Finally, Section VI summarizes the results and outlines possible future research.

II. Related Work

The principles of parallel discrete event simulation (PDES) have been well researched for over thirty years. With fundamental work being done by Chandy et al. [3], Fujimoto [4] and Nicol et al. [6], PDES, however, still has not yet found its way into mainstream electronic system level simulation.

In the recent years, there have been several approaches pursuing a parallelization of SystemC. Still, most methodologies impose strong restrictions on the simulator software, e.g., they forbid using parts of the SystemC API, or even require a structural redesign. As a consequence, complications arise when applying those techniques to existing simulators or third party intellectual property, where source code is typically not freely available. Examples for such approaches are described in [7], [8] and [9]. For the domain of SpecC, a parallelization mechanism requiring a statical analysis of the simulator has been proposed in [10].

A synchronous approach to parallel SystemC simulation is presented in [11]. Synchronous techniques, however, are unable to exploit the full parallelism of a simulation, given that they enforce strict time synchronization among all threads. The approach presented in this work attempts to overcome the limitations of synchronous simulation by applying a conservative lookahead technique. Furthermore, existing simulators should be supported without requiring structural redesign or modifications to the source code.

III. Lookahead Simulation Concept

The parallel simulation is split up into n different sequential simulators, each one running on its own thread and managing its own state, e.g., event queues, process lists and simulation time. All state except events is considered to be private to the thread that it was assigned to, e.g., no thread is allowed to run processes from another thread. Only events may be notified from a different thread.

State belonging to a simulator running on thread i will be referred to using subscript i, e.g., the local time of thread i is t_i. Local times of all threads are not allowed to deviate from another by more than the *lookahead* t_{la}. Therefore, a time limit $t_{\mathrm{lim},i}$ is introduced as defined in equation 1.

$$t_i \leq t_{\mathrm{lim},i} = \min_{0 \leq j < n} t_j + t_{\mathrm{la}} \tag{1}$$

Every thread i is allowed to simulate without synchronization until it needs to process an event e_i with a trigger timestamp t_{trigger,e_i} beyond $t_{\mathrm{lim},i}$. If a thread has no events left to trigger before $t_{\mathrm{lim},i}$, it sets its local time t_i to $t_{\mathrm{lim},i}$ and computes a new limit according to equation 1.

An event notification of an event e_i originating from a thread j, with $i \neq j$ is called *remote notification* in this work. This is the designated way for two sequential simulators to communicate without risking causality errors or data races. Remote notifications must be handled carefully, since local times of both threads will likely be different. The time at which j wants e_i to be triggered by i might have already passed in the context of i. In that case, triggering e_i anyway would likely lead to causality errors, since e_i could operate on state that future events might have already modified.

The trigger timestamp of e_i is t_{trigger,e_i}. It depends on the local time t_j of the thread that invokes the notification and a constant time delta t_{notify} imposed by the architecture that is being modeled. This is shown in equation 2 for a remote notification from j.

$$t_{\mathrm{trigger},e_i} = t_j + t_{\mathrm{notify}} \tag{2}$$

To avoid causality errors, remote notifications must be issued sufficiently ahead of time. Since thread i is only allowed to simulate until $t_{\mathrm{lim},i}$ it is sufficient to enforce $t_{\mathrm{trigger},e_i} > t_{\mathrm{lim},i}$. This expression can be further simplified by combining the equations 1 and 2. The result is presented in equation 3.

$$t_{\mathrm{notify}} > t_{\mathrm{la}} \tag{3}$$

Since t_{notify} is usually defined by the simulated architecture, one has to adjust t_{la} to meet the requirements imposed by equation 3. This safely enables time decoupled parallel simulation, allowing cross-thread communication using remote event notifications without incurring causality errors.

IV. Implementation Aspects

This section describes the *SCope* simulation kernel proposed in this work, focusing on the extensions necessary to allow seamless cross thread communication using TLM.

A. Simulator Partitioning

As described in Section III, *SCope* consists of one sequential simulation kernel per worker thread. To avoid simulation state being accessed from multiple threads, each object within the simulation must be assigned to a specific thread. Only this thread is allowed to perform operations on that object, i.e., access members or call methods.

Within the context of SystemC, objects such as modules, events, processes, ports or channels, are organized hierarchically. Objects created within the constructor of another object become the child of this object. Typically, a module will be the parent of a number of events and processes that operate on plain C/C++ data types to model a specific behavior. To avoid data races between such processes, child objects automatically inherit the thread affinity of their parent. However, it is possible to reassign those objects manually.

Objects without a parent are called *top level objects*. Such objects are initially assigned to the main thread ($id = 0$) and can be manually reassigned to a different thread to improve simulator load distribution and increase simulation speed. This can be done either manually during elaboration or by specifying a *mapping file* before the simulator is invoked. Such a file links objects to specific threads using their name.

B. Parallel SystemC

The prototype implementation of the *SCope* SystemC kernel proposed in this work was done according to the specifications set forth in [1]. Its main difference compared to the *OSCI* proof of concept implementation [12] is that it consists of multiple simulation engines, each one running on its own thread. The number of threads and engines to use is fixed throughout the simulation and is specified before the simulator is started.

The master simulator is running on the main thread and invokes the *sc_main* function where the user can instantiate all simulation components and call *sc_start*. This function spawns the additional simulation threads that immediately begin to execute their simulation loops in parallel.

First, each thread invokes the elaboration callbacks such as *end_of_elaboration* for each of its objects. After elaboration, all threads begin executing the simulation loop. However, each thread is only allowed to advance time to $t_{\mathrm{lim},i}$. Once a thread reaches this limit, it checks the local times of the other threads and computes a new limit according to equation 1. The lookahead t_{la} must remain constant throughout the simulation and must be specified either before simulator start or during elaboration.

C. Remote Events

Communication between two objects being simulated on different threads is only allowed using *remote event notifications*. To maintain backwards compatibility with sequential SystemC simulation engines it was decided to implement this feature in a new primitive called *sc_remote_event* rather than augmenting the existing *sc_event*.

The *sc_remote_event* primitive behaves like a regular SystemC event in that it can be notified and processes can be made sensitive to it. However, due to the limitations imposed by equation 3, immediate and delta notifications as well as timed notifications with a notification time of not more than t_{la} are strictly forbidden. As the SystemC standard states for regular events, notifications with a smaller t_{notify} override previous notifications. Event cancellations are allowed as well, but must be stated sufficiently ahead of time, i.e., a remote event can only be canceled up to t_{la} before it is triggered. Canceling a remote event any later is considered an error.

Deciding whether a notification of a remote event should result in triggering the remote event is difficult. The notification could get canceled or overridden by an earlier one that has not yet been stated since the issuing thread might have not yet advanced to that point. To cope with that, each remote event holds a history of notify and cancel requests made to it, associated with the timestamp when the request was made. Each time an event is notified, it is always scheduled to be triggered at the desired time, ignoring whether the notification might have been canceled or overridden before. Using the history it is possible to decide whenever an event is triggered, if it was valid to trigger the event. Only then all sensitive processes are scheduled for execution. Algorithm 1 illustrates this decision process. It takes as input the history of the remote event e_i sorted by request issue time and returns whether it is valid to trigger the remote event at t_i.

Algorithm 1 Trigger decision algorithm for *sc_remote_event*

```
 1: $requests \leftarrow history[t_{\mathrm{trigger},e_i} \dots t_i - t_{\mathrm{la}}]$
 2: $cur\_req \leftarrow 0$
 3: while requests not empty do
 4:     $next\_req \leftarrow$ extract first item from requests
 5:     if next_req is cancel then
 6:         $cur\_req \leftarrow 0$
 7:     else if $cur\_req = 0$ then
 8:         $cur\_req \leftarrow next\_req$
 9:     else if $t_{\mathrm{notify},next\_req} < t_{\mathrm{notify},cur\_req}$ then
10:         $cur\_req \leftarrow next\_req$
11:     end if
12: end while
13: return $t_{\mathrm{notify},cur\_req} == t_i$
```

D. Parallel TLM

While remote events and remote event queues offer a mechanism for race free and temporal correct communication inside a parallel simulator, it is still unsafe to run TLM based simulations in parallel. If initiator and target of a transaction reside on different threads, the callback function of the target is executed within the context of another thread, which is likely to result in races or causality errors. Therefore, such *remote transactions* must be protected. To achieve that, the TLM *simple_target_socket* has been augmented using *remote payload event queues* to allow seamless integration with existing TLM models. While this work focuses on the blocking transport interface, a similar mechanism could be designed for non-blocking or custom TLM interfaces.

Figure 1 shows the modifications done to the simple initiator socket. Whenever the socket receives a *b_transport* call, it first checks whether the call originated from the same thread the target has been assigned to. If this is the case, no synchronization is necessary and the transaction is forwarded as usual. Otherwise the target socket puts the transaction into a remote

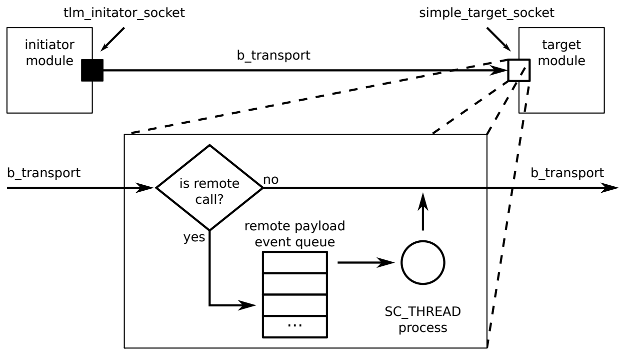

Fig. 1. Local and remote TLM transaction flow

payload event queue. Using remote events this queue notifies a relay process at the time the transaction is supposed to be recieved by the target. This time is defined by the local time of the sending thread plus the delay given in the *b_transport* call. Since this relay process runs on the same thread as the target, it can safely forward the transaction to the target after extracting it from the queue.

E. Limitations

The implementation uses the delay parameter of the *b_transport* call to queue transaction objects to be forwarded to the target. This implies that this parameter must be greater than t_{la} (see equation 3), i.e., the initiator must be able to perform the transaction *ahead of time*.

Currently it is not allowed for the target to advance time while processing the transaction or reporting errors back to the target. The reason for this is, that there is no backward path at the moment that can be used to communicate the extra time taken or errors back to the initiator.

V. Experimental Results

This section will outline experimental results when applying the *SCope* simulation kernel to the European FP7 project EURETILE [5] system simulator.

A. EURETILE simulator

The EURETILE hardware structure consists of a set of computational tiles that are connected in a 3D torus network. Each tile is equipped with a distributed network processor (DNP) [13] that handles communication with neighboring tiles, a RISC processor that serves as the central processing unit for a tile, as well as on-tile memory and peripheral components such as a timer and an interrupt controller. For the performance analysis, the simulator was set up to run 64 tiles in a $4 \times 4 \times 4$ configuration.

The set of applications used for performance evaluation consist of a distributed 32 point FFT application (*FFT*) running on DNAOS [14] and a DNP driver stress test (*presto*), that executes typical network traffic patterns (scatter/exchange/gather).

Preparation for time-decoupled parallel simulation was done in less than one person-day. The simulator was divided into 4 groups of 16 tiles each and each group was assigned to an individual worker thread. The latency of $400\,\mathrm{ns}$ for sending a packet to a neighboring tile enabled a lookahead of $399\,\mathrm{ns}$.

Furthermore it should be noted that most simulation components were not developed for use in a parallel environment. Thanks to the design of the partitioning mechanism of the

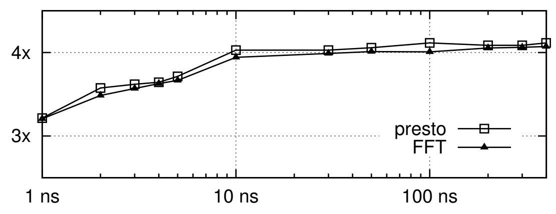

Fig. 2. Speedup with varying lookahead t_{la}

proposed kernel, all components can interact with their peers in a *virtual sequential environment*. This allows the use of mechanisms such as *immediate notifications* and *direct memory interface*.

B. Performance Results

First the parallel performance of the proposed simulation kernel *SCope* is compared to the sequential SystemC implementation *OSCI* [12] and the synchronous parallel SystemC kernel *parSC* [11]. All tests were performed on a quad-core Intel i920 workstation PC, with both parallel simulators using 4 threads. The results are presented in table I.

TABLE I
PERFORMANCE RESULTS

benchmark	*OSCI*	*parSC* (4 threads)		*SCope* (4 threads)	
application	time	time	speedup	time	speedup
presto	0:09:32	0:04:29	2.13×	0:02:19	4.12×
FFT	4:29:27	2:03:54	2.18×	1:07:10	4.01×

Due to the high lookahead (399 ns) relative to the processor model clock cycle time (10 ns), each thread of the time-decoupled simulator can simulate 40 processor cycles in parallel before it needs to synchronize with the other threads, leading to a linear speedup of $4.12\times$ and $4.01\times$ for *presto* and *FFT*, respectively. In comparison, *parSC* can only simulate one delta-cycle before it needs to synchronize with the other threads again. In the context of EURETILE, at least two delta-cycles are needed to execute one processor cycle.

C. Lookahead Analysis

A second experiment illustrates how a reduction of the lookahead t_{la} changes the possible speedup. The *presto* and *FFT* benchmarks were repeated with varying lookahead and the relative speedups were calculated for both benchmarks using the sequential reference execution time from table I. Figure 2 shows the results.

The possible speedup remains almost constant until the lookahead drops below the processor clock cycle time of 10 ns. However, even with a lookahead of only 1 ns, the simulator still maintains a reasonable speedup of $3.2\times$.

As already noted in the previous experiment, the application does not have a huge impact on simulation speedup. Although the *presto* application sends more traffic over the network within short time intervals and thus stressing the parallel TLM implementation much more, simulation speed stays the same.

D. Single-threaded Performance

Finally, the performance of the proposed simulation kernel was analyzed when running with only one thread. The purpose of this experiment is to give an estimation of the quality of the sequential components built for the simulation engine, i.e., the process scheduler and the event notification mechanisms.

TABLE II
SINGLE THREADED PERFORMANCE RESULTS

application	*OSCI*	*SCope* (1 thread)	speedup
presto	0:09:32	0:09:30	1.0035 ×
FFT	4:29:27	4:40:03	0.9622 ×

Table II illustrates that both benchmarks show almost identical performance between the reference SystemC implementation and *SCope*.

VI. CONCLUSION

With rising demands towards fast system level simulators, time decoupled simulation appears a promising field of research. The work in this paper presented a new approach to parallel SystemC simulation using time decoupling. A first prototype implementation called *SCope* showed linear speedups, accelerating the execution of the EURETILE simulator by factor 4.01 on a four core host machine.

Future work includes improving compatibility with existing TLM models, e.g., support for the non-blocking transport interface and enabling a backward communication path to eliminate some limitations of the current design.

ACKNOWLEDGMENTS

This work has been supported by the European FP7 project EURETILE and the German excellence cluster UMIC.

REFERENCES

[1] *IEEE standard SystemC language reference manual*, IEEE Std. 1666-2005, 2006.
[2] Qualcomm inc. snapdragon platform. (Sep 2013). [Online]. Available: http://www.qualcomm.com/snapdragon
[3] K. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Trans. Softw. Eng.*, 1979.
[4] R. M. Fujimoto, "Parallel discrete event simulation," in *Proceedings of the 21st conference on Winter simulation*, 1989.
[5] P. S. Paolucci, I. Bacivarov, G. Goossens, R. Leupers, F. Rousseau, C. Schumacher, L. Thiele, and P. Vicini, *EURETILE 2010-2012 summary: first three years of activity of the European Reference Tiled Experiment*, 2013, arXiv:1305.1459 [cs.DC].
[6] D. Nicol and P. Heidelberger, "Parallel execution for serial simulators," *ACM Transactions on Modeling and Computer Simulation*, Jul. 1996.
[7] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2010.
[8] M. Moy, "Parallel programming with SystemC for loosely timed models: a non-intrusive approach," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013.
[9] R. Sinha, A. Prakash, and H. Patel, "Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs," in *Design Automation Conference, 2012 17th Asia and South Pacific*, 2012.
[10] W. Chen, X. Han, and R. Dömer, "Out-of-order parallel simulation for ESL design," in *Proceedings of the Conference on Design, Automation and Test in Europe*.
[11] C. Schumacher, J. H. Weinstock, R. Leupers, G. Ascheid, L. Tosoratto, A. Lonardo, D. Petras, and T. Grötker, "legaSCi: Legacy SystemC model integration into parallel SystemC simulators," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013.
[12] Accellera Systems Initiative. OSCI SystemC 2.2. (Sep 2013). [Online]. Available: http://www.accellera.org
[13] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. Paolucci, D. Rossetti, A. Salamon, G. Salina, F. Simula, L. Tosoratto, and P. Vicini, "APEnet+: high bandwidth 3D torus direct network for petaflops scale commodity clusters," *Journal of Physics: Conference Series*, 2011.
[14] X. Guerin and F. Petrot, "A system framework for the design of embedded software targeting heterogeneous multi-core SoCs," in *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, 2009.